LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-225936

CCG-LCONE CT Reconstruction Code User and Programmer's Guide

J. A. Jackson

November 8, 2006

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

CCG-LCONE

CT Reconstruction Code

User and Programmer's Guide

Jessie A. Jackson

September 27, 2006

Table of Contents

I. Introduction

A. Overview

This document describes a Computed Tomography (CT) reconstruction code called CCG-LCONE. CCG-LCONE is used to reconstruction objects from projections acquired on a cone beam radiographic system.

This document will describe in brief the theory behind parts of the code, as well as detail the structure of the code, so it will function as both a *User's Guide* and a *Programmer's Guide*.

The Introduction will describe CT in general and cone beam systems in particular. It will explain why CCG-LCONE was developed and give an overview of the design and function.

The following chapters will discuss the various parts of the system, both theory and code structure.

B. Computed Tomography

There is often a need to examine, or characterize, the interior of an object without damaging it; this is referred to as non-destructive testing. One method of non-destructive testing is radiography.

Radiography is the creation of an image by exposing a detector to high-energy particles that have been passed through an object. These high-energy particles can penetrate solid objects, however the object will absorb some of the particles. Since different materials absorb different amounts of particles the total amount of absorption through a given object will be related to the different materials in the object and the distance through each material. When the particles are passed through an object in straight rays the image on the detector will essentially represent a shadow or a profile of the object as viewed from one direction.

The high-energy particles typically used are x-ray and gamma ray photons, and neutrons. The code being discussed here was originally developed for images generated by x-ray photons, so the following discussion will focus on x-rays. However the code has been used successfully on images generated by neutrons.

In a standard x-ray system x-rays pass directly from a ***source*** through an ***object*** to a ***detector***. It is assumed that the source produces x-ray photons that travel in straight lines called beams. X-ray source configurations typically produce beams that are defined as parallel, fan or cone in shape, as shown in Figure I.1.

The detector records the x-rays that reach it, which are those which have not been absorbed by the object. Each point on the detector reflects the path along one beam from the source to detector through the object.

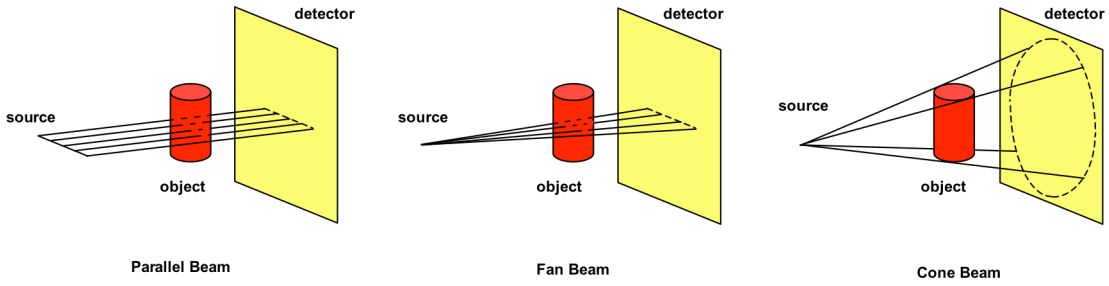


Figure I.1– Standard X-ray System Beam Types

The 2D image formed on the detector is called a ***projection***. Historically film is used as a detector for x-rays. It is essential for analysis purposes that the image at the detector is digitized. Digitizing entails dividing the image into separate equally sized elements called ***pixels*** and giving each pixel a value that represents its intensity on the image and storing those values on a computer. Each radiographic projection will then consist of a digitized array of ***nrays*** x ***nslices*** pixels, where the size of each pixel is ***px*** by ***px***, as shown in Figure I.2. If the detector is film the image on the film can be digitized with standard digital scanning software. It is however more efficient and accurate to use a detector that collects the x-rays digitally directly.

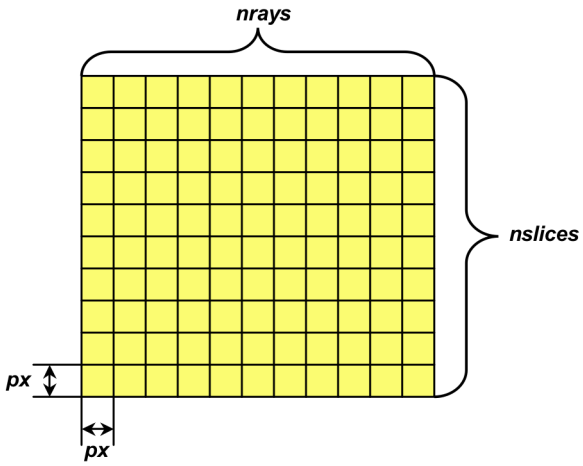


Figure I.1– Digitized Radiographic Projection

An x-ray of an object provides a limited amount of information; a 2D view of a 3D object. However if the object is rotated and a series of digital projections are collected from a number of angles a 3D digital image of the object can be created. The process of creating, or reconstructing, a 3D image

from a series of 2D digital images is called ***Computed Tomography (CT)***. Figure I.3 shows the setup for a cone-beam CT scan.

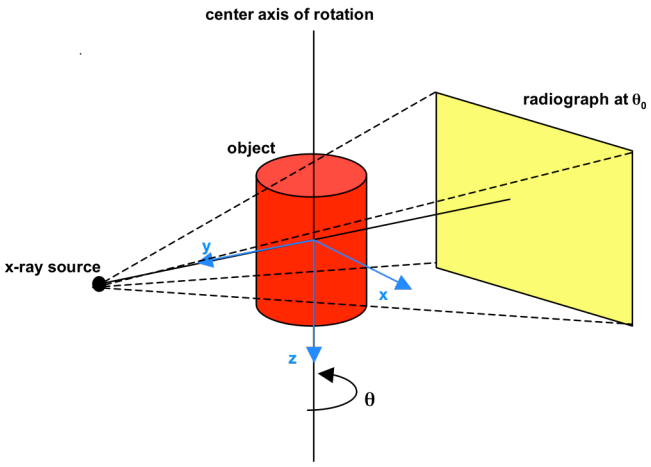


Figure I.2 – Cone Beam X-Ray CT Acquisition System

The resulting series of ***radiographs*** is seen in Figure I.4. The variable θ indicates the rotational angle at which a radiograph is acquired. ***nangles*** represents the total number of angles, or views, that are scanned. Images can also be formed by taking the values of each radiograph at a particular location along the z-axis for all the radiographs. This results in a 2D image along the x and θ axes, which is called a ***sinogram***.

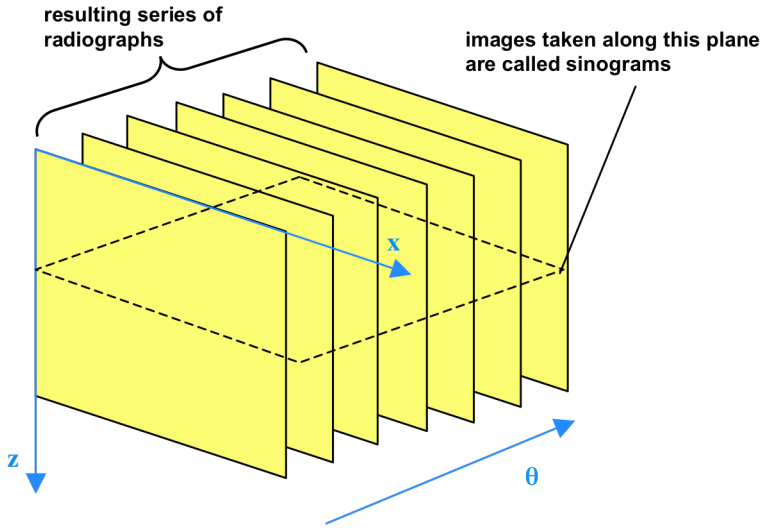


Figure I.3– Radiographic Projections from a CT Scan

The objects that are created with CT reconstruction will be in the form of a digitized 3D volume as shown in Figure I.5. The volume is divided into equally sized 3D elements called ***voxels***. Each side of a voxel is ***vx***, and the number of voxels along the respective axes are ***nx*** x ***ny*** x ***nz***. A z-plane image of the reconstruction is referred to as a ***slice***. The value of a voxel represents a physical property of the volume of the object corresponding to the voxel location.

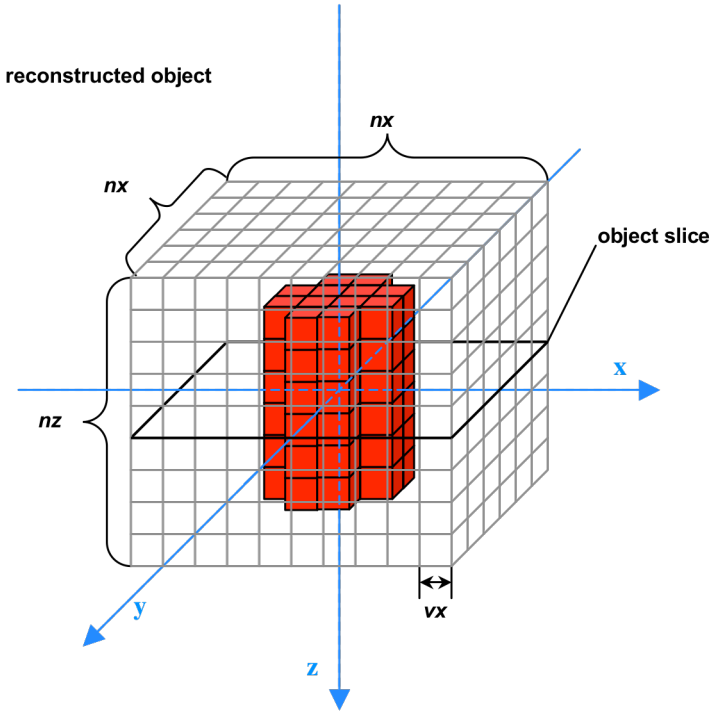


Figure I.4– CT Reconstruction

There are many CT techniques to create the 3D images. These methods consist of processing the projection data; by filtering, scaling and/or FFTs etc, and then essentially backprojecting and summing the data. These methods include Filtered Backprojection (FBP), Convoluted Backprojection (CBP), and the Feldkamp Algorithm.

C. CCG-LCONE

1. Cone Beam Systems

As seen in Figure I.1 parallel and fan beam systems produce beams only on one z-plane, or slice. Therefore only the pixels along one row of the detector, for all angles, are needed to reconstruct one slice, and the slices can be reconstructed independently of each other. In this situation only

enough processor memory is needed for the row of pixels at all angles and the object slice. Processing speed is generally reasonable even for large detectors. Also the slices can be processed in parallel, further reducing the overall processing time.

Cone beam systems are more complicated. In a cone beam system a ray-path can go from the source to a pixel off the z-plane and these ray-paths will a include number of object slices. FBP and CBP are not designed to work for this case at all. The Feldkamp algorithm was developed for cone beams, however for the sake of speed and memory it makes certain simplifying assumptions, as a result it works reasonably well for small cone angles but it is less effective for larger cone angles.

In order to develop a method that would produce a more accurate reconstruction for large cone angles an iterative optimization cone beam system was designed. A general iterative optimization system is shown in Figure I.6.

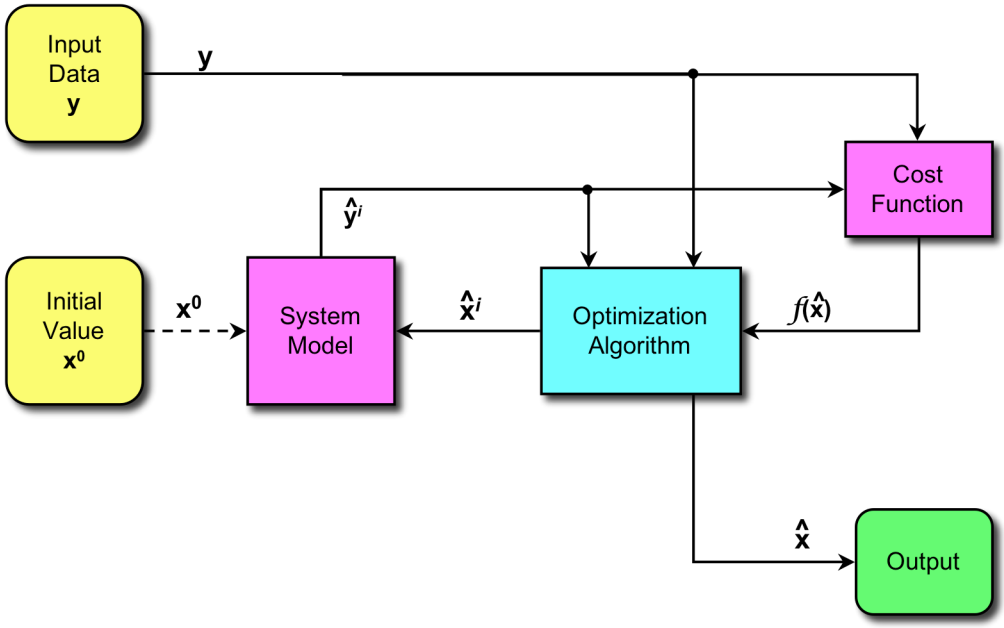


Figure I.5– General Iterative Optimization System

The iterative process starts with an initial guess $\mathbf{x}^0$ for the value of the $\mathbf{x}$. Then the system model is then used to calculate $\hat{\mathbf{y}}^i$. The input data $\mathbf{y}$ and the calculated value $\hat{\mathbf{y}}^i$ are used to calculate the cost function, $f(\hat{\mathbf{x}})$, a measure of the difference between the measured and estimated values of $\mathbf{y}$. The values $\mathbf{y}$ and $\hat{\mathbf{y}}$, and the result of the cost function are input into the Optimization Algorithm. The Optimization Algorithm searches for a value of $\hat{\mathbf{x}}$ that reduces the cost function further and then process is repeated until an appropriate stopping test is satisfied. The system

model and the cost function are dependent on the application. An optimization algorithm is independent of the system model and can be used for many different applications.

In order to use this method on the cone beam problem a mathematical system model must be determined, an appropriate cost function selected and an effective optimization algorithm determined. The following sections will describe these elements in brief as they pertain to the cone beam system.

2. System Model – LCONE

a. Absorption

The mathematical model used for cone beam reconstruction is based on the physics of the absorption of the beam along the ray-path. The quantitative measure of the absorption factor of a material is μ, ***the linear attenuation coefficient***. Different materials will have different values of μ.

If I_o is the number of x-ray photons entering an object on a given path and I is the number of x-rays exiting, then the attenuation of the x-rays is represented by Equation I.1.

$$I = I_o e^{-\int \mu(\ell) d\ell} \qquad [I.1]$$

The term $\int \mu(\ell) d\ell$ is a line integral where $\mu(\ell)$ represents the value of μ at each location along the line the x-ray photon travels through an object. Even though the path an x-ray takes through an object can be complicated by scattering of various kinds, the model being considered here will take an idealized approach in which it will be assumed that the x-rays travel in a straight line through an object.

In the case of a CT reconstruction the object voxel values will represent the linear attenuation coefficient. This means that if an x-ray beam traces a path through the object the sum of the length through each voxel along the path multiplied by the value of that particular voxel will represent the attenuation of the x-ray along that path. Let *i* represent the voxels along the beam path, and then Equation I.2 can represent the attenuation along one beam for the digitized model.

$$I = I_o e^{-\sum_i \mu_i \ell_i} \qquad [I.2]$$

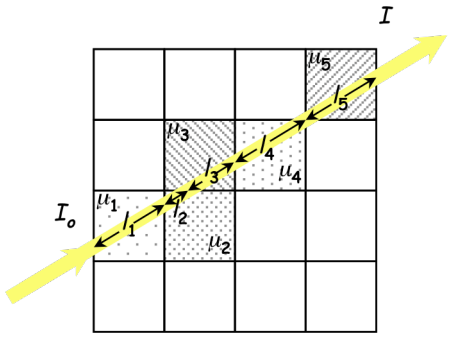
Equation I.2 can be linearized by taking the natural logarthim of both sides of the equation.

$$\ln(I) = \ln(I_o) \cdot \left(-\sum_i \mu_i \ell_i \right) \qquad \text{[I.3a]}$$

or

$$\ln(I/I_o) = \left(-\sum_i \mu_i \ell_i \right) \qquad \text{[I.3b]}$$

The 2D representation of this ray sum is shown in Figure I.7.



$$-ln\left(I/I_0 \right) = \mu_1 \ell_1 + \mu_2 \ell_2 + \mu_3 \ell_3 + \mu_4 \ell_4 + \mu_5 \ell_5$$

Figure I.6– 2D Ray Sum

b. Projection Vector

The value $ln(I/I_0)$ is determined for each pixel on the detector sized ***nray*** x ***nslices*** for each of the ***nangles*** radiographs. This value could be represented as

$$q_{ijk} = ln\left(I_{ijk} / I_{0ijk} \right) \qquad \text{where} \quad i = 0, 1, 2, \ldots \ (\textbf{\textit{nrays}}\text{-}1) \qquad \text{[I.4]}$$

$$j = 0, 1, 2, \ldots \ (\textbf{\textit{nslices}}\text{-}1)$$

$$k = 0, 1, 2, \ldots \ (\textbf{\textit{nangles}}\text{-}1)$$

However if

$$\textbf{\textit{nr}} = \textbf{\textit{nrays}} \bullet \textbf{\textit{nslices}} \bullet \textbf{\textit{nangles}} \qquad \text{[I.5a]}$$

and

$$p = i + j \bullet \textbf{\textit{nrays}} + k \bullet \textbf{\textit{(nrays} \bullet \textbf{\textit{nslices)}}} \qquad \text{[I.5b]}$$

The (***nr*** X 1) vector **y** can be defined as

$$\mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{\mathbf{nr}} \end{bmatrix} \qquad \text{[I.6a]}$$

where

$$y_p = ln\left(I_p / I_{0p}\right) \qquad for \qquad p = 0,1,2,\ldots(\mathbf{nr}-1) \qquad \text{[I.6b]}$$

So a three-dimensional set of data is reduced to a one-dimensional vector. This is shown in Figure I.8.

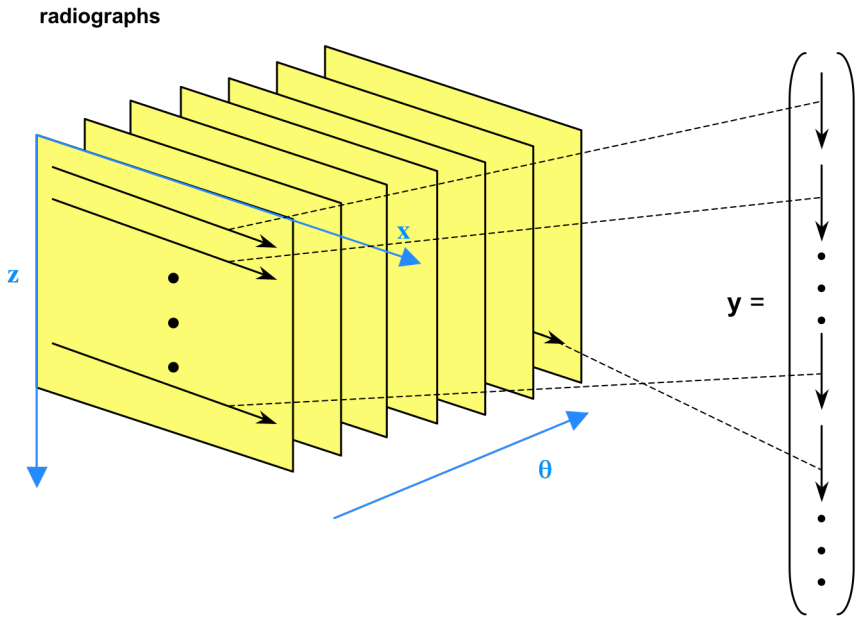


Figure I.7 - Projection Vector **y**

c. Object Vector

The same process can be applied to the object. The object also has three-dimensions, ***x***, ***y***, and ***z*** with sizes ***nx***, ***ny***, and ***nz*** respectively. The object voxels represent the linear attenuation coefficient so the voxel values would be

$$\mu_{ijk} \qquad \text{where} \quad i = 0, 1, 2, \ldots \ (\mathbf{nx}\text{-}1) \qquad \text{[I.7]}$$

$$j = 0, 1, 2, \ldots \ (\mathbf{ny}\text{-}1)$$

$$k = 0, 1, 2, \ldots \ (\mathbf{nz}\text{-}1)$$

So if

$$\boldsymbol{nc} = \boldsymbol{nx} \bullet \boldsymbol{ny} \bullet \boldsymbol{nz} \qquad \text{[I.8a]}$$

and

$$v = i + j \bullet \boldsymbol{nx} + k \bullet (\boldsymbol{nx} \bullet \boldsymbol{ny}) \qquad \text{[I.8b]}$$

The (***nc*** x 1) vector **x** can be defined as

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{\boldsymbol{nc}} \end{bmatrix} \qquad \text{[I.9a]}$$

where

$$x_v = \mu_v \qquad \textit{for} \qquad v = 0,1,2,\ldots(\boldsymbol{nc} - 1) \qquad \text{[I.9b]}$$

Again a three-dimensional set of data is reduced to a one-dimensional vector, which shown in Figure I.9.

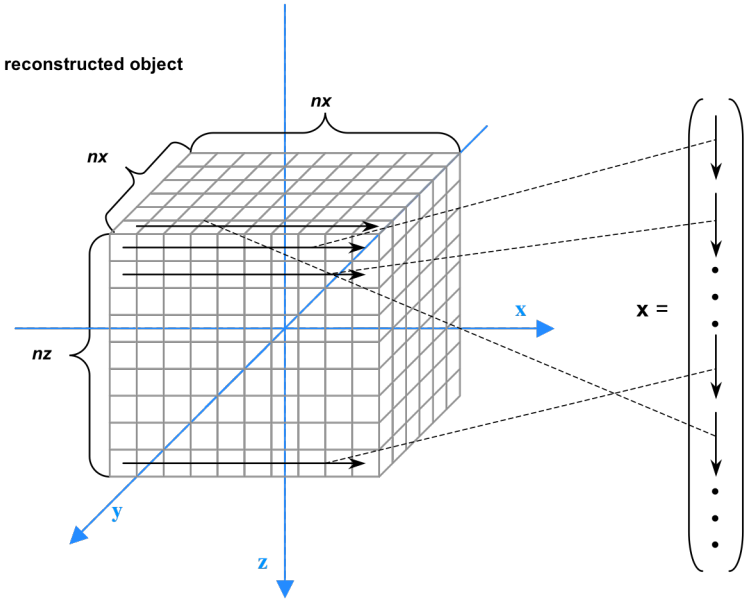


Figure I.8 – Object Vector **x**

d. System Matrix

The relationship between the known projection values, $ln(\boldsymbol{I/I_0})$, and the unknown voxel values, μ, are given by Equation I.3b. Since the end points of the ray-path are the source location

and a pixel location, and these values are known the values of ℓ_i in each voxel can be calculated. Therefore the ℓ_i values are also known.

Equation I.3b can be rewritten in terms of the vectors defined in the two previous sections. For a projection value at a given pixel *p*, which essentially defines the path, the equation will be

$$y_p = \sum_{v=0}^{\mathbf{nc}-1} \ell_v \cdot x_v \qquad \text{[I.10]}$$

Obviously for a single ray-path many of the values of ℓ_v will be zero since the ray-path only touchs a small number of the total voxels. The value of ℓ_v for a given value of *v* will vary for each ray-path, or value of *p,* so the ray-path length through a voxel is dependant on the value of both *v* and *p.* Therefore the variable a_{vp} is defined as the length of a ray-path defined by pixel *p* through a voxel *v* and Equation I.10 becomes

$$y_p = \sum_{v=0}^{\mathbf{nc}-1} a_{vp} \cdot x_v \qquad \textit{for} \qquad p = 0,1,2,...\left(\mathbf{nr}\text{ - }1\right) \qquad \text{[I.11]}$$

The variables a_{vp} will be the elements of a matrix **A**, where **A** is ***nc*** x ***nr,*** and Equation I.11 becomes

$$\mathbf{y = A\ x} \qquad \text{[I.12]}$$

Since the matrix **A** represents the position of the source, object and detector, or basically the physical nature of the system, it will be called the **system matrix**. Equation I.12 is a mathematical system model of a linear cone beam so it will be referred to a **LCONE**.

With the problem presented in the form of **y = A x** the reconstruction could be simply an inverse problem, **y** and **A** are known and **x** is the object solution and is not known, so

$$\mathbf{x = A^{-1}\ y} \qquad \text{[I.13]}$$

A however is very large, for example a 512 x 512 detector with 360 rotational views and a 512 x 512 x 512 object will have a system matrix of 48,000 terabytes in size. Obviously a matrix this size cannot be inverted which is why the iterative optimization method is used.

Calculating the lengths along the ray-paths to determine the matrix elements a_{pv}, can be a time intensive proposition and there are many methods that can be used. Over the development cycle of this project three different methods have been used. Two methods allow a very flexible placement of rays and rotation angles. They are known as **jcone** and **ncone**. These methods are based on the rectangular structure of the output object. These

two methods generate some uncorrectable side effects. A third method, **pcone**, solves for the object in a polar coordinate system and converts the result into the needed rectangular form. This method is much faster and the side effects present in the rectangular methods can be corrected, however the acquired views must be equally spaced rotationally, though they do not have to encompass the entire 360˚ range.

3. Cost Function – Least Squares

A cost function compares an actual measured value with a calculated value. In order to use an optimization search a cost function must be determined. The success of the optimization procedure depends in large part on the specification of cost function.

It can be shown that the maximum likelihood approach leads to a least-squares fitting criterion when the predominant source of random error is Gaussian measurement noise which is independent and identically distributed over all the detectors. This is a reasonable assumption for this situation. Therefore the cost function will be

$$\mathcal{L}\left(\hat{\mathbf{x}}\right) = \frac{1}{2}\left(\hat{\mathbf{y}}\left(\hat{\mathbf{x}}\right) - \mathbf{y}\right)^2 \qquad \text{[I.14]}$$

It is noted that the cost function can also be written in terms of the residual, $\Delta\mathbf{y}$, the difference between the measured and calculated input data

$$\Delta\mathbf{y} = \left(\hat{\mathbf{y}} - \mathbf{y}\right) \qquad \text{[I.15a]}$$

Therefore

$$\mathcal{L}\left(\hat{\mathbf{x}}\right) = \frac{1}{2}\left(\Delta\mathbf{y}\right)^2 \qquad \text{[I.15b]}$$

4. Optimization Algorithm – CCG

The purpose of an optimization algorithm is to find the minimum, or maximum, of a function as accurately and efficiently as possible. In order to understand how an optimizing algorithm works consider the cost function $\mathcal{L}(\hat{\mathbf{x}})$. $\mathcal{L}(\hat{\mathbf{x}})$ is a multi-dimensional concave surface whose minimum is being sought. The gradient of the function, $\frac{\partial\mathcal{L}\left(\hat{\mathbf{x}}\right)}{\partial\hat{\mathbf{x}}}$, is also a vector. The negative of the gradient at a particular $\hat{\mathbf{x}}$ will point to the steepest slope of the function at the location. A new value of $\hat{\mathbf{x}}$ can be determined along this direction which decreases the value of $\mathcal{L}(\hat{\mathbf{x}})$. Then at the new position, or value of $\hat{\mathbf{x}}$, a new direction, gradient, is calculated and the next value of $\hat{\mathbf{x}}$ is determined. This optimization method is called *steepest descent*. This however is not the most efficient method. This search can end up going in the same direction repeatedly and many matrix multiplications are needed. A method has been developed to determine a set of search directions so that all the directions are conjugate, or orthogonal. Conjugate directions are

considered to be non-interfering directions and each direction is unique. A set of conjugate directions have the special property that minimization along one direction is not spoiled by the subsequent minimization along another direction in the set. This method is called *Conjugate Gradient* (CG).

The *Constrained Conjugate Gradient* (CCG) method allows the addition of constraints on the value of the vector $\hat{\mathbf{x}}$ to the CG optimization method. The user supplies vectors of minimum and maximum values for $\hat{\mathbf{x}}$. The effect of adding constraints to CG is to reduce the degrees-of-freedom in the problem and improve the likelihood of finding an effective solution.

The CCG method has proved to effective for CT problems. It is slow and uses a great deal of memory. However it is very flexible and accurate.

An important note is that the gradient of the least-squares cost function will be dependent upon the residual $\Delta\mathbf{y}$, not the individual terms $\mathbf{y}$ and $\hat{\mathbf{y}}$. This is reflected in the RECON/CCG-LCONE Model as seen in Figure I.10.

5. RECON

A number of years ago the NDE section at LLNL developed a suite of reconstruction codes called RECON. Included in the system were routines for reading and writing parameter files, known as SCT files, and routines for reading and writing data files in the VIEW file format. This system was run on an SGI UNIX system. As this system became obsolete part of the RECON system was refreshed and ported to an OSX/UNIX system. The parameter file and data I/O sections were maintained and the CBP and Feldkamp routines were also ported. CCG-LCONE was developed within this system. Figure I.10 shows an overview of the relation between RECON and CCG-LCONE and actual CCG-LCONE model.

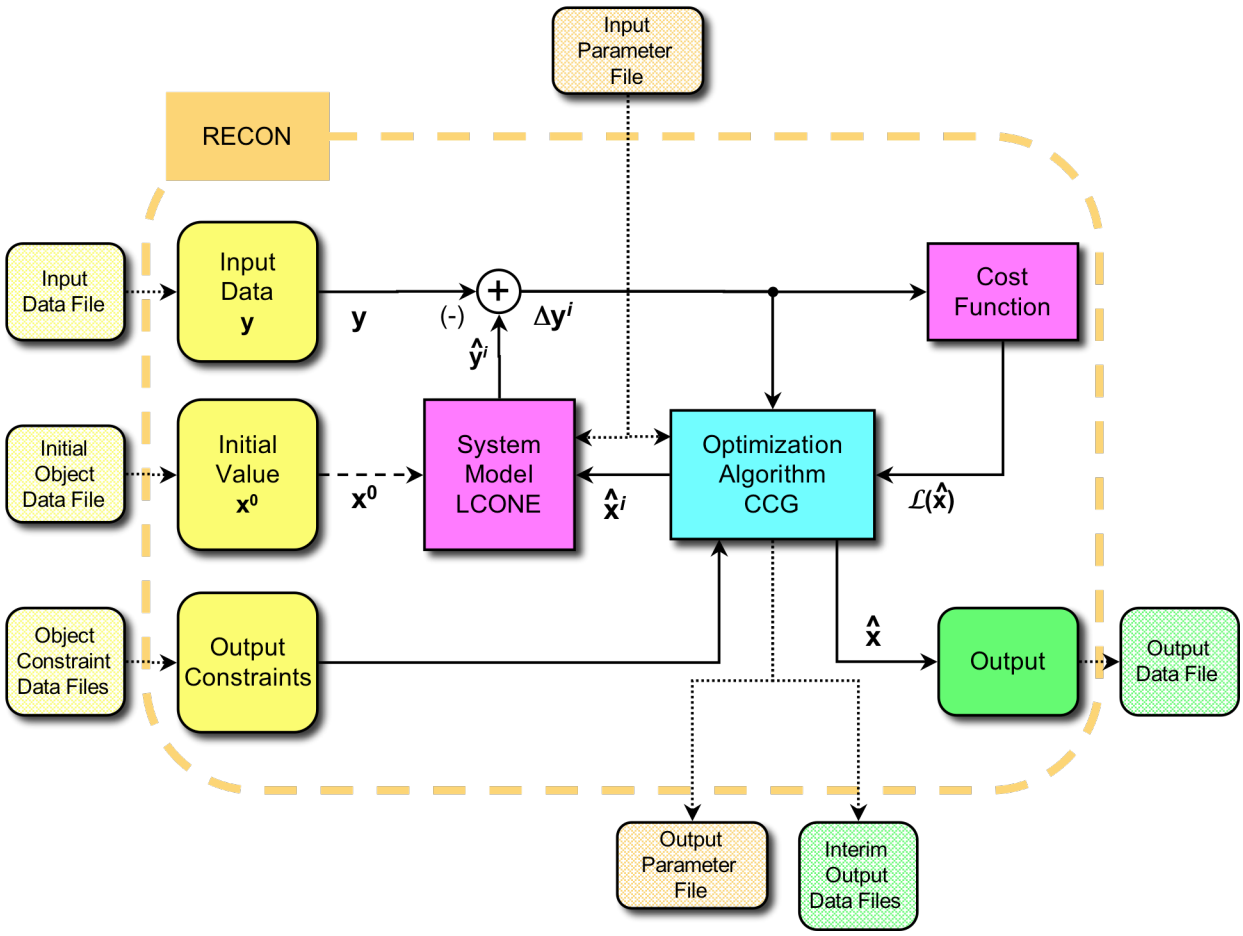


Figure I.9 – RECON - CCG-LCONE Model

D. Cone Beam Reconstruction Results Comparison

A brief comparison of simulated results is presented here to give an idea of the effectiveness of CCG-LCONE.

The goal is to examine a fairly large cone angle; since CCG-LCONE is slow for large objects a tall narrow object was created. The object has 350 slices of 128 x 128 voxels. The object bottom is placed at z = 0, so it is in only half of the cone. The object has a cylinder of value 10, radius 40, and height of 360 slices placed in the center of the object on the bottom slice of the object. There is a 4 x 4 voxel square of value 40 placed on 8 slices. Starting from the top of the object these squares are started on slice 61 and placed every 40 slices after that. Other pertinent information is

pixels 2mm x 2mm	sod 2000 mm
voxels 1mm x 1mm x 1mm	sdd 4000 mm
360 views	

The object, slice and simulated test setup is shown in Figure I.11.

The results are also shown in Figure I.11. At four different cone angles the reconstruction results are shown for CBP, Feldkamp and CCG-LCONE. CCG-LCONE obviously out performs CBP and Feldkamp.

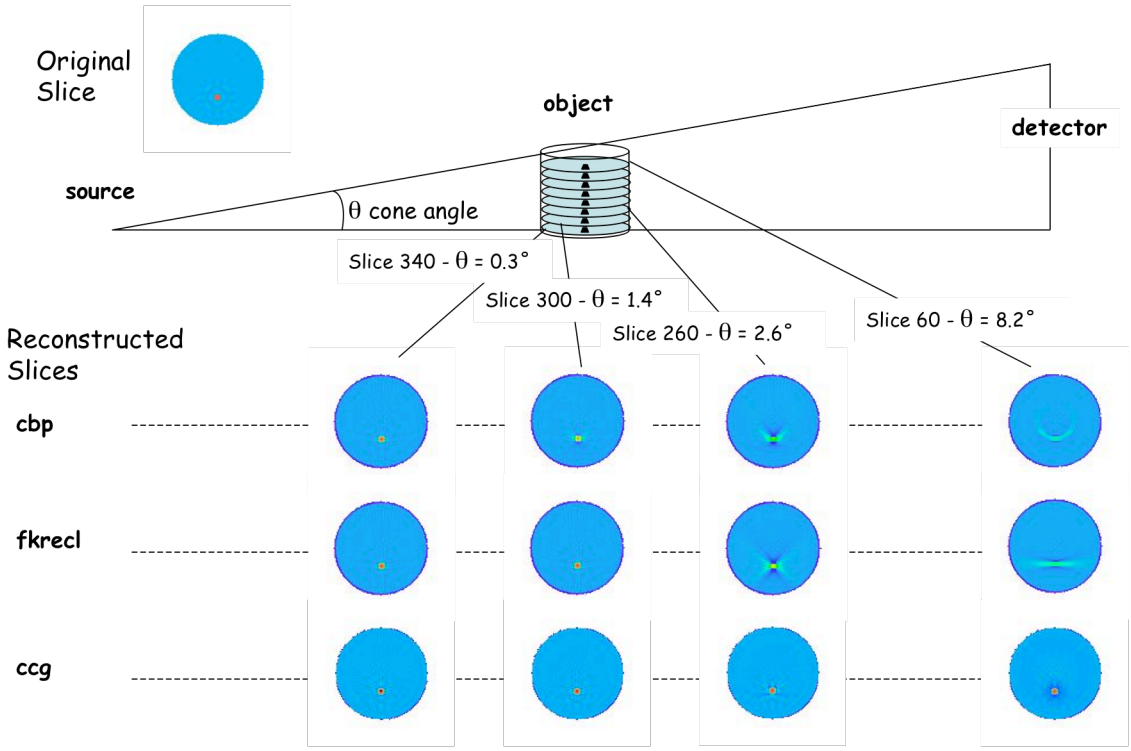


Figure I.10 – Simulated Cone Beam Reconstruction Comparison

II. RECON

A. CT Parameters

Any reconstruction algorithm depends on the acquired projection data but it also needs other information such as pixel and voxel size. All the non-data information are referred to as ***CT Parameters***. CT Parameters can be input via a text file, the command line, or as a user edit during start up. The extension of the test file containing CT Parameters information is '.sct' so this text file is called an ***SCT File***.

CT Parameters have specifically defined names. For example the voxel x-axis size is '*pxsize*'. The names are defined in the application function '***init_ctp'*** described in section II.C.3.1.

1. SCT File

The format of an *SCT File* line is

-ctparametername value

For example

-pxsize 0.02

A comment line in the *SCT File* starts with '!'.

! This is an SCT File

An example of an *SCT File* is shown in Appendix A.

B. Code Design

The RECON suite of codes are written in C and run on a UNIX/LINEX operating system. The directory structure, build scripts and Makefiles are all very simple and straight forward to allow of ease of maintenance and possible future porting of the codes to other systems.

1. Overview

The RECON system is not built as one large code with multiple application options. It is structured to have one executable for each application, applications being a reconstruction algorithm or other processing. (In the original version there was a start up code that would allow users to select which application to use, it would then fork off the selected application.) Each application executable is built from codes particular to it and from a number of shared codes. In

fact all applications share the same ***main.c***. Figure II.1 shows the relationship of the shared codes to the application codes and executables.

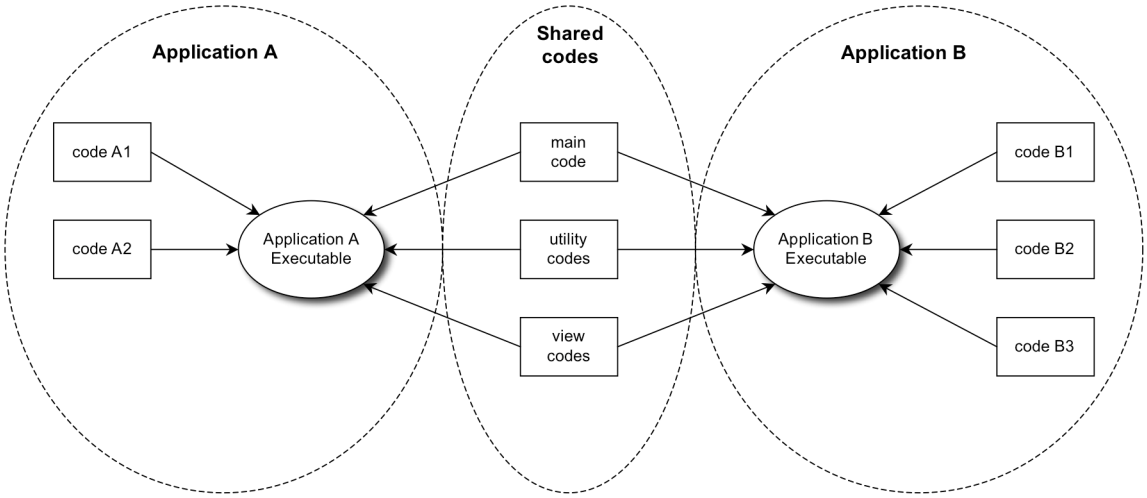


Figure II.1 – RECON Shared and Application Codes

2. Directory Structure

The top level RECON directory structure has the sub-directories **bin** and **src.** The top level also contains shell scripts with the build information for each application, ***build.cbp***, ***build.lpcone***, etc. The directory structure in shown in Figure II.2.

a. bin Directory

The ***bin*** directory contains the application executables**.**

b. src Directory

The **src** directory contains the source codes and headers. It is divided into a number of subdirectories.

The shared non-application codes are separated by function into three directories: **main, util** (utility), and **view**.

The **main** directory has only the code ***main_nw.c***. this is the main program for all applications. The ***nw*** indicates this is a non-window (no GUI) version.

The **util** directory contains codes for reading command line inputs, reading and writing SCT files, error checking, and low-level routines for binary data file I/O.

The **view** directory contains codes for VIEW data file I/O.

The directory **apps** contains the sub-directories for each application and some shared application files. **apps-util** contains routines that are used by multiple routines, such as *fft842.c*. **cbp** and **fkrecl** are directories for the applications. Since CCG is an optimization algorithm that can be used by mutliple applications the CCG codes are contain in their own directory, **ccg**. The directory **lcone** contains codes and directories for the various linear cone beam models. The three ray-path models are in the directories **jcone**, **ncone**, and **pcone**.

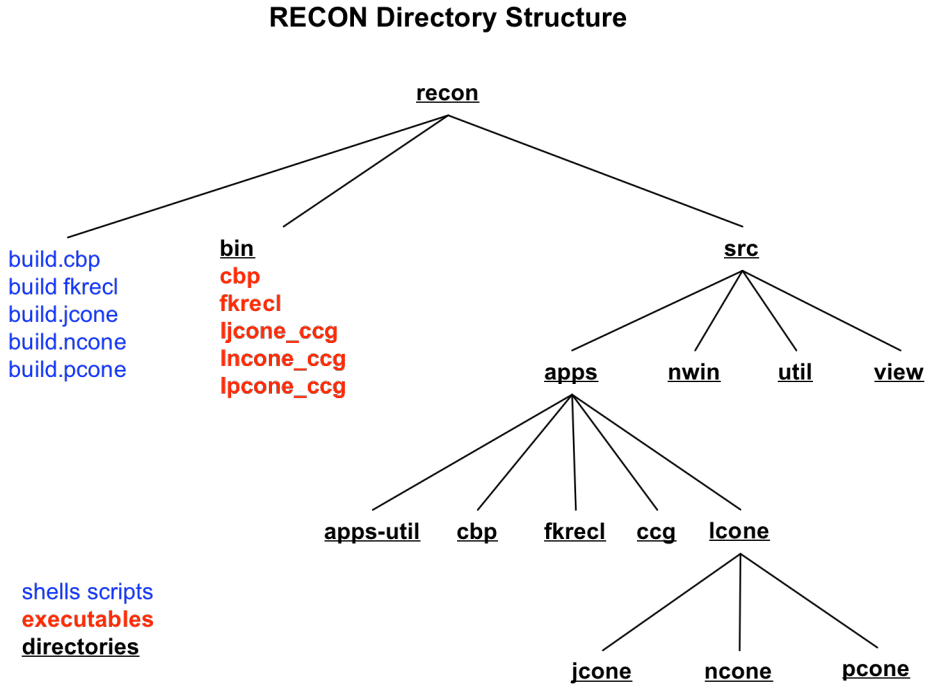


Figure II.2 – RECON Directory Structure

3. Build Scripts/Makefiles

a. Build Scripts

The *build scripts* are located on the top level of the **recon** directory. These *build scripts* are quite simple. There is a *build script* for each application. Each script starts by setting the paths for the directories of the source codes needed by the application. Then the script changes directory to each needed directory and calls the *Makefile* located in the directory. The last directory called is the one containing the application.

Programming Note: The user needs to set a line in each build script to the location of recon directory, for example:

setenv RECON /Users/bob/recon

b. Makefiles

The *Makefiles* are located in each directory with source codes. In the non-application directories the *Makefile* simply compiles each of the *C* codes. The *Makefile* in each application directory compiles the application code and links it to all the necessary codes to build the application, and finally copies the executable to the **bin** directory. The *Makefiles* contain the dependencies to the header files.

4. Code Structure

This version of RECON has only a few system global variables, all the other information and data are stored and passed by structures. The structures are defined in ***header*** *(.h)* files. In designing each application the idea is to pass only the needed structures, both shared and application specific, to the applications codes. As mentioned earlier every application uses the same ***main()*** function (found in *main_nw.c*). This ***main()*** calls the shared functions necessary to read command line inputs, input *SCT Files*, and edit *CT Parameters*. It includes the application processing by calling functions of specific names that every application must have. These functions are ***init_ctp()***, ***init_proc()***, ***proc_app()***, and ***close_proc()***. The applications can have other functions but they must have these functions. The program flow is shown in Figure II.3.

The functions called by ***main()*** are passed certain structures. The structures are *CTvariables*, *CTparams*, *gen_info_struct* and *process_struct.* The application defines the elements in the *CTvariables* and *process* structures. ***main()*** has the pointers to the these structures and it passes the pointers to the application functions. However neither ***main()*** or the other the shared RECON routines know anything about the actual elements within the application structures. The other two structures, *CTparams* and *gen_info_struct* are defined by the shared codes and are always the same. The values in these structures can be set by both the application and shared codes. Figure II.4 shows the relationship between the shared and application functions and the shared and application structures.

a. Error Checking – *error_check(), error_set(), error_reset* – *error_check.c*

The header file ***recon_err.h*** contains predefined error codes and error messages. If an error condition has occurred the function ***error_check(errnum)*** is called with the appropriate error code. ***error_check()*** will print the error message associated the error code. Errors with codes less than 64 are designated FATAL. These errors will cause the program to halt.

The function ***error_set(fname)*** adds "/fname" to the global string *g_errloc*. Each function starts by calling ***error_set()*** with the name of the function. The function ends by calling ***error_reset*** which removes the last "/" and everything after it. The result of this is that the global variable string *g_errloc* maintains the function path to the current location in the code. This information is useful when printing error messages.

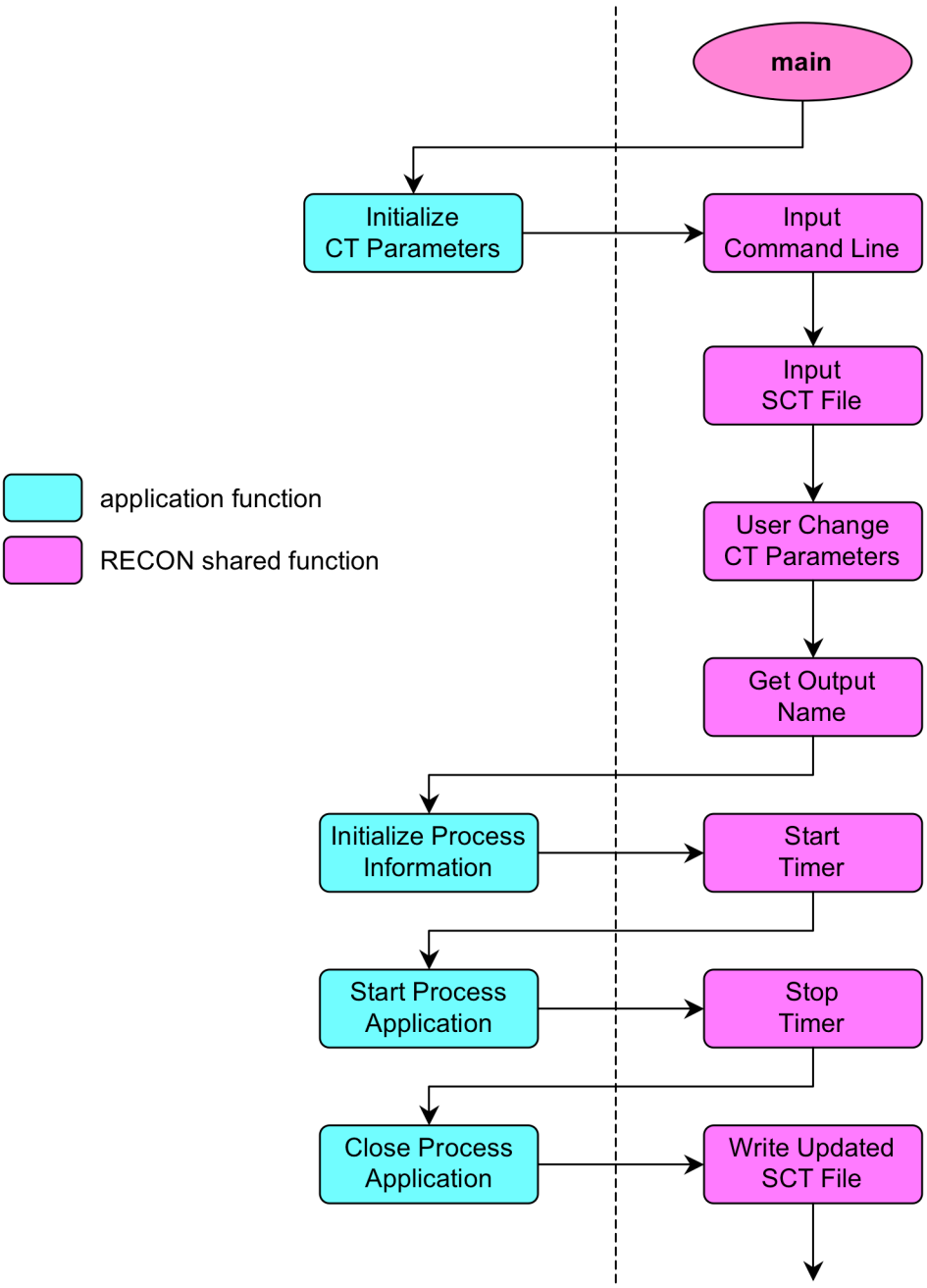


Figure II.3 – RECON Program Flow

b. Data I/O – view Directory

The **view** directory contains functions that handle data files in the VIEW file format. The I/O code was designed to allow other data file formats to be used. The I/O functions the application codes call to input and output data are not VIEW specific. If another file format is needed a set of functions to handle the new format could be developed in its own directory and then linked to the application code in the *Makefiles* and *buildscripts*. The new functions just need to have high level functions of the same name as the ones currently used by the high level VIEW functions.

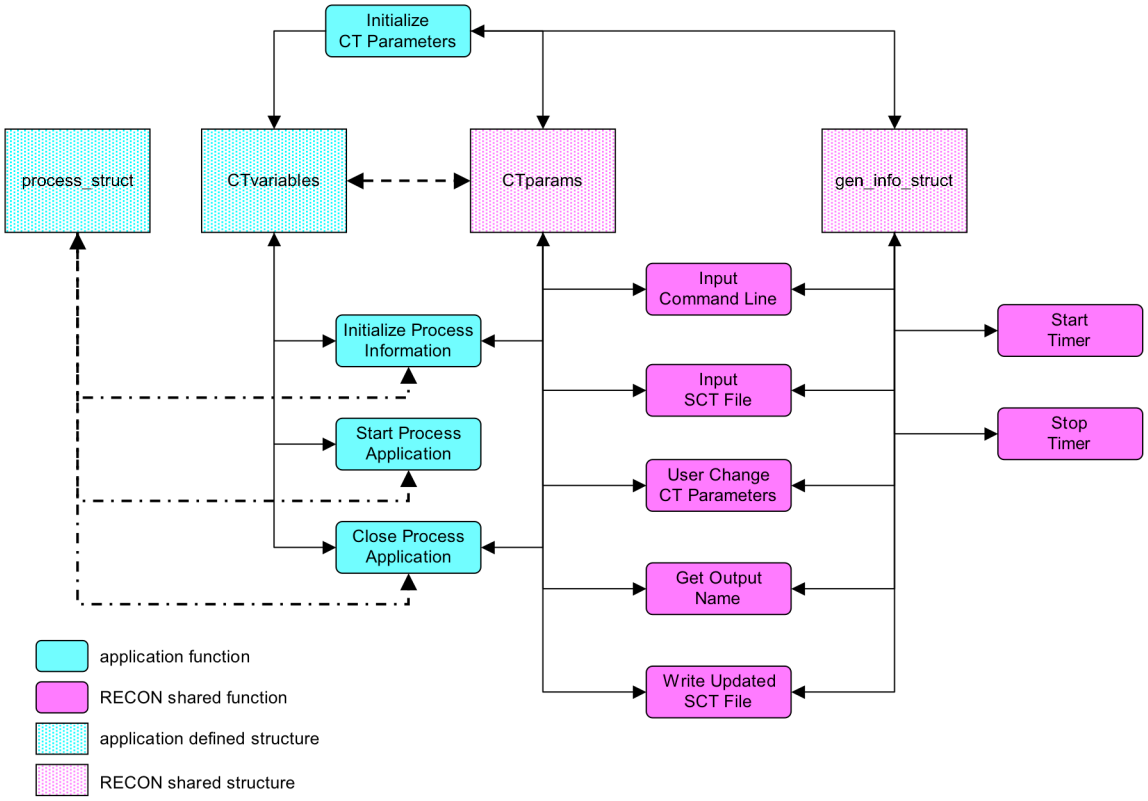


Figure II.4 – RECON Shared and Application Structures

C. *main( )* Functions

Each of the functions called by *main()* are described.

1. Initialize CT Parameters – *init_ctp() - (cbp.c, fkrecl.c, ljcone_ccg.c lncone_ccg.c, lpcone_ccg.c)*

a. CT Parameter Structure - *CTparams*

CTparams structure is defined in ***get_ctp.h*** (**util** directory) The elements in the structure are

char *name	Pointer to the name of the parameter
char *value	Pointer to the value of the paramter
int datatype	Parameter data type – types are defined in ***recon.h***
int src_loc	Location from where the parameter was set – default, command line, edit, sct file, internal – types are defined in ***get_ctp.h***
char *range	Pointer to range of possible values

char *prompt	Pointer to prompt for the paramter – message to ask user for input

b. CT Variables Structure – *CTvariables*

The *CTvariables* structure is defined in an application header file. It is used to transfer variables among the application codes. The shared RECON functions will pass the pointer to the structure without knowing what is in it. All applications have parameters such as pixel and voxel size, sod, and sdd that will be set in the *SCT File*. Some of the parameters defined in the structure will not be set in ***init_ctp()*** but they may be set or used in the other application functions.

The CT parameters that input to the application will have to have variables in the *CTvariables* structure that contain their name, value, range, and prompt. When the variables are defined ***default values*** are set, if the CT parameter is not set by the *SCT File*, the command line or user edit the default value is used.

c. Purpose

The purpose of the application ***init_ctp()*** is to define the CT parameters needed by the application in the *CTvariables* structure and to set the pointer values in *CTparams* to the appropriate values in *CTvariables.*

2. Command Line Inputs – *parse_cmd()* – *parse_cmd.c*

a. Format

The command line format is

> appname sctfilename –ctpname ctpvalue …. –help –debug debugoption

The ***appname*** is the application name, the options are ***cbp, fkrecl, ljcone_ccg, lncone_ccg*** and ***lpcone_ccg*.**

After the user starts an application from the command line the code prints out RECON start up information. This information contains status information on the debug level and run status. Then it prints out all the CT parameters needed by the application with the set values, ranges and description. If default values are used they are marked with a "*". At this point the user is allowed to make changes to the CT parameters. An example of RECON start up information is shown in Appendix B.

b. *sctfilename* - SCT File

sctfilename is the name of the *SCT File* to be input. It's use in the command line is optional.

c. –ctpname ctpvalue - CT Parameters

CT Parameters can be input on the command line. The name is entered as *–ctpname* followed by the value *ctpvalue*. A number of parameters can be set. A value set on the command line will override values input from the *SCT File.*

d. *–help* - Help

If the ***–help*** option is on the command line help information is printed and the program quits.

e. –debug debugoption – Debug Messages – *debug_msg() – error_check.c*

The debug option sets the level for messages printed out during a run. The options for ***debugoption*** are

OFF	No message are printed
UPDATE	Small selection of messages are printed indicating current processing
STATUS	More messages about the status of the processing
ALL	All the available messages from all the functions, including file I/O etc

The function ***debug_msg****(level)* checks the global variable *g_debug_flag* to see if the input value *level* less than *g_debug_flag,* if it is a TRUE value is returned. Also if *g_debug_flag* is set to ALL the string containing the function path is printed so the function path will precede each debug message.

f. –run

If the *–run* option is on the command line the RECON start up information will not be printed, the user will not be allowed to change CT parameters, and the application will start processing immediately. This option is used to save time when the user does not need to see the start up information or during batch processing with a script file.

3. CT Parameter I/O

a. Default

init_ctp() defines default values for each CT parameter, these default values will be the CT parameter values unless reset by the *SCT File*, the command line or the user edit.

b. Read SCT File – *get_ctp()* - *get_ctp.c*

The format for an *SCT File* line is

-ctparametername value

A comment line in the *SCT File* starts with !.

```
! This is a comment line
```

The function ***get_ctp()*** reads each line of the *SCT File.* If the line starts with "!" the line is ignored. If the line starts with "-" the code searches through the information *CTparams* structure for a variable whose name matches *ctparametername*. If it does not find a match the line is ignored. If it finds a match it reads in the *value* and checks it against the variable range values associated with the variable in the *CTparams* structure.

If the value is out of range the variable's default value is used and an error message is printed.

If the value is not the correct type, integer, floating point etc, the default value will be used and an error message is printed.

c. Command Line – *parse_cmd()* – *parse_cmd.c*

When the ***parse_cmd()*** function detects a "-" if is not followed *debug, run* or *help* it assumes it is a CT parameter. The function then performs the same operations as ***get_ctp()***; check for variable name match in *CTparams* structure, check ranges, and check data type. The value of a CT parameter set from the command line will override a value set in the *SCT File.*

d. User Edit – *edit_ctp()* – *get_ctp.c*

After the application is started, if the *–run* option is not on the command line, the RECON start up information will be printed and the lines

```
Enter parameter to be changed and value [-parameter value]
Enter <return> to exit edit mode
```

will follow. At this point the user can change CT parametes, one at a time as shown

```
> -pxsize 0.02
```

When the user is done changing parameters <return> will start the application processing.

4. Get Output Name – *get_out_name()* – *get_ctp.c*

This function creates the name for the reconstructed data file. If the CT parameter *–rfile* has been set it will use that. If *–rfile* has not been set it will create it from the *SCT File* name. It generally strips the "p" from the front of the *SCT File* name, put an "r" there and call that the output file name. After an output name has been created the *outdir* is checked, if a file of that name already exists the message

WARNING: Output CT parameter file r2400 already exist.

Enter <return> to overwrite it, or enter a new file name:

will be printed. This allows the user to decide whether to change the name or overwrite the existing file.

5. Initialize Application Process – *init_proc()*

Every application must have a function ***init_proc()*.** It is generally in the same C file as ***init_ctp().*** ***init_proc()*** takes the input CT parameters and calculates variables needed by the application. It may also allocate application memory. The variables that it calculates and the memory locations can be saved in the *CTvariables* structure or in the structure *process_struct.* Both of these structures are passed to the application process.

6. Timer – *start_timer(), end_timer()* – *timing.c*

start_time() starts a timer before the application processing starts. ***end_time()*** stops the timer after the application processing is done. It calculates an elapsed time. The start, end and elapsed times are stored in the *gen_info_struct* and written to the user.

7. Application Process – *proc_app()*

Every application must have a function ***proc_app()*. *proc_app()*** simply calls the function that starts the application processing. It may copy information from one structure to another if necessary and print status messages.

8. Close Application Process – *close_app()*

Every application must have a function ***close_app()*. *close_app()*** outputs the reconstructed object data to designated output file. It may set CT parameters that need to be written to the output *SCT File*. It also closes all the memory opened by the application that has not been closed up to this point.

9. Write Updated SCT File – *write_sctfile()* – *get_ctp.c*

write_sctfile() opens the output *SCT File* and writes header information such as “LLNL-NDE Computed Tomography Information © Copywrite … “, it includes run information: date, user name, run time, and method. Then it writes out any CT parameters that were changed from the input *SCT File*. This includes any values the user changed from the command line or by editing or that the processing. Finally the original *SCT File* is copy to the end of the output *SCT File.*

III. CCG

A. Theory

The purpose of CCG is to find the minimum of the cost function $\mathcal{L}(\mathbf{x})$ with respect to **x** as efficiently and accurately as possible. $\mathcal{L}(\hat{\mathbf{x}})$ is a multi-dimensional concave surface whose minimum is being sought. The gradient, **g**, of the function, $\frac{\partial \mathcal{L}(\hat{\mathbf{x}})}{\partial \hat{\mathbf{x}}}$, is also a vector. The negative of the gradient at a particular $\hat{\mathbf{x}}$ will point to the steepest slope of the function at the location. A new value of $\hat{\mathbf{x}}$ can be determined along this direction which decreases the value of $\mathcal{L}(\hat{\mathbf{x}})$. The direction vector is **d**. An important consideration is how far along **d** should the next value of $\hat{\mathbf{x}}$ be. This distance is found by a *line search*. After the first update to $\hat{\mathbf{x}}$ is done, new gradient is determined and the next direction will be chosen to be orthogonal to the preceding directions, making for a more efficient search.

The basic steps in the CCG algorithm are shown in Figure III.1. There is an initialization phase that joins the main processing loop in the middle. The description of the algorithm will begin with the initialization and then follow the processing into the middle of the loop and back around until all the steps have been described once.

1. Initial Object

The first step is to initialize $\hat{\mathbf{x}}^0$. These initial values are checked against the constraints. Any value of $\hat{\mathbf{x}}^0$ that exceeds a constraint will be set to the constraint.

2. Initial Residual

Then the values for $\hat{\mathbf{y}}^0$ and $\Delta\mathbf{y}^0$ are calculated

$$\begin{aligned} \hat{\mathbf{y}}^0 &= \mathbf{A} \cdot \hat{\mathbf{x}}^0 \\ \Delta\mathbf{y}^0 &= \hat{\mathbf{y}}^0 - \mathbf{y} \end{aligned} \qquad \text{[III.1]}$$

The cost function $\mathcal{L}(\mathbf{x})$ is calculated as the least-squares maximum likelihood function

$$\mathcal{L}(\mathbf{x}) = \tfrac{1}{2}\left(\Delta\mathbf{y}^0\right)^2 \qquad \text{[III.2]}$$

3. Initial Gradient

The next step is to calculate the gradient of the cost function with respect to **x** at the initial value of $\hat{\mathbf{x}}^0$.

$$\mathbf{g} = \nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}) \qquad \text{[III.3a]}$$

The gradient for the least-squares cost function is

$$\mathbf{g}^i = \mathbf{A}^T \Delta \mathbf{y}^i \qquad \text{[III.3b]}$$

so $$\mathbf{g}^0 = \mathbf{A}^T \Delta \mathbf{y}^0 \qquad \text{[III.3c]}$$

In CT the calculation in Equation III.3b is referred to as a back projection. The values in the projection sized vector $\Delta \mathbf{y}^i$ are multiplied by the weights in **A** and placed the object sized vector $\mathbf{g}^i$.

4. Initial Direction

The initial direction $\mathbf{d}^0$, will simply be the negative of the gradient

$$\mathbf{d}^0 = -\mathbf{g}^0 \qquad \text{[III.4]}$$

If the initial value of a member of **x**, x_v, is on a constraint then $d_v^0 = 0$.

5. Updated Object

The next estimate of **x** is $\hat{\mathbf{x}}^{i+1}$. This is determined by changing the current $\hat{\mathbf{x}}^i$ in the direction determined by $\mathbf{d}^i$. The distance moved in the $\mathbf{d}^i$ direction is the step size defined by α^i. So the new value of $\hat{\mathbf{x}}^{i+1}$ will be

$$\hat{\mathbf{x}}^{i+1} = \hat{\mathbf{x}}^i + \alpha^i \mathbf{d}^i \qquad \text{[III.5]}$$

a. Initial Step Size

The step size is determined using a line search to find the α that produces the smallest $\mathcal{L}(\mathbf{x})$ in the given direction. The initial step size, α_0^i, is selected using a 1-D Newton estimate. Let

$$dd = \left. \frac{\partial \mathcal{L}\left[\mathbf{x}(\alpha)\right]}{\partial \alpha} \right|_{\alpha=0} \qquad \text{[III.6a]}$$

and $$dd2 = \left. \frac{\partial^2 \mathcal{L}\left[\mathbf{x}(\alpha)\right]}{\partial \alpha^2} \right|_{\alpha=0} \qquad \text{[III.6b]}$$

then $$\alpha_k^i = \frac{dd}{dd2} \qquad \text{[III.6c]}$$

For $\hat{\mathbf{x}}^{i+1} = \hat{\mathbf{x}}^i + \alpha^i \mathbf{d}^i$

$$dd = \frac{\partial \mathcal{L}\left[\mathbf{x}(\alpha)\right]}{\partial \alpha} = \Delta \mathbf{y} \cdot \mathbf{A}\mathbf{d}^i = \mathbf{d}^i \cdot \mathbf{g}^i \qquad \text{[III.7a]}$$

and

$$dd2 = \frac{\partial^2 \mathcal{L}\left[\mathbf{x}(\alpha)\right]}{\partial \alpha^2} = \left(\mathbf{A}\mathbf{d}^i\right)^{\mathsf{T}}\left(\mathbf{A}\mathbf{d}^i\right) \qquad \text{[III.7b]}$$

b. Line Search - Step Size Iteration

$\hat{\mathbf{x}}^{i+1}$ is calculated with the current value of α_k^i. The values of $\hat{\mathbf{x}}^{i+1}$ are compared to the constraints, if any elements of $\hat{\mathbf{x}}^{i+1}$, x_v^{i+1}, exceed the constraints they are set to the constraint and the associated element of $\mathbf{d}^{i+1}$, d_v^{i+1}, is set to zero. If a x_v^{i+1} which was previously on a constraint has moved away from the constraint the d_v^{i+1} is set back to its original value as calculated in Equation III.4 or III.10.

The new values for $\hat{\mathbf{y}}^{i+1}$, $\Delta\mathbf{y}^{i+1}$, $\mathcal{L}(\mathbf{x})$, $\mathbf{A}\mathbf{d}^{i+1}$ and *dd* are calculated as

$$\begin{aligned} \hat{\mathbf{y}}^{i+1} &= \mathbf{A}\hat{\mathbf{x}}^{i+1} \\ \Delta\mathbf{y}^{i+1} &= \hat{\mathbf{y}}^{i+1} - \mathbf{y} \\ \mathcal{L}(\mathbf{x}) &= \frac{1}{2}\left(\Delta\mathbf{y}^{i+1}\right)^2 \\ dd &= \mathbf{A}\mathbf{d}^{i+1} \cdot \Delta\mathbf{y}^{i+1} \end{aligned} \qquad \text{[III.8]}$$

The results are evaluated and if it is determined that α_k^i is the best solution the processing moves on, otherwise another value of α_k^i is determined and the α loop is repeated.

6. Updated Gradient

Once α^i is determined the processing moves on to calculate a new gradient

$$\mathbf{g}^{i+1} = \mathbf{A}^{\mathsf{T}}\Delta\mathbf{y}^{i+1} \qquad \text{[III.9]}$$

7. Stopping Tests

At this point there are some stopping tests. There are a number of stopping tests scattered throughout the code. They do not seem to be working well for the data that is now being processed. The stopping will not be described in this document.

8. Updated Direction

Now the conjugate direction $\mathbf{d}^{i+1}$ will be calculated as

$$\mathbf{d}^{i+1} = -\mathbf{g}^{i+1} + \beta^{i+1}\mathbf{d}^i \qquad \text{[III.10]}$$

a. Beta

β^{i+1} is determined by the Gram-Schmidt technique

$$\beta^{i+1} = \frac{\sum g_v^{i+1}\left(g_v^{i+1} - g_v^i\right)}{\sum g_v^i g_v^i} \qquad \text{[III.11]}$$

β^{i+1} will create a direction $\mathbf{d}^{i+1}$ that is orthogonal to all the previous directions.

The process will continue until the stopping criteria are met or the preset maximum number of iterations are reached.

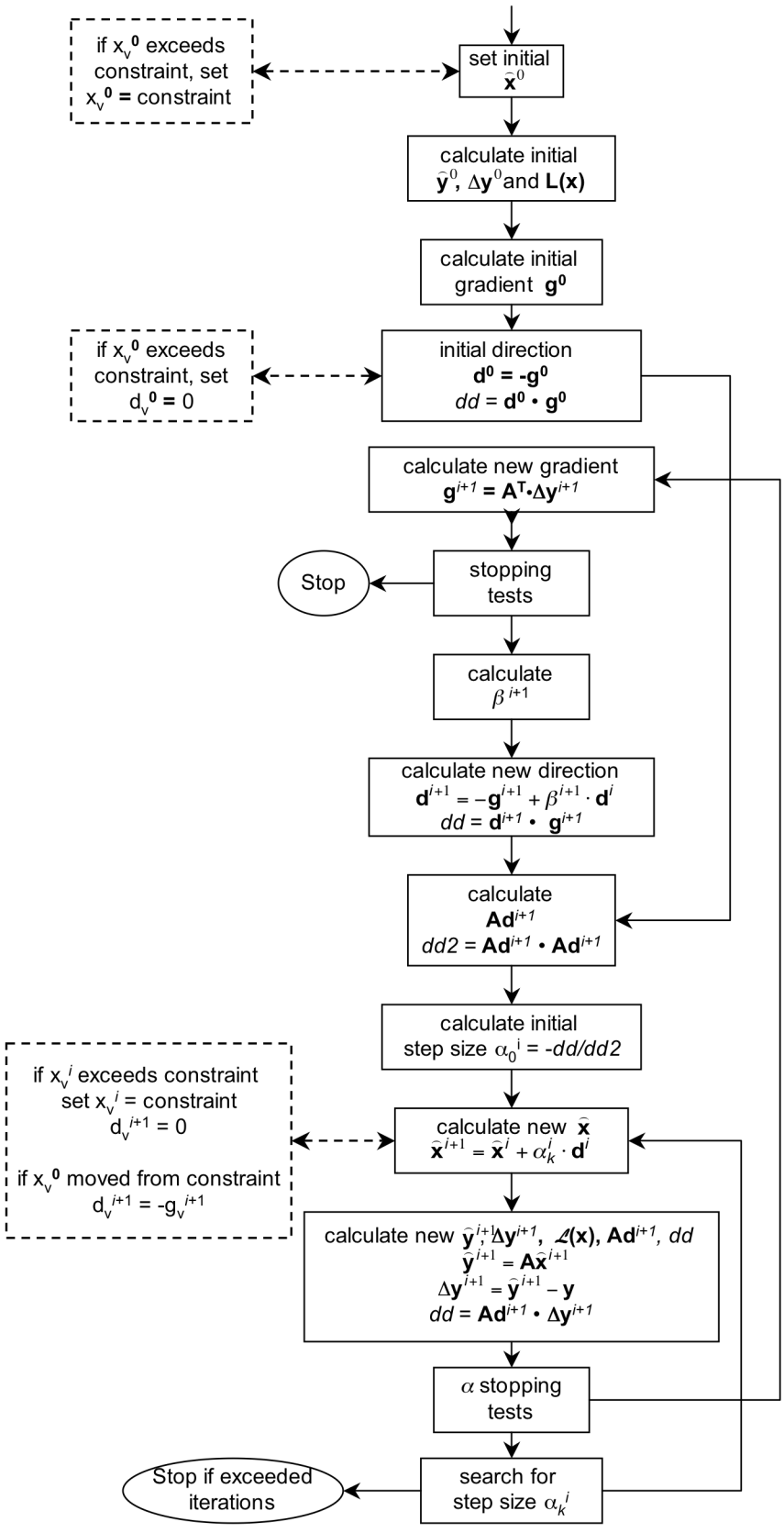


Figure III 1 – CCG Processing

B. Implementation

1. CCG CT Parameters

The following are the input CT parameters needed by the CCG for processing

Name	Type	Description
rmaxiters	integer	Maximum number of iterations
saveiters	NONE	Save interim information in VIEW volumes
	IMAGE	
	ALL	
cost_type	LST_SQ	Cost function type - least squares
	POISSON	poisson (not available)
	LST_SQ_LN	Least squares with ln (not available)
l1penalty	floating point	L1 Penalty parameter (0 = no penalty)
l2penalty	floating point	L2 Penalty parameter (0 = no penalty)
matcol	y/n	Use matcol (always no for pcone)
reconradius	floating point	Reconstruction radius (not used for pcone)
initialize	ZERO	Initialize to zero
	FROMFILE	Read initial data from file
constraints	NONE	No constraints
	NONNEG	Non-negative constraint
	FROMFILE	Read constraints from file
	FIXED	Fixed lower and upper constraints
fix_low_constr	floating point	Value for fixed lower constraints
fix_high_constr	floating point	Value for fixed upper constraints
fixedvars	y/n	Indicate whether to fix points
noisesig	floating point	Noise variance
initial_file	string	Initial image filename
min_const_file	string	Minimum constraint image filename
max_const_file	string	Maximum constraint image filename
fix_mask_file	string	Fix mask filename
fix_image_file	string	Fix image filename

2. Code Structure

The CCG portion of RECON is structured to be independent to allow for general-purpose use. The inputs to CCG are put into a specified form by the calling routines, in this case LCONE. CCG needs to use the System Matrix to update values during the processing. To do this the code calls certain functions that are part of LCONE. These functions can be modified to represent many

different systems without effecting CCG. Figure III.2 shows the algorithm steps from Figure III.1 along with the names of the *functions/files/directory* that perform the various steps. The source of the functions (CCG, LCONE, or Ray-Path) are indicated by color.

Figure III.3 is a data flow diagram of the CCG code itself. CCG has to maintain arrays for the current object data $\mathbf{x}^i$, the next object value $\mathbf{x}^{i+1}$, the current gradient $\mathbf{g}^i$, the next gradient $\mathbf{g}^{i+1}$, the current direction $\mathbf{d}^i$, the next direction $\mathbf{d}^{i+1}$ and a copy of the direction unmodified by the constraint checking $\bar{\mathbf{d}}$. Projection arrays also have to be maintained, the actual projection values **y**, the new estimate of projection $\hat{\mathbf{y}}^i$, and the residual $\Delta\mathbf{y}^i$. However $\hat{\mathbf{y}}^i$ and $\Delta\mathbf{y}^i$ can be combined into one in most cases.

The following files/codes are used in CCG.

a. – getsol.c

getsol() is the main function of CCG, it is the function that is called from the RECON code ***proc_app()***. ***alloc_mem()*** allocates the memory needed for the CCG processing, all the object and projection arrays that were not allocated in ***init_proc()*** to read in the data files. The processing loop that is shown in Figure III.1 is done in ***getsol()***.

b. – clsrch.c

The only function in this file is ***clsrch()*** it performs the line search for α^i. When each $\alpha_k{}^i$ is determined by ***clsrch()*** it calls the function ***update()*** to calculate the new values **x**.

c. – update.c

The ***update()*** function calculates the new values of **x**, and checks for the constraints. If values of **x** are changed because of the constraints ***update()*** can call ***matcol()*** if it is enabled. (This is the only time a CCG program directly calls a ray-path function, eliminating ***matcol()*** would improve the code structure.) If *matcol* is not enabled ***update()*** will call the standard LCONE functions to update the object vectors. (Note: ***matcol()*** is explained in Section IV).

d. – ivecops.c

This file contains codes that operate on the object vectors, **x, g, d**, either updating them or checking constraints and calculating products and sums of the object vector values.

e. – vector.c

This file contains low level vector operation codes for calculating things like dot products and sums.

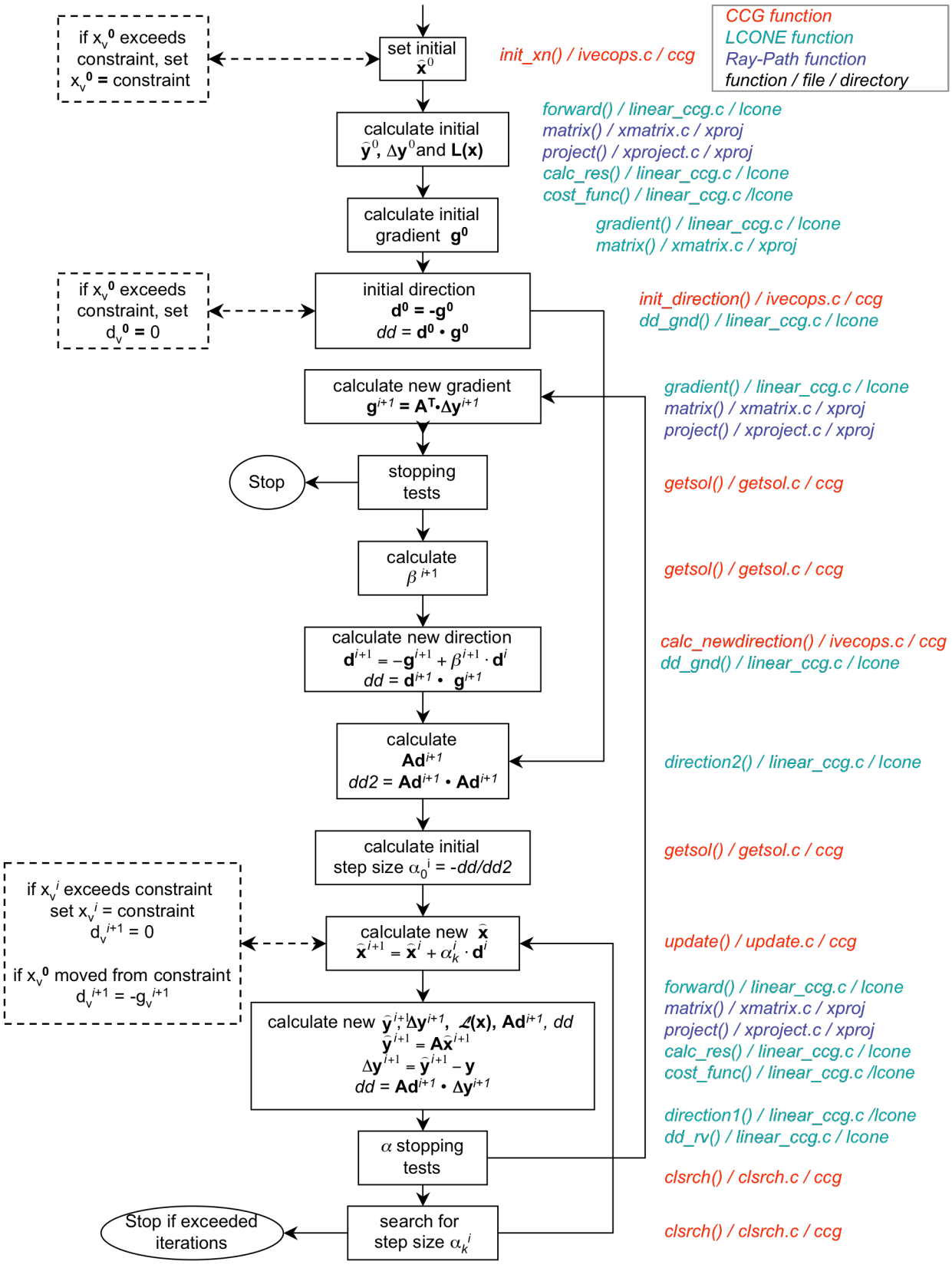


Figure III.2 – CCG Processing with Functions and Files

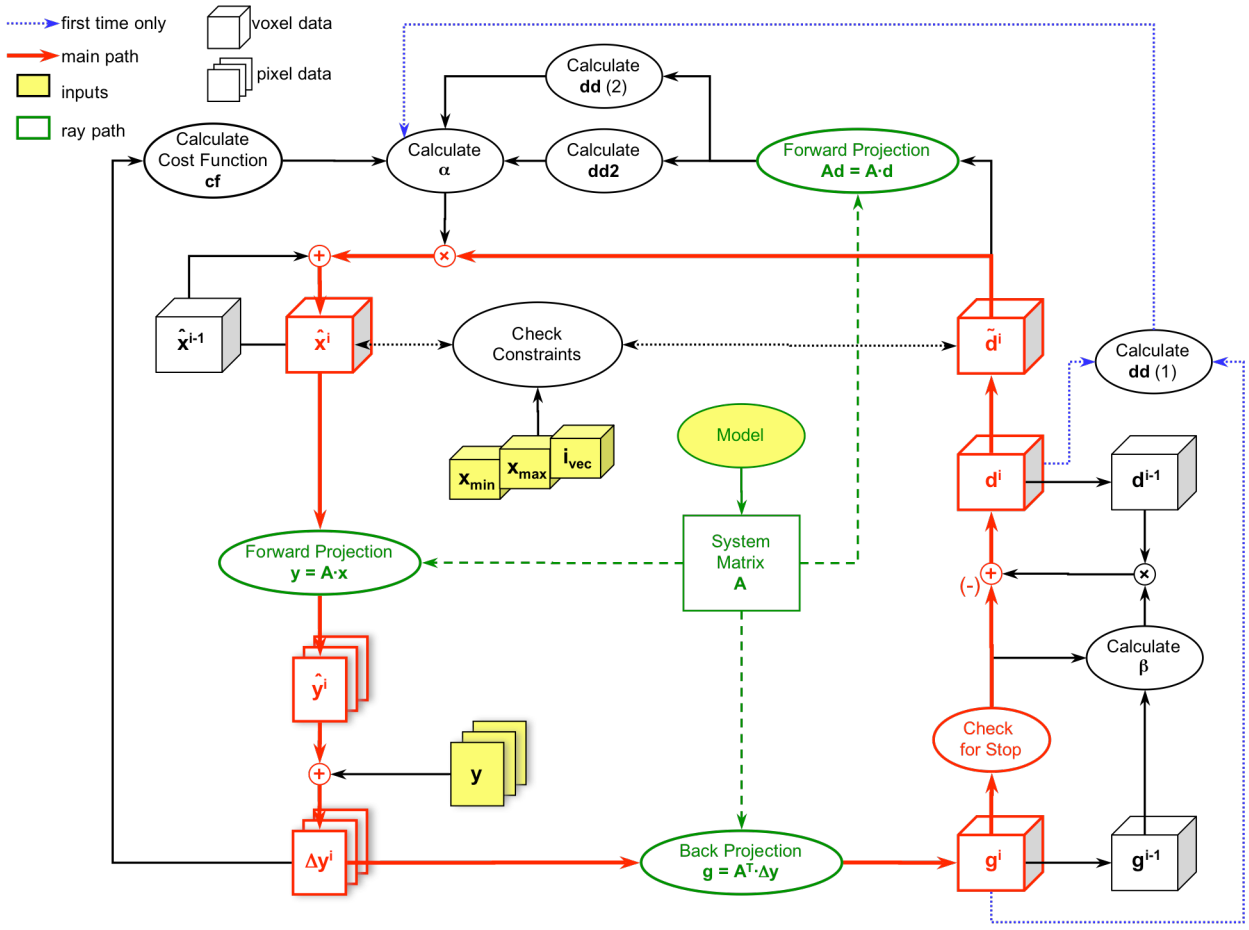


Figure III.3 – CCG Data Flow

IV. LCONE

A. Overview

LCONE is a linear ray-path model of a cone beam. It is assumed that an x-ray travels from the source to a pixel location on the detector in a straight line. As covered in Section I.C.2.d the value of the projection at a pixel is given be Equation I.11

$$y_p = \sum_{v=0}^{\mathbf{nc}-1} a_{vp} \cdot x_v \qquad for \qquad p = 0,1,2,...(\mathbf{nr}-1) \qquad [IV.1]$$

LCONE calculates the values of a_{vp} and performs the summation. There are a number of ways both of these operations can be done.

The code has been structured to allow different methods to be used easily. Each method will be compiled into a separate executable. As with RECON where certain function names are expected (such as ***init_ctp()***) LCONE expects certain function names: ***matrix(), matcol(),*** and ***project()***.

The ***recon/src/apps/lcone*** directory contains the file ***linear_ccg.c***. This file contains a number of functions that are called by the CCG codes. Some of these functions in turn call ***matrix(), matcol(),*** and ***project()***. The code structure in shown in Figure III.1.

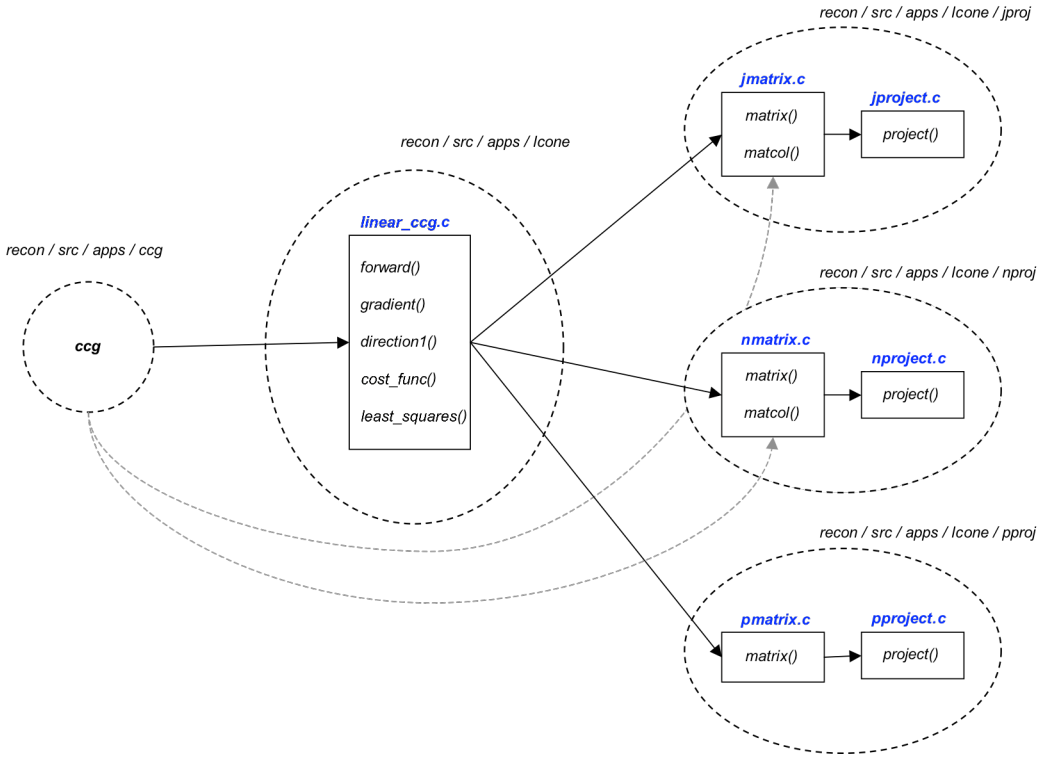


Figure IV.1– CCG-LCONE Program Structure

B. Common LCONE functions

1. - ***matrix() – jmatrix.c, nmatrix.c, pmatrix.c***

The ***matrix()*** function for each of the ray-path methods provides the outer loop. It calculates the location of the source and center of the detector for each angle of rotation. Then it calculates the location of each pixel for that detector position and calls ***project()*** giving it the source and pixel location as input. If a forward projection is being calculated, **y=Ax**, the **x** vector is also input, and ***project()*** then returns the y_p value. If a back projection, $\mathbf{g}=\mathbf{A}^T\Delta\mathbf{y}$, is being calculated the residual value Δy_p is input to ***project()*** along with the gradient vector, **g**. The gradient vector will also be the output.

2. - ***matcol() – jmatrix.c, nmatrix.c***

In **CCG** the object values, **x**, are checked to see if they exceed certain user defined constraints and if they do the value is set to the constraint. The direction vector **d** is also changed. This means that **y**, where **y=Ax**, and the calculated vector **Ad** have to be recalculated. This occurs each time a new α_k^i is determined during the line search. Recalculating these vectors is a matrix operation and is time consuming. The *matcol* method was developed to reduce this time.

matcol involves making the minimum changes necessary to **y** and **Ad**. Each time value of **x** and **d** is changed the values of **y** and **Ad** are changed so that

$$\begin{aligned} y_p^{new} &= y_p^{old} + a_{vp} \cdot x_v \\ & \qquad\qquad \text{for} \quad p = 0,1,2,\ldots(\mathbf{nr}-1) \\ Ad_p^{new} &= Ad_p^{old} + a_{vp} \cdot d_v \end{aligned} \qquad \text{[IV.2]}$$

This technique can speed up the processing but it can also cause problems. It is a difficult method to code which in turn creates errors. It also turns out that as the size of the problems increase, i.e. the size of the detectors, the time to access the memory to make the changes is much greater than simply performing the entire matrix calculation, at least for the first couple of iterations when a large number of values reach the constraints.. The choice to use *matcol* is available through the CT parameter **–matcol** for **jcone** and **ncone**.

Since **pcone** is so much faster *matcol* is not used at all. This can slow the processing down in the later iterations when few constraints are reached. *However if better stopping criteria are developed it is likely the program will stop before this becomes an issue.*

3. - ***project() – jmatrix.c, nmatrix.c, pmatrix.c***

For each application JCONE, NCONE, and PCONE ***project()*** calculates the values of a_{vp} with a different method. ***project()*** also performs the ray-sum for the forward projection, or updates the output vector for the back projection.

4. LCONE Input CT Parameters

The following are the CT parameters needed by the LCONE for reading and writing data, and calculating the ray-paths for all the projection implementations.

Name	**Type**	**Description**
sctfile	string	SCT filename
simflag	y/n	Simulate a radiograph, no reconstruction
Projection Parameters		
pfile	string	Projection filename
pfilegeom	SINOGRAM RADIOGRAPH	Projection file geometry
nrays	integer	Number of rays per projection
nslices	integer	Number of planer slices
nangles	integer	Number of projection angles (views)
arrange	floating point	Range of projection angles in degrees
pxdist	floating point	Distance between detectors in x (ray spacing) (mm)
pzdist	floating point	Distance between detectors in z (slices) (mm)
pxcenter	floating point	Distance in pixels from center of left-most pixel to location of axis-of-rotation on projection
pzcenter	floating point	Distance in pixels from center of top-most pixel to location of center of z-axis on projection
pxsrcctr	floating point	Distance in pixels from center of left-most pixel to center of source beam on x-axis
pzsrcctr	floating point	Distance in pixels from center of top-most pixel to center of source beam on z-axis
sod	floating point	Distance between source and origin of object axes
sdd	floating point	Distance between source and center of detector as defined by pxcenter and pzcenter
afile	string	Projection angle filename (optional –not used by pcone)

Name	Type	Description
Object Parameters		
rfile	string	Reconstruction object filename
volumeout	y/n	Save object data as a volume or sequence
rxelements	integer	Number of voxels in x
ryelements	integer	Number of voxels in y
rzelements	integer	Number of voxels in z
rxsize	floating point	Voxel size in x (mm)
rysize	floating point	Voxel size in y (mm)
rzsize	floating point	Voxel size in z (mm)
rxorigin	floating point	Physical location of center of first voxel in x (mm)
ryorigin	floating point	Physical location of center of first voxel in y (mm)
rzorigin	floating point	Physical location of center of first voxel in z (mm)

5. LCONE Calculated CT Parameters

There are a number of CT parameters calculated from the input CT parameters that can be used by all the projections method. Some of these values are determined in the function init_proc() others will need to be calculated in matrix().

The basic starting CT alignment is shown in Figure IV.2.

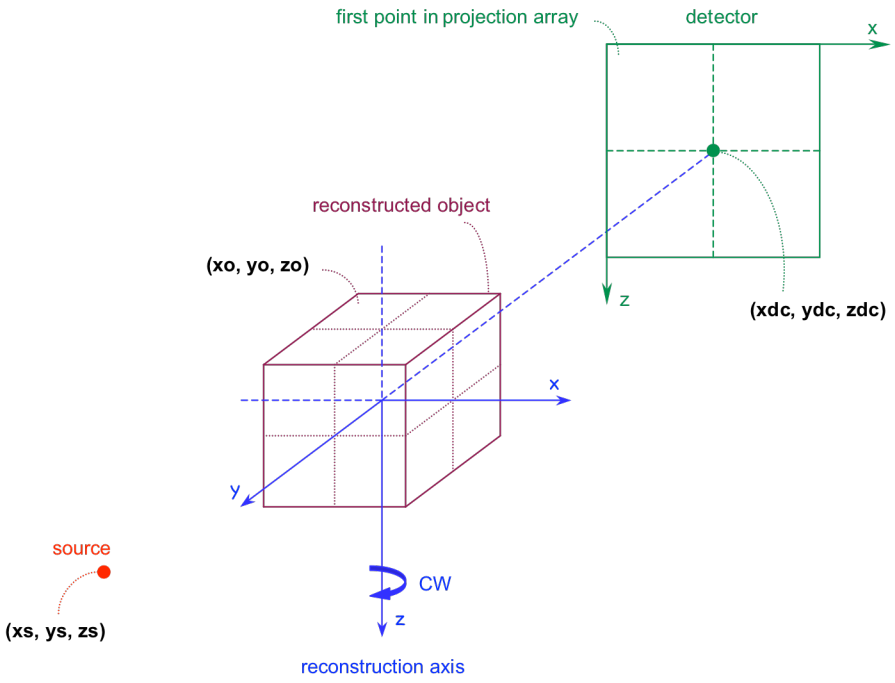


Figure IV.2 – CT Starting Alignment

The stage with the object rotates to perform the CT projection acquisition. To make the processing simpler it will be assumed that the stage is stationary and the source and detector rotate the opposite direction from the stage rotation. In the case shown in Figure IV.2 the source and detector will rotate CCW.

On Figure IV.2 the origin of the reconstruction object on the axis (the center of the lowest voxel) is defined as **(xo, yo, zo),** which corresponds to the input CT parameters **(rxorigin, ryorigin, rzorigin)**. In order to calculate the ray-paths the end points of ray line, the source location and the detector pixel location, must be determined. In Figure IV.2 the source location is defined as **(xs, ys, zs)**. and the center of the detector is defined as **(xdc, ydc, zdc)**. These values can be determined by the input CT parameter values, however a number of interim variables will need to be calculated to do this.

The angle of rotation θ is defined as shown in Figure IV.3. The angle δ represents an azimuth tilt that can be applied to system.

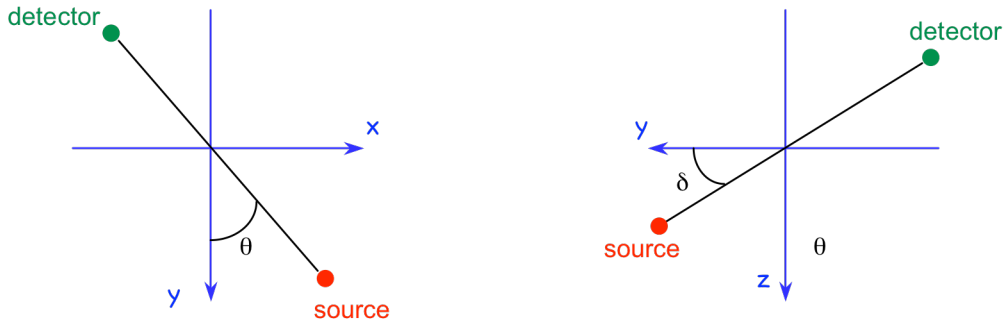


Figure IV.3 – CT Angular Definitions

Figure IV.4 is looking down on the x-y plane, **pxcenter, pxsrcctr, odd,** and **sdd** are input CT parameters. *pxm', pxm",* and *pxc'* are calculated as

$$\begin{aligned} pxm'' &= \mathbf{pxscrctr} - \mathbf{pxcenter} \\ pxm' &= pxm'' \cdot \frac{\mathbf{sod}}{\mathbf{sdd}} \\ pxc' &= \mathbf{pxsrcctr} - pxm' \end{aligned} \qquad \text{[IV.3]}$$

Figure IV.5 is viewing the x-z plane, **pzcenter, pzsrcctr, odd,** and **sdd** are input CT parameters. *pzm", pzm'*, and *pzc'* are calculated as

$$\begin{aligned} pzm'' &= \mathbf{pzsrcctr} - \mathbf{pzcenter} \\ pzm' &= pzm'' \cdot \frac{\mathbf{sod}}{\mathbf{sdd}} \\ pzc' &= \mathbf{pzsrcctr} - pzm' \end{aligned} \qquad \text{[IV.4]}$$

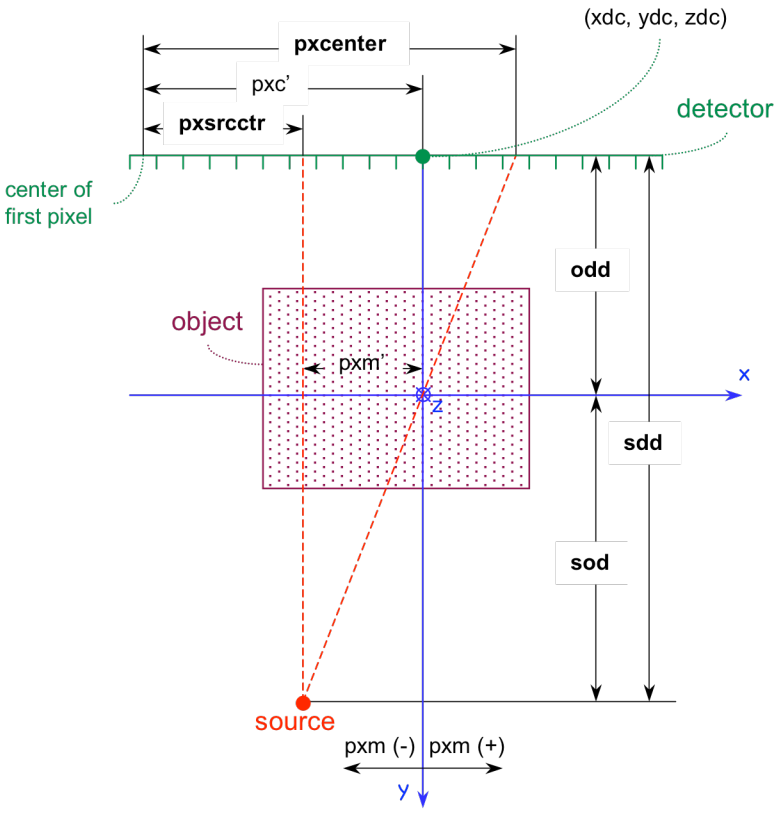


Figure IV.4 – CT X-Y Plane Definitions

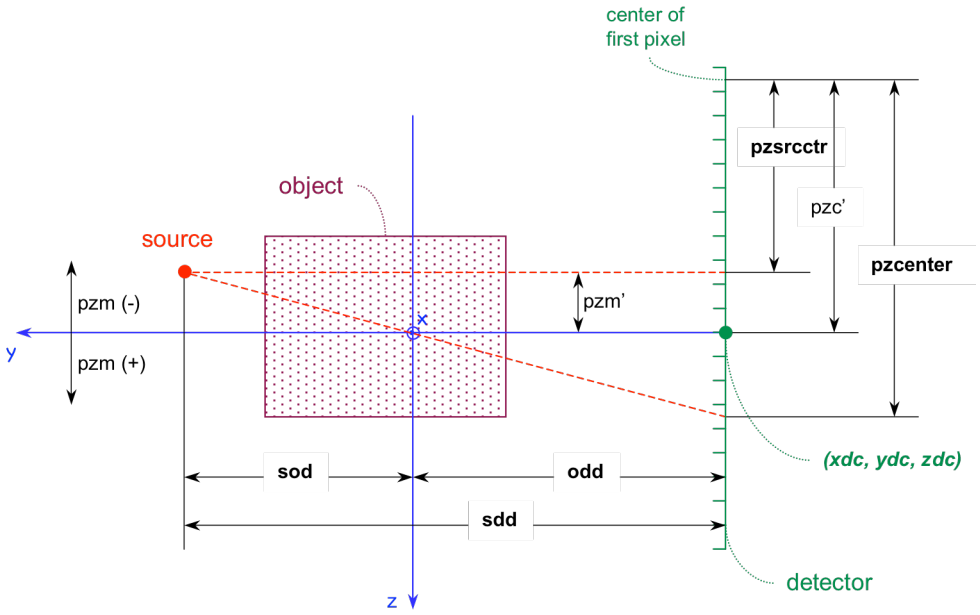


Figure IV.5 – CT X-Z Plane Definitions

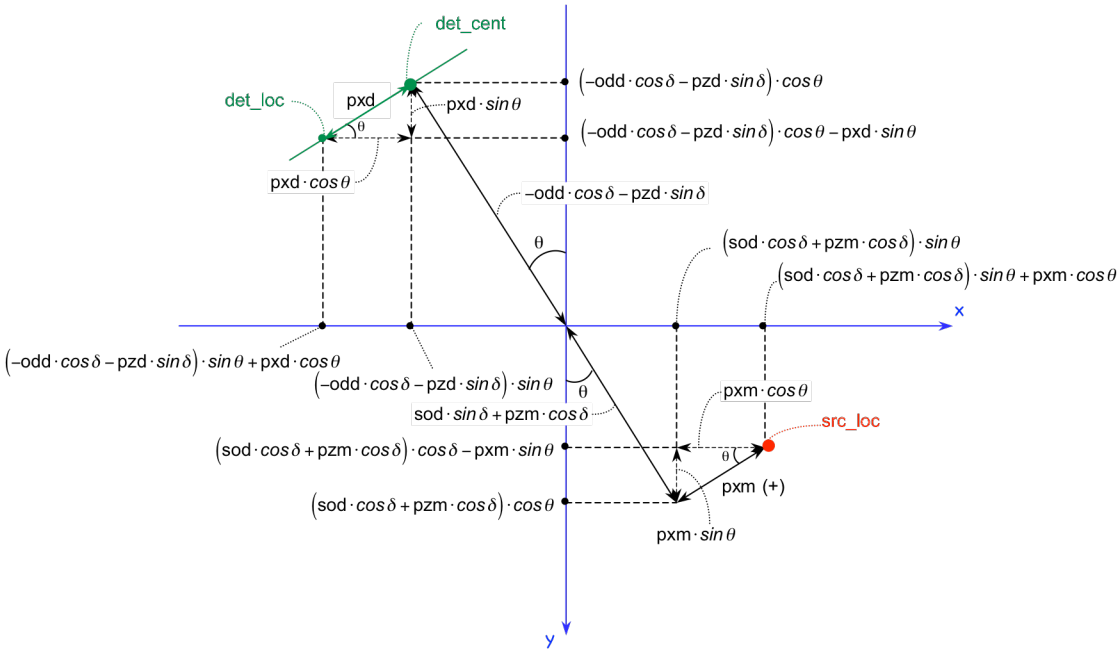


Figure IV.6 – CT X-Z Plane Pixel Definition

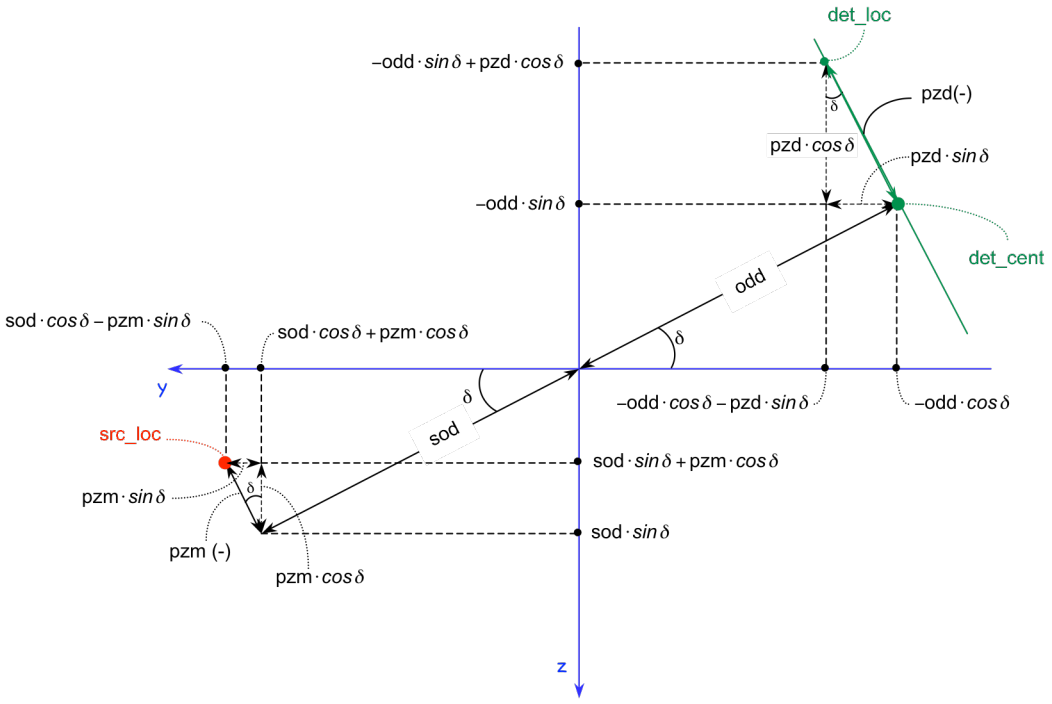


Figure IV.7 – CT X-Y Plane Pixel Definition

Now the pixel offset has to be taken into account. Figure IV.6 shows the x-z plane with the pixel offset and Figure IV.7 shows the x-y plane with the pixel offset.

The x and y axis position of the pixel, seen in Figures IV.6 and IV.7, is defined by number of pixels, *i,* from the smallest value of x and the number of pixels, *j*, from the smallest value of z, and the input CT parameters **pxdist** and **pzdist**.

$$\begin{aligned} pxm &= pxm' \cdot \mathbf{pxdist} \\ pxc &= pxc' \cdot \mathbf{pxdist} \\ pxd &= (i - pxc') \cdot \mathbf{pxdist} = i \cdot \mathbf{pxdist} - pxc \end{aligned} \qquad \text{[IV.5a]}$$

$$\begin{aligned} pzm &= pzm' \cdot \mathbf{pzdist} \\ pzc &= pzc' \cdot \mathbf{pzdist} \\ pzd &= (j - pzc') \cdot \mathbf{pzdist} = j \cdot \mathbf{pzdist} - pzc \end{aligned} \qquad \text{[IV.5b]}$$

Finally

$$\begin{aligned} \mathbf{xs} &= \mathbf{sod} \cdot \cos\delta \cdot \sin\theta + pzm \cdot \cos\delta \cdot \sin\theta + pxm \cdot \cos\theta \\ \mathbf{ys} &= \mathbf{sod} \cdot \cos\delta \cdot \cos\theta + pzm \cdot \cos\delta \cdot \cos\theta + pxm \cdot \sin\theta \\ \mathbf{zs} &= sod \cdot \sin\delta + pzm \cdot \cos\delta \end{aligned} \qquad \text{[IV.6a]}$$

$$\begin{aligned} \mathbf{xdc} &= -\mathbf{odd} \cdot \cos\delta \cdot \sin\theta \\ \mathbf{ydc} &= -\mathbf{odd} \cdot \cos\delta \cdot \cos\theta \\ \mathbf{zdc} &= -\mathbf{odd} \cdot \sin\delta \end{aligned} \qquad \text{[IV.6b]}$$

Then the pixel location is **(xd, yd, zd)**

$$\begin{aligned} \mathbf{xd} &= \mathbf{xdc} - pzd \cdot \sin\delta \cdot \sin\theta + pxd \cdot \cos\theta \\ \mathbf{yd} &= \mathbf{ydc} - pzd \cdot \sin\delta \cdot \cos\theta - pxd \cdot \sin\theta \\ \mathbf{zd} &= \mathbf{zdc} + pzd \cdot \cos\delta \end{aligned} \qquad \text{[IV.6c]}$$

C. JCONE Projection

The **jcone** method was the original ray-path projection. It is based on the traditional rectangular coordinate system. It worked but it was fairly slow so a new faster projection method, **ncone**, was developed. **jcone** still works but is not documented here. The executable is ***ljcone_ccg***.

D. NCONE Projection – *nproject()* – *nproject.c*

ncone is a ray-path projection method. It is based on a rectangular coordinate system. It is faster than **jcone**. However both JCONE and **ncone** have some problems. pxcenter can not be a whole number, a divide by zero error is generated that though it can be detected can not really be fixed.

Also the back projection used in CCG generates artifacts that would be very difficult to fix. The executable is ***lncone_ccg.***

The following is a description of the **ncone** calculations.

1. ncone Variables

The following structures are used in NCONE. Each structure contains the elements *x, y, z*. The structures are in ***bold italics*** and the elements are in *italics*.

ps - starting point of line
pe - ending point of line

vio - center of minimum voxel of object
vim - center of maximum voxel of object
vo - minimum point of object
vm - maximum point of object

vs - size of each voxel
n - number of voxels

d - direction unit vector of line
pf - point where line intersects object
ipf - index of voxel where line intersects object
inc - increment of voxel index for each coordinate

dl - difference between start and end point
lv - length in one voxel
vn - next grid line
ln - length from **pf** to next grid line
length - array of lengths in each voxel along the line
index - array of index of each voxel along the line

The following variables are also used

lt total length of line from ***ps*** to ***pe***
lf length of line from ***ps*** to the front face
lxo, lyo, lzo length from ***ps*** to the lower object boundary planes
lxm, lym, lzm length from ***ps*** to the upper object boundary planes

ps, ***pe***, ***vio***, ***vs***, and ***n*** are given

ps = (xs, ys, zs)

pe = (xd, yd, zd)

vio = (rxsize, rysize, rzsize)

vs = (rxsize, rysize, rzsize)

The ***length*** and ***index*** structures are used in the ray-sum calculation

Figure IV.8 show some of the **ncone** object variables.

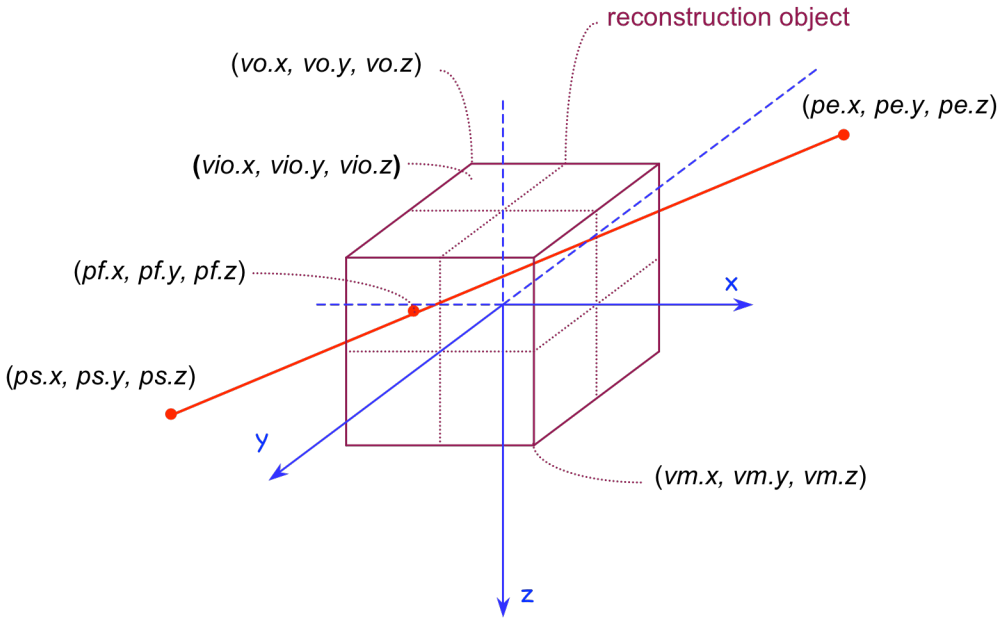


Figure IV.8 – NCONE Object Variables Definition

Determine the lowest corner of the object, ***vo***

$$vo.x = vio.x - \frac{vs.x}{2}$$
$$vo.y = vio.y - \frac{vs.y}{2} \qquad \text{[IV.7]}$$
$$vo.z = vio.z - \frac{vx.z}{2}$$

and the largest corner of the object, ***vm***

$$vm.x = vo.x + vs.x \cdot n.x$$
$$vm.y = vo.y + vs.y \cdot n.y \qquad \text{[IV.8]}$$
$$vm.z = vo.z + vs.z \cdot n.z$$

The unnormalized direction vector from ***ps*** to ***pe***, ***dl***

$$\begin{aligned} dl.x &= pe.x - ps.x \\ dl.y &= pe.y - ps.y \\ dl.z &= pe.z - ps.z \end{aligned} \qquad \text{[IV.9]}$$

The distance from ***ps*** to ***pe*** is ***lt***

$$lt = \sqrt{\left(dl.x\right)^2 + \left(dl.y\right)^2 + \left(dl.z\right)^2} \qquad \text{[IV.10]}$$

2. Location of Line-Object Front Intersection

Now it is needed to find where the line first intersects object. If the line intersects the object at all, it will do it twice, front (entering) and back (exiting). Define the six faces of the object as the planes at *vo.x (xmin), vm.x (xmax), vo.y (ymin), vm.y (ymax), vo.z (zmin), and vm.z (zmax).* Calculate the length from ***ps*** to each of these planes. The distances to the minimum planes are

$$\begin{aligned} lxo &= \left(vo.x - ps.x\right) \cdot \frac{lt}{dl.x} \\ lyo &= \left(vo.y - ps.y\right) \cdot \frac{lt}{dl.y} \\ lzo &= \left(vo.z - ps.z\right) \cdot \frac{lt}{dl.z} \end{aligned} \qquad \text{[IV.11]}$$

and the distances to the maximum planes are

$$\begin{aligned} lxm &= \left(vm.x - ps.x\right) \cdot \frac{lt}{dl.x} \\ lym &= \left(vm.y - ps.y\right) \cdot \frac{lt}{dl.y} \\ lzm &= \left(vm.z - ps.z\right) \cdot \frac{lt}{dl.z} \end{aligned} \qquad \text{[IV.12]}$$

Of these six lengths only two, if any, will be where the line intersects the object. These two points will have one coordinate equal to a plane of the object and the other two coordinates will be within the bounds of the object.

In the example shown in Figure IV.9 for *lxm* the coordinates for the point at *lxm* is (*vm.x, yxm, zxm*). By definition, if this point intersects the object

$$\begin{aligned} vo.y \le yxm \le vm.y \\ vo.z \le zxm \le vm.z \end{aligned} \qquad \text{[IV.13]}$$

Now

$$yxm - ps.y = lxm \cdot \frac{dl.y}{lt}$$
$$zxm - ps.z = lxm \cdot \frac{dl.z}{lt} \qquad \text{[IV.14]}$$

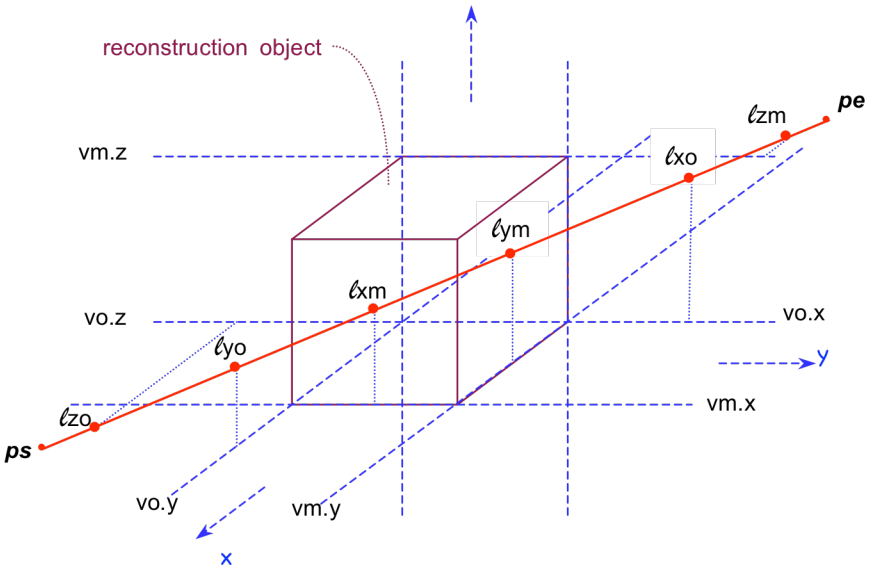


Figure IV.9 – NCONE Object Intersection Boundary Lengths

So

$$vo.y \le lxm \cdot \frac{dl.y}{lt} + ps.y \le vm.y$$
$$vo.z \le lxm \cdot \frac{dl.z}{lt} + ps.z \le vm.z \qquad \text{[IV.15]}$$

or

$$\left(vo.y - ps.y\right) \cdot \frac{lt}{dl.y} \le lxm \le \left(vm.y - ps.y\right) \cdot \frac{lt}{dl.y}$$
$$\left(vo.z - ps.z\right) \cdot \frac{lt}{dl.z} \le lxm \le \left(vm.z - ps.z\right) \cdot \frac{lt}{dl.z} \qquad \text{[IV.16]}$$

So finally

$$lyo \le lxm \le lym$$
$$lzo \le lxm \le lzm \qquad \text{[IV.17]}$$

This show that for *lxm* to be a valid point it must have two points (*lyo* and *lzo*) less than it and two points (*lym* and *lzm*) greater than it. This means that if the six lengths are sorted into order *lxm* would have to be one of the two middle lengths. If it's the first intersecting point it must obviously be the third length, *lf*: length to front face.

It is important to note that *dl.x, dl.y, dl.z* can all be potentially be equal to zero. This means that calculations in Equations IV.11 and IV.12 would be impossible. So before calculating Equations

IV.11 and IV.12 *dl.x, dl.y,* and *dl.z* must be checked. If they are zero the corresponding *lxo*, *lyo*, and *lzo* should be set to -1.0e+30 (-∞) and *lxm*, *lym*, and *lzm* to 1.0e+30 (∞). This represents reality and the sort will work correctly. After *lf* is found the coordinates of that point need to be calculated

$$\begin{aligned} pf.x &= lf \cdot \frac{dl.x}{lt} + ps.x \\ pf.y &= lf \cdot \frac{dl.y}{lt} + ps.y \\ pf.z &= lf \cdot \frac{dl.z}{lt} + ps.z \end{aligned} \qquad \text{[IV.18]}$$

These points need to be checked to see that they are in bounds of the object. Obviously the one that corresponds to the plane of intersection (ie. *lxo*) will be in bound. So need to check if

$$\begin{aligned} vo.x \le pf.x \le vm.x \\ vo.y \le pf.y \le vm.y \\ vo.z \le pf.z \le vm.z \end{aligned} \qquad \text{[IV.19]}$$

If these conditions are not met the line does not intersect the object.

3. Voxel Location of Line-Object Front Intersection

Now calculate the voxel where the line intersects, Figure IV.10 shows the voxel indexing.

$$\begin{aligned} ipf.x &= int\left[\frac{\left(pf.x - vo.x\right)}{vs.x}\right] \\ ipf.y &= int\left[\frac{\left(pf.y - vo.y\right)}{vs.y}\right] \\ ipf.z &= int\left[\frac{\left(pf.z - vo.z\right)}{vs.z}\right] \end{aligned} \qquad \text{[IV.20]}$$

Because of round off errors need to check

if ipf.x = n.x then ipf.x = n.x - 1
if ipf.x = -1 then ipf.x = 0
if ipf.y = n.y then ipf.y = n.y – 1 [IV.21]
if ipf.y = n.y then ipf.y = 0
if ipf.z = n.z then ipf.z = n.z - 1
if ipf.z = n.z then ipf.z = 0

4. Direction of Line

Now need to determine the direction, ***ds***, of the line

if $dl.x > 0$ then $ds.x = 1$ else $ds.x = -1$

if $dl.y > 0$ then $ds.y = 1$ else $ds.y = -1$

if $dl.z > 0$ then $ds.z = 1$ else $ds.z = -1$

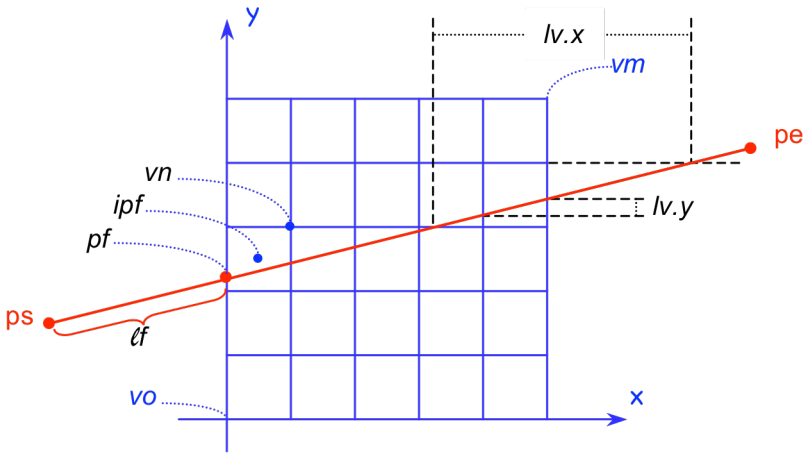


Figure IV.10 - NCONE Voxel Indexing

vn is the next grid line for each dimension in the direction of the line

$$\begin{array}{llll} \text{if} & dl.x > 0 & vn.x = vo.x + \left(ipf.x + 1\right) \cdot vs.x & \text{else} \quad vs.x = vo.x + ipf.x \cdot vs.x \\ \text{if} & dl.y > 0 & vn.y = vo.y + \left(ipf.y + 1\right) \cdot vs.y & \text{else} \quad vs.y = vo.y + ipf.y \cdot vs.y \\ \text{if} & dl.x > 0 & vn.z = vo.z + \left(ipf.z + 1\right) \cdot vs.z & \text{else} \quad vs.z = vo.z + ipf.z \cdot vs.z \end{array} \qquad \text{[IV.22]}$$

5. Line Index Values

When the line moves from one voxel to the next the voxel pointer index increases/decreases. Instead of calculating the index each time where *index* would be

$$index = ix + iy \bullet n.x + iz \bullet n.x \bullet n.y \qquad \text{[IV.23]}$$

just calculate the change. This change depends on which dimension(s) are changing and the direction of the change.

$$\begin{array}{llll} \text{if} & dl.x > 0 & inc.x = 1 & \text{else} \quad inc.x = -1 \\ \text{if} & dl.y > 0 & inc.y = n.x & \text{else} \quad inc.y = -n.x \\ \text{if} & dl.z > 0 & inc.z = n.x \cdot n.y & \text{else} \quad inc.z = -n.x \cdot n.y \end{array} \qquad \text{[IV.24]}$$

The first index is

$$index = ipf.x + ipf.y \bullet n.x + ipf.z \bullet n.x \bullet n.y \qquad \text{[IV.25]}$$

6. Line Length Values

As the line steps through the object it moves from grid line to grid line. The change between grid lines for each dimension is constant

$$\begin{aligned} lv.x &= vs.x \cdot \frac{lt}{dl.x} \cdot ds.x \quad if \quad dl.x = 0 \quad lv.x = 0 \\ lv.y &= vs.y \cdot \frac{lt}{dl.y} \cdot ds.y \quad if \quad dl.y = 0 \quad lv.y = 0 \\ lv.z &= vs.z \cdot \frac{lt}{dl.z} \cdot ds.z \quad if \quad dl.z = 0 \quad lv.z = 0 \end{aligned} \qquad \text{[IV.26]}$$

If *dl.x, dl.y,* or *dl.z* is zero the change in that direction will be zero.

Now ***ln*** will be the length from *lt* to the next grid line, as the line progresses through the object. So the first ***ln*** is from ***pf*** to ***vn***

$$\begin{aligned} ln.x &= \left(vn.x - pf.x\right) \cdot \frac{lt}{dl.x} \quad if \quad dl.x = 0 \quad ln.x = lt \\ ln.y &= \left(vn.y - pf.y\right) \cdot \frac{lt}{dl.y} \quad if \quad dl.y = 0 \quad ln.y = lt \\ ln.z &= \left(vn.z - pf.z\right) \cdot \frac{lt}{dl.z} \quad if \quad dl.z = 0 \quad ln.z = lt \end{aligned} \qquad \text{[IV.27]}$$

If *dl.x, dl.y,* or *dl.z* are zero then the associated ***ln*** element will be infinity, so for practical purposes set it equal to *lt.*

The *ln's* can be calculated ahead of time. First determine the number of grid lines

$$\begin{aligned} &if \quad ds.x > 0 \quad then \quad ix = n.x - ipf.x \quad else \quad ix = ipf.x + 1 \\ &if \quad ds.y > 0 \quad then \quad iy = n.y - ipf.y \quad else \quad iy = ipf.y + 1 \\ &if \quad ds.z > 0 \quad then \quad iz = n.z - ipf.z \quad else \quad iz = ipf.z + 1 \end{aligned} \qquad \text{[IV.28]}$$

Then

$$\begin{aligned} &for \quad i = 1 \quad to \quad ix - 1 \quad ln[i].x = ln[0].x + i \cdot lv.x \\ &for \quad i = 1 \quad to \quad iy - 1 \quad ln[i].y = ln[0].y + i \cdot lv.y \\ &for \quad i = 1 \quad to \quad iz - 1 \quad ln[i].z = ln[0].z + i \cdot lv.z \end{aligned} \qquad \text{[IV.29]}$$

Finally to step through the object choose which ***ln*** is the smallest, (nearest grid line, could be more than one). Once the appropriate dimension(s) is selected increment the *index*, the *ln* for that dimension and determine the length through that grid. A portion of the code to accomplish this is given here.

```
dindex = 0
if  ln[ix].x ≤ ln[iy].y  &&  ln[ix].x ≤ ln[iz].z
    dindex += inc.x
    lnew = ln[ix].x
    ix++
if  ln[iy].y ≤ ln[ix].x  &&  ln[iy].y ≤ ln[iz].z
    dindex += inc.y
    lnew = ln[iy].y
    iy++
if  ln[iz].z ≤ ln[ix].x  &&  ln[iz].z ≤ ln[iy].y
    dindex += inc.z
    lnew = ln[iz].z
    iz++
length = (lnew – last) • lt
last = lnew
```

E. PCONE Projection – Polar Projection

The idea of the polar projection method is to take advantage of the rotation of the object about the axis of rotation. If the angles of rotations are equal this method can be applied. The object is reconstructed into a polar reconstruction cylinder instead of the traditional rectangular reconstruction object. When the reconstruction is done the polar object is translated into a rectangular object.

The advantage of this method is that the lengths through the polar voxels only have to be calculated for one view, these lengths will be the same for all views. However because the width, or arc, of the polar voxels will vary greatly from the center of the object to the outer edge a graduated polar system will be used. A graduated polar slice is shown in Figure IV.11, in this figure ***nviews***=12.

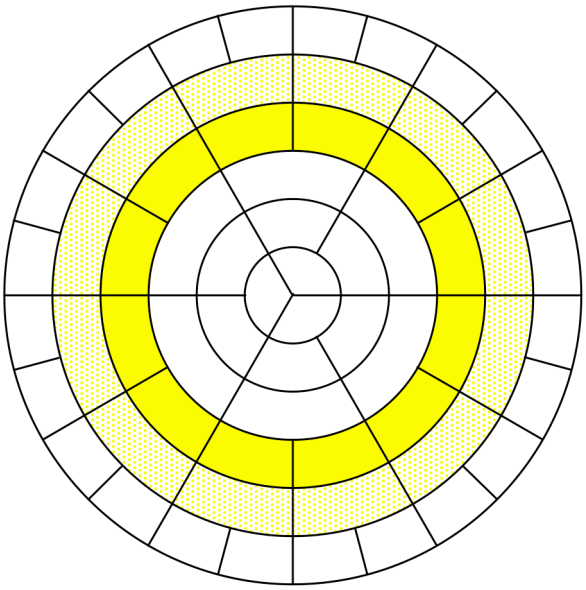


Figure IV.11 – Graduated Polar Slice

The radial voxel size will be equal to **rxsize.** A minimum radius, *irc*, will be found where there is one angular voxel of each of the **nviews**. This radius is shown in dark yellow in the figure. Below this radius the number of angular voxels will be factors of **nviews.** For these radii more than one length will have to be calculated. For radii above *irc* the number of angular voxels will be increased by multiples of two when appropriate to keep the voxel size reasonable.

The calculations involved in **pcone** are described in the following sections.

1. Graduated Polar Reconstruction Cylinder Design

A goal of the graduated polar system is to design the polar reconstruction cylinder so that the polar voxels at any point are smaller in volume than the rectangular voxels. The polar voxel volumes will vary with radius.

Let **dx = rxsize**
dy = rysize
dz = rzsize

Then the rectangular voxel, *rvol*, volume is defined as

$$rvol = dx \cdot dy \cdot dz \qquad \text{[IV.30]}$$

The polar voxel volume, $pvol_i$, is calculated subtracting the area of a circle for a given radius from the area of the next larger radius and then dividing by the number of angular divisions, $\boldsymbol{nangles_i}$, for the given radius, as seen in Figure IV.12

$$pvol_i = \left[\frac{\pi\left(\left(\boldsymbol{rx}\left(i+1\right)\right)^2 - \pi\left(\boldsymbol{rx}\cdot i\right)^2\right)}{\boldsymbol{nangles_i}} \right] \cdot zx = \frac{\pi \cdot \left(2 \cdot i + 1\right) \cdot \boldsymbol{rx}^2 \cdot \boldsymbol{zx}}{\boldsymbol{nangles_i}} \qquad \text{[IV.31]}$$

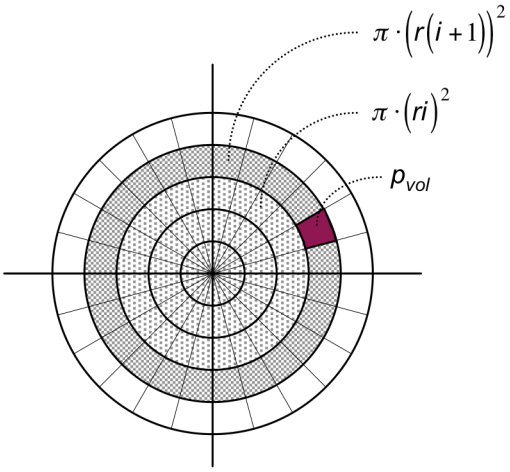


Figure IV.12 – pvol Calculation

The first step is to find the largest radius index, *irct*, where $pvol_i$ is less than *hrange*•*rvol*, when $\boldsymbol{nangles_{irct}}$ = **nviews**, or

$$pvol_{(irct)} < hrange \bullet rvol \qquad \text{[IV.32a]}$$

$$irct = (\text{long})\left[\frac{1}{2}\cdot\left[\frac{hrange\cdot\mathbf{dx}\cdot\mathbf{dy}\cdot\mathbf{dz}\cdot\mathbf{nviews}}{\pi\cdot\boldsymbol{rx}^2\cdot\boldsymbol{zx}} - 1\right]\right] \qquad \text{[IV.32b]}$$

The inequality is taken care of when *irct* is truncated to make it an integer value. *hrange* = 0.8 has worked well so far.

The next step is to determine the value for $\boldsymbol{nangles_i}$ for the radii with an index above *irct*. In order for the polar reconstruction to be efficient the number of angles for these radii will be restricted to powers of two of **nviews** or $\boldsymbol{nangles_i}$ = **nviews** • 2^n. The value of 2^n will be needed in the processing so it will be saved in the variable $\boldsymbol{pfactor_i}$. The starting value of $\boldsymbol{pfactor_{(irct+1)}}$ = 1. So for

$$i = (irct + 1), \ldots (\boldsymbol{nr} - 1)$$

$pvol_i$ will be calculated. When

$$pvol_i > hrange \bullet rvol \qquad \text{[IV.33a]}$$

$$\boldsymbol{nangles_i} = \boldsymbol{nangles_{i-1}} \bullet 2 \qquad \text{[IV.33b]}$$

and

$$\boldsymbol{pfactor_i} = \boldsymbol{pfactor_{i-1}} \bullet 2 \qquad \text{[IV.33c]}$$

The final step of this phase is to determine the values for $\boldsymbol{nangles_i}$ and $\boldsymbol{pfactor_i}$ for radii with indices from 0 to *irct*. The idea is to use the smallest value for $\boldsymbol{nangles_i}$ while keeping $pvol_i <$ *hrange.* For a given value of $\boldsymbol{nangles_i}$ as the radius increases the $pvol_i$ will increase so at some point $\boldsymbol{nangles_i}$ will have to be increased to reduce $pvol_i$. In this case the calculation will work backward from *irct* to 0. First the prime factors of **nviews** are determined in monotonically increasing order, $factors_k$ for $k = 0, 1, \ldots K$. Then the values for num_k are calculated as follows

$$num_0 = \textbf{nviews} \qquad \text{[IV.34a]}$$

$$num_k = \frac{num_{k-1}}{factors_k} \qquad for \qquad k = 1, 2, \ldots, K \qquad \text{[IV.34b]}$$

For example if **nviews** = 60: $factors_k$ = 2, 2, 3, 5 and num_k = 60, 30, 15, 5.

The values of $nangles_i$ will equal the values of num_k, the problem is to determine the radii at which to change from one value of num_k to the next.

The value $pvol'_{ik}$ is calculated for a given *i* and *k*, this is the polar voxel volume for a given radius with the next value of $\boldsymbol{nangles_i}$.

$$pvol'_{ik} = \frac{\pi \cdot \left(2 \cdot i + 1\right) \cdot \boldsymbol{rx}^2 \cdot \boldsymbol{zx}}{num_{k+1}} \qquad \text{[IV.35]}$$

This value will be greater than *hrange* until the radius decreases to the point $\boldsymbol{nangles_i}$ needs to be decreased.

To start set a new variable *pnum* = 1, and *k* = 0, and the variable ***irc*** is initialized ***irc*** = *irct+1.*

Then for *i* = *irct, irct*-1, … 0 calculate $pvol'_{ik}$.

If $pvol'_{ik} < hrange$ and $k < K$-1 then:

$$pnum = pnum \bullet factors_k \qquad \text{[IV.36a]}$$

and if *k* = 0 then

$$\boldsymbol{irc} = i + 1 \qquad \text{[IV.36b]}$$

this is the smallest radii with $\boldsymbol{nangles_i}$ = **nviews**,

and

$$k = k + 1 \qquad \text{[IV.36c]}$$

Then

$$\boldsymbol{nangles_i} = num_k \qquad \text{[IV.36d]}$$

$$\text{If } pnum = 1 \quad \text{then} \quad \boldsymbol{pfactor_i} = pnum \qquad \text{[IV.36e]}$$

$$\text{If } pnum \neq 1 \quad \text{then} \quad \boldsymbol{pfactor_i} = -pnum \qquad \text{[IV.36f]}$$

$\boldsymbol{pfactor_i}$ is set negative for $i <$ ***irc*** to be used as a flag in later processing.

If there are no prime factors for **nviews** the following values will be set:

$$irc = 0 \qquad \text{[IV.37a]}$$

$$\boldsymbol{nangles_i} = \textbf{nviews} \qquad \text{for i = 0, 1, } \ldots irct \qquad \text{[IV.37b]}$$

$$\boldsymbol{pfactor_i} = 1 \qquad \text{for i = 0, 1, } \ldots irct \qquad \text{[IV.37c]}$$

There a other variables that need to be determined.

$\boldsymbol{rr_i}$ is the actual radial distance to the outside edge of radius i and $\boldsymbol{rr2_i}$ is the radial distance squared.

$$\boldsymbol{rr_i} = (\boldsymbol{rx}(i+1)) \qquad \text{for } i = 0,1, \ldots (\boldsymbol{nr}-1) \qquad \text{[IV.38a]}$$

$$\boldsymbol{rr2_i} = (\boldsymbol{rx}(i+1))^2 \qquad \text{for } i = 0,1, \ldots (\boldsymbol{nr}-1) \qquad \text{[IV.38b]}$$

xdtheta is the anglar size of the outer most voxel

$$\boldsymbol{xdtheta} = \frac{2 \cdot \pi}{\boldsymbol{nangles_{nr-1}}} \qquad \text{[IV.39]}$$

$\boldsymbol{toff_i}$ is the value $\boldsymbol{nangles_i}$ needs to be multiplied by to equal $\boldsymbol{nangles_{nr-1}}$.

$$\boldsymbol{toff_i} = \frac{\boldsymbol{nangles_{nr-1}}}{\boldsymbol{nangles_i}} \qquad for \quad i = 0,1,\ldots(\boldsymbol{nr}-1) \qquad \text{[IV.40]}$$

Finally $\boldsymbol{roffset_i}$ is the offset in memory for the first voxel of each radii.

$$\boldsymbol{roffset_i} = \sum_{k=0}^{i-1} \boldsymbol{nangles_k} \qquad for \quad i = 1,2,\ldots\boldsymbol{nr} \qquad \text{[IV.41]}$$

Figure IV.13 shows all the variables defining a graduated polar slice and Figure IV.14 shows how the memory for the voxels is arranged.

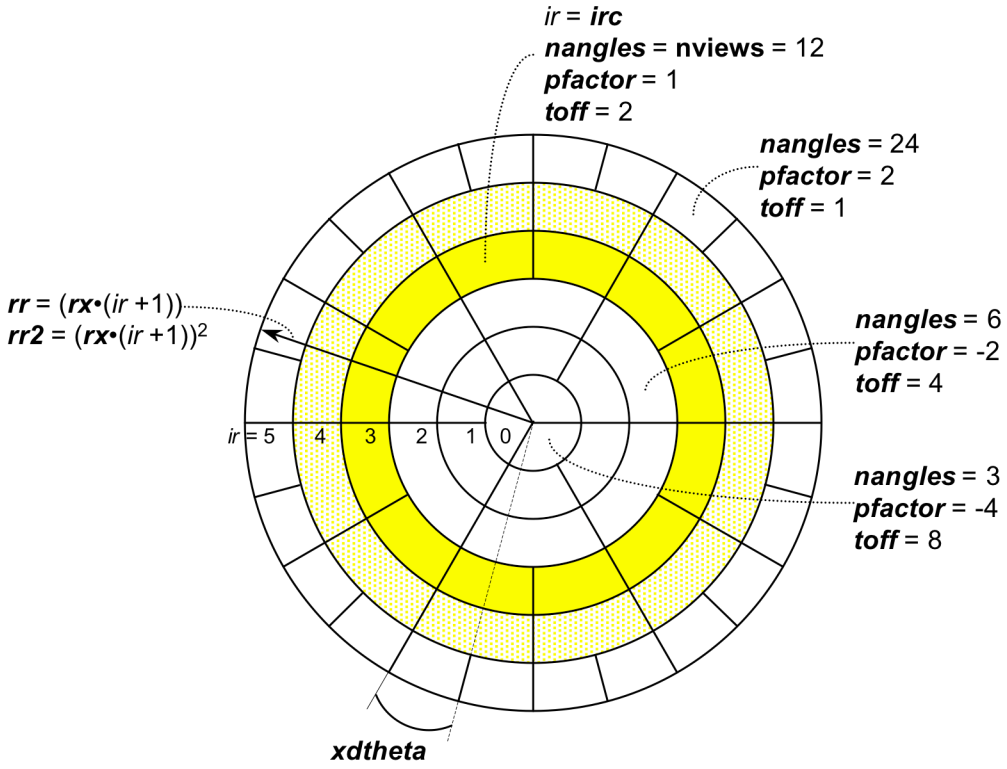


Figure IV.13 – Graduated Polar Definitions

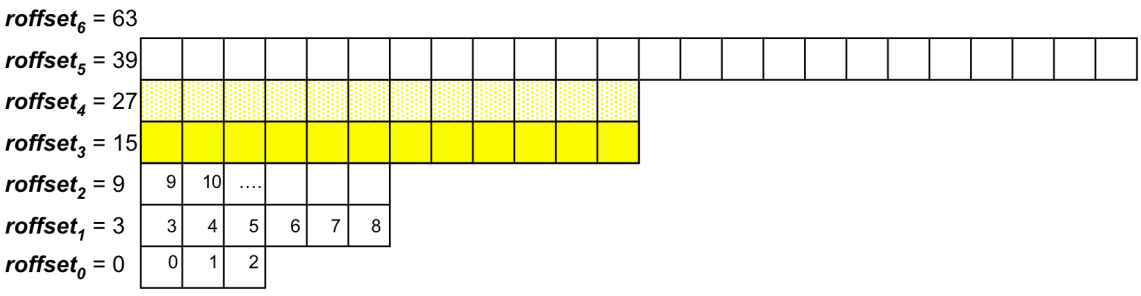


Figure IV.14 – Graduated Polar Memory

2. Ray-Path Definitions

The line ℓ through the polar reconstruction cylinder is defined by the source location **(xs, ys, zs)** and a given pixel location **(xd, yd, zd)**. The line ℓ and the polar reconstruction cylinder are shown in Figure IV.15. Line ℓ goes through a set of voxels in the polar reconstruction cylinder. The location of each voxel along the line and the length of the section of the line in each voxel is the information that is needed to calculate a ray-sum.

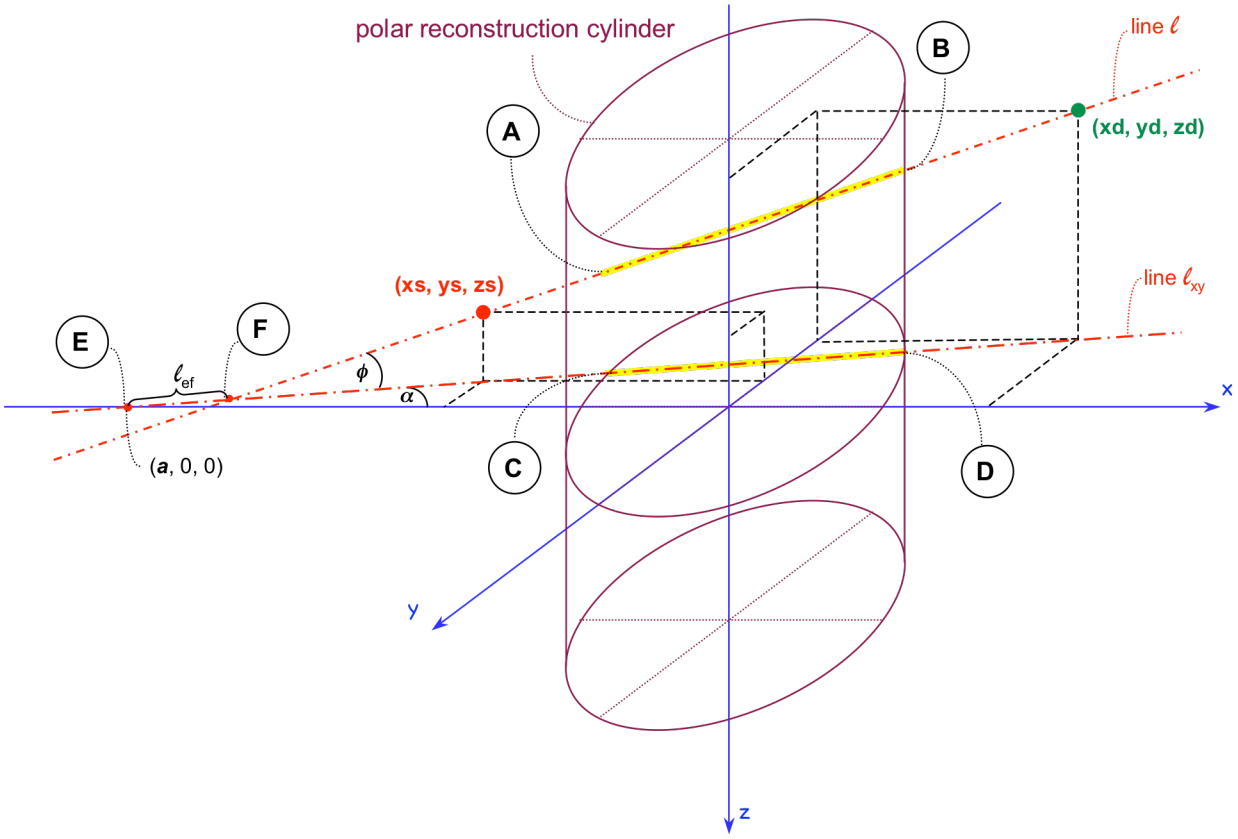


Figure IV.15 – Graduated Polar Ray-Path Definition

The basic idea is to determine lengths from the point where the line enters the polar reconstruction cylinder, point A on Figure IV.15, to each of the voxel boundaries along the line. The lengths are determined in groups; first the radial boundaries then the angular boundaries and finally the slice boundaries. The three separate groups of results are then sorted by length, during the sorting process the length in each voxel and the voxel location are determined.

At this point a number of variables need to be determined before the actual lengths to voxel boundaries can be calculated.

As seen on Figure IV.15, line ℓ enters the polar reconstruction cylinder at point A and exits at point B. Line ℓ can be projected on to the x-y plane forming ℓ_{xy}. Line ℓ_{xy} enters the polar reconstruction cylinder at point C and exits at point D. The angle between ℓ and ℓ_{xy} is $\boldsymbol{\phi}$, which can be calculated from the known values **xs**, **ys**, **zs**, **xd**, **yd**, and **zd**.

$$\boldsymbol{\phi} = tan^{-1}\left[\frac{\mathbf{zd}-\mathbf{zs}}{\sqrt{\left(\mathbf{xd}-\mathbf{xs}\right)^2+\left(\mathbf{yd}-\mathbf{ys}\right)^2}}\right] \qquad \text{[IV.42]}$$

The line ℓ_{xy} can be extended until it intersects with the x-axis at point E. (If the line ℓ_{xy} is parallel to the x-axis it of course cannot intersect the x-axis, this exception will be discussed later.) The variables $\boldsymbol{a}$ and α define ℓ_{xy}. These variables can be also determined from the known values **xs**, **ys**, **zs**, **xd**, **yd**, and **zd**.

$$\boldsymbol{a} = \mathbf{xs} - \frac{\left(\mathbf{xs}-\mathbf{xd}\right)}{\left(\mathbf{ys}-\mathbf{yd}\right)}\cdot\mathbf{ys} \qquad \text{[IV.43a]}$$

$$\alpha = tan^{-1}\left[\frac{\mathbf{yd}-\mathbf{ys}}{\mathbf{xd}-\mathbf{xs}}\right] \qquad \text{[IV.43b]}$$

Line ℓ can be extended until it intersects the x-y plane and point F, this point will also be on line ℓ_{xy}.

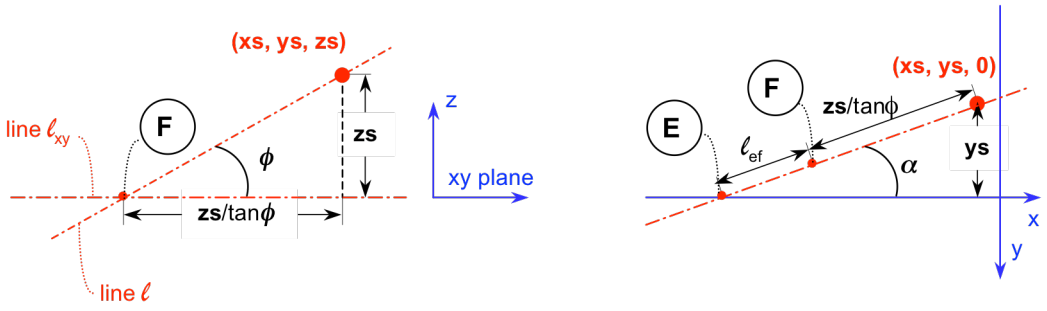


Figure IV.16 – Calculation of ℓ_{ef}

The distance between points E and F is ℓ_{ef} as shown on Figure IV.15. ℓ_{ef} can be determined by observing Figure IV.16.

$$\ell_{ef} = \frac{\mathbf{ys}}{\sin\alpha} - \frac{\mathbf{zs}}{\tan\phi} \qquad \text{[IV.44]}$$

All the needed variables shown on Figure IV.15 have to be determined. Figure IV.17 shows the x-y plane at z=0.

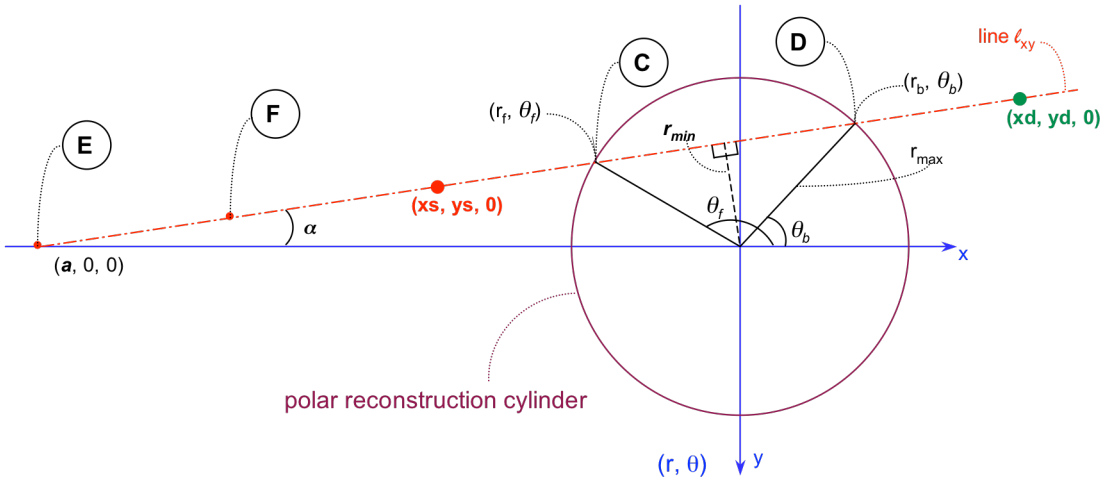


Figure IV.17 – X-Y Plane of Polar Reconstruction Cylinder

The x-y axis can also be viewed in polar coordinates as an r-θ axis. Each point on the x-y axis (x, y) can be represented on the r-q axis as (r, θ). The points (r_f, θ_f) and (r_b, θ_b) are polar coordinate locations of where the line enters and leaves the polar reconstruction cylinder, previously shown as points C and D. r_{max} is defined as the radius of the polar reconstruction cylinder. $\boldsymbol{r_{min}}$ is the perpendicular distance from line ℓ_{xy} to the polar coordinate origin, it is the closest line ℓ_{xy} comes to the polar origin. $\boldsymbol{r_{min}}$ can be negative because it contains a directional component. It is important to have the alignment of angles and directions correct in terms of the coordinate system, not just geometrically, because of the rotational nature of CT the equations will need to be correct for any locations of the source and detector. Therefore Figure IV.18 will be used to determine the variables shown on Figure IV.17.

The placement of line ℓ_{xy} to the coordinate axis is in the first quadrant. The following values can be determined:

$$r_f = r_b = r_{max} \qquad \text{[IV.45a]}$$

$$\theta_b = \alpha - \sin^{-1}\left[\frac{\mathbf{a}\cdot\sin\alpha}{r_{max}}\right] \qquad \text{[IV.45b]}$$

$$r_{min} = a \cdot \sin\alpha \qquad \text{[IV.45c]}$$

$$\theta_f = 180 + 2 \cdot \alpha - \theta_b \qquad \text{[IV.45d]}$$

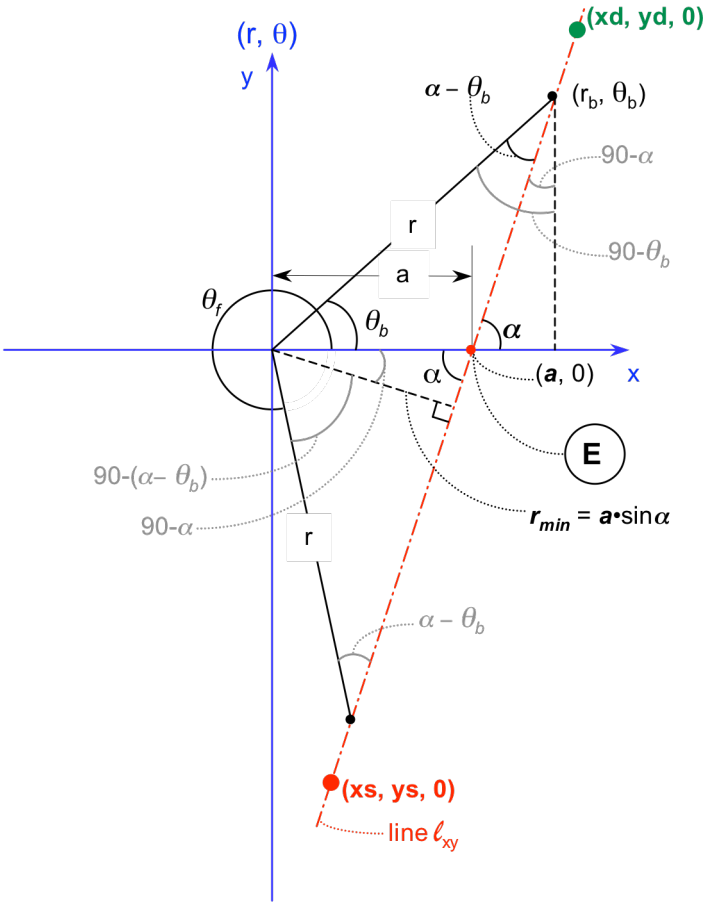


Figure IV.18 – Rotational Variables

To calculate and z_b first determine the length from point F to points C and D as shown in Figure IV.19.

$$y_f = r_{max} \cdot \sin\theta_f$$
$$y_b = r_{max} \cdot \sin\theta_b \qquad \text{[IV.46a]}$$

$$\ell_{fc} = \frac{y_f}{\sin\alpha} - \ell_{ef}$$
$$\ell_{fd} = \frac{y_b}{\sin\alpha} - \ell_{ef} \qquad \text{[IV.46b]}$$

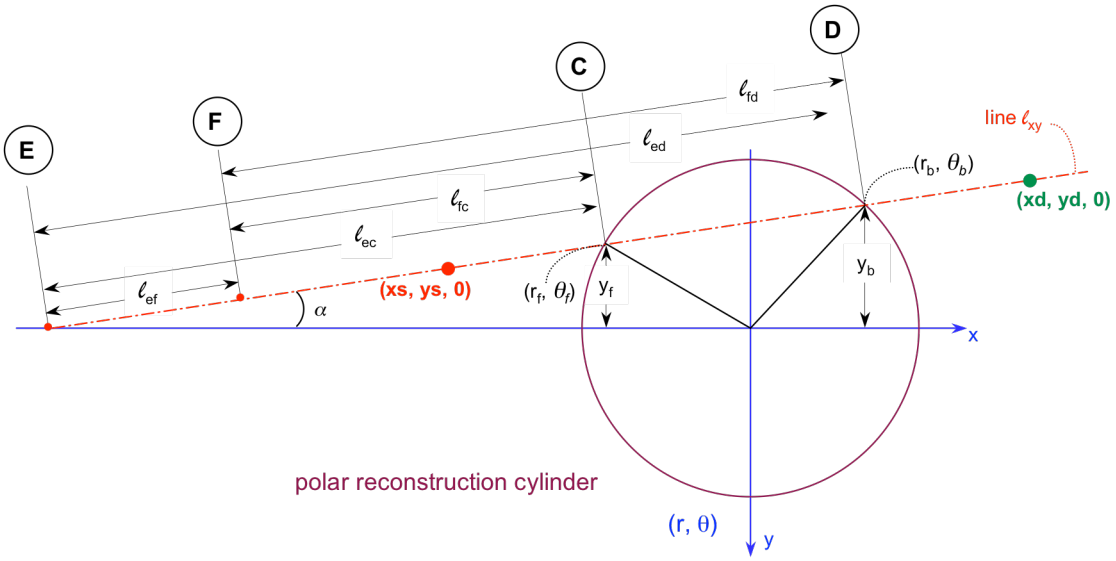


Figure IV.19 – Z Intersect on X-Y Plane

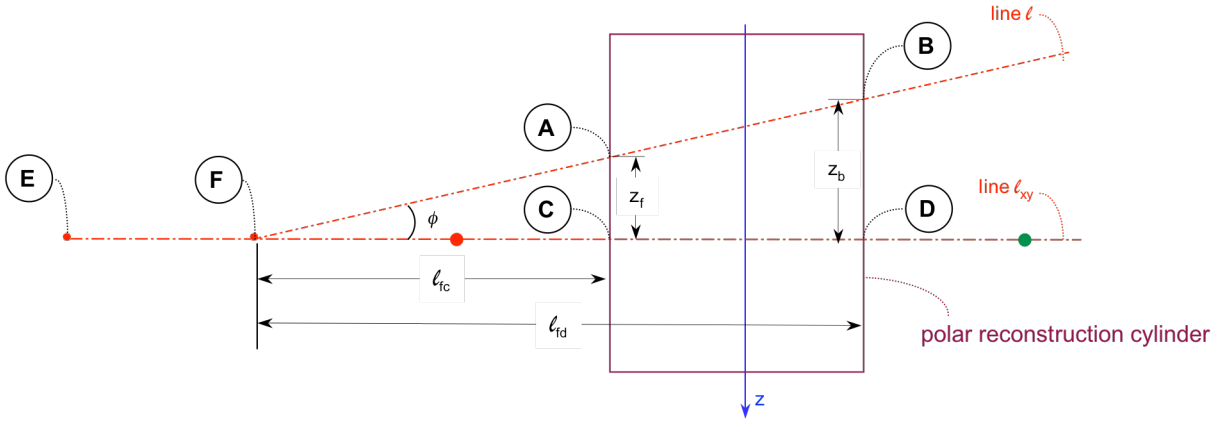


Figure IV.20 – Z Intersect on Vertical Plane

Then as seen in Figure IV.20 the value of z at points A and B can be determined.

$$z_f = \left(\frac{r_{max} \cdot \sin\theta_f}{\sin\alpha} - \ell_{ef} \right) \cdot \sin\phi$$
$$z_b = \left(\frac{r_{max} \cdot \sin\theta_b}{\sin\alpha} - \ell_{ef} \right) \cdot \sin\phi \qquad \text{[IV.47]}$$

Note that z_f and z_b can actually be above or below the polar reconstruction cylinder.

3. Length Calculations

Figures IV.21 and IV.22 show the basic idea of the length calculation process. On Figure IV.21 ℓ_{r0}, ℓ_{r1}, and ℓ_{r2} are the lengths on the x-y plane from point C to each of the radial boundaries the line crosses. Likewise ℓ_{t0}, ℓ_{t1}, and ℓ_{t2} are the lengths from point C to each of the angular boundaries. Figure IV.22 shows the radial and angular lengths on line ℓ_{xy} and the lengths ℓ_{z0} and ℓ_{z0} from point A to the z-slice boundaries. The radial and angular lengths must be divided by $\sin\phi$ to project them onto line ℓ. The z-slice lengths and the projected radial and angular lengths are sorted to form the final group ℓ_i.

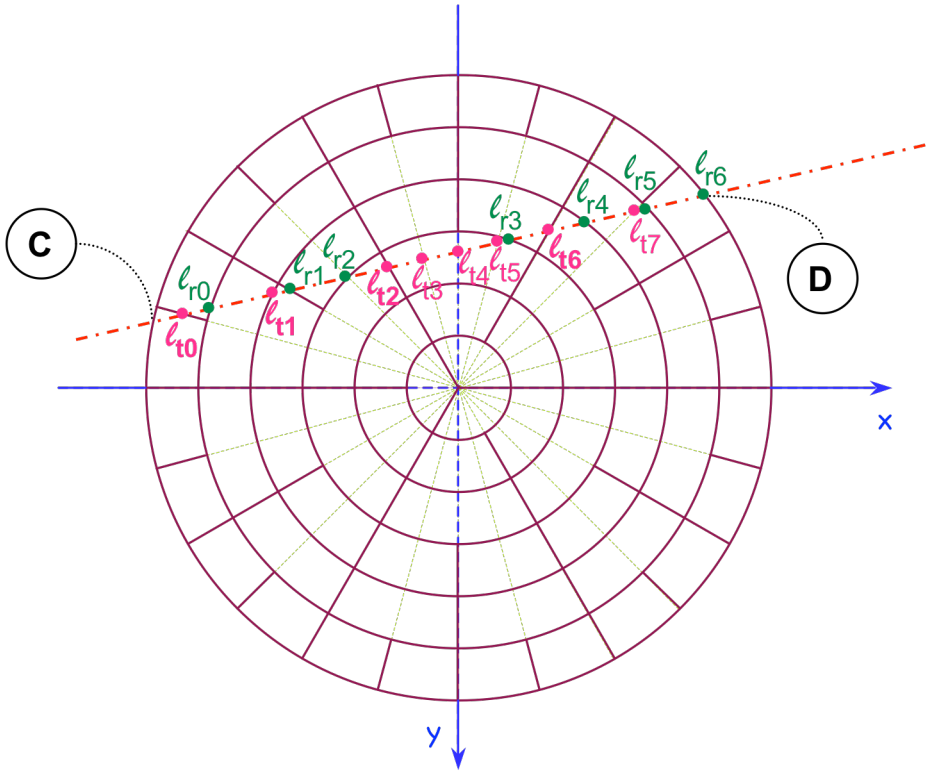


Figure IV.21 – Horizontal Length Definitions

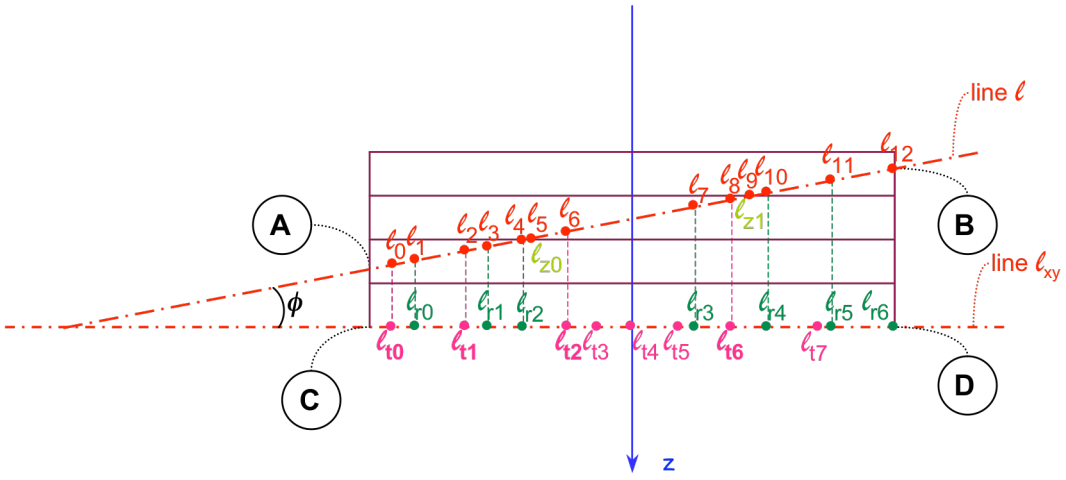


Figure IV.22 – Vertical Length Definitions

a. Z-Slice Lengths

The first step to finding the ℓ_{zi} values is to determine the z-axis value of the top and bottom of the polar reconstruction cylinder.

$$zlow = \mathbf{rxorigin} - \frac{\mathbf{rxsize}}{2}$$
$$zhigh = zlow + \mathbf{rzsize} * \mathbf{dz} \qquad \text{[IV.47]}$$

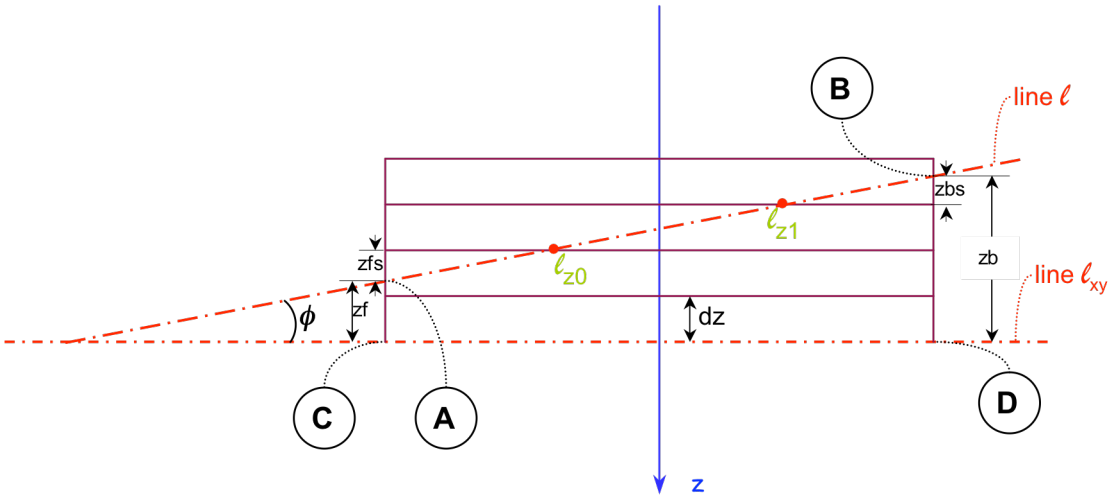


Figure IV.23 – Z-Slice Lengths

Now it can be determined whether the line intersects the object at all. If *zb* and *zb* > *zhigh* or *zb* and *zb* < *zlow* the line *does not* intersect the object and the ray-path processing will stop for the particular ray.

The next step is to determine *zfs* as shown in Figure IV.23. *zfs* is the z-axis distance from *zf* to the next z-slice along the line. *zdir* is a variable that indicates the direction of the line and *izf* is an index of the z-slice (the lowest slice is 0).

If $zf < zlow$ then

$$\begin{cases} zdir = 1 \\ izf = -1 \\ zfs = | zlow - zf | \end{cases} \qquad \text{[IV.48b]}$$

If $zf > zhigh$ then

$$\begin{cases} zdir = -1 \\ izf = \mathbf{nz} \\ zfs = | zhigh - zf | \end{cases} \qquad \text{[IV.48b]}$$

If $zlow \le zf \le zhigh$

$$izf = (int)\left(\frac{zf - zlow}{\mathbf{dz}}\right) \qquad \text{[IV.49a]}$$

if $zf < zb$

$$\begin{cases} zdir = 1 \\ izfa = izf+1 \end{cases} \qquad \text{[IV.49b]}$$

if $zf \ge zb$

$$\begin{cases} zdir = -1 \\ izfa = izf \end{cases} \qquad \text{[IV.49c]}$$

and

$$zfs = | izfa \cdot \mathbf{dz} + zlow - zf | \qquad \text{[IV.49d]}$$

Next the z-slice index, *izb,* where the line leaves the cylinder is determined.

If $zb > zhigh$ $\quad izb = \mathbf{nz}$

If $zb < zlow$ $\quad izb = -1$ $\qquad$ [IV.50]

If $zlow \le zb \le zhigh$

$$izb = (int)\left(\frac{zb - zlow}{\mathbf{dz}}\right)$$

The total number of slices the line passes through is *nzl.*

$$nzl = | izb - izf | \qquad \text{[IV.51]}$$

The length of the line through a whole slice is

$$\ell_{dz} = \mathbf{dz} \cdot \sin\phi \qquad \text{[IV.52a]}$$

Then the ℓ_{zi} values are calculated as

$$\ell_{zi} = zfs \cdot \sin\phi + i \cdot \ell_{dz} \qquad for \qquad i = 0,1,2,\ldots(nzl-1) \qquad \text{[IV.52b]}$$

b. Radial Lengths

The radial boundary lengths, rr_i, are calculated during initialization and do not vary by line location. $rr_{(nr-1)}$ is the largest radii and rr_0 is the smallest. The radial voxel boundaries for a particular line ℓ will be from

$$rr_{(nr-1)},\ rr_{(nr-2)},\ \ldots\ rr_k,\ rr_k,\ \ldots\ rr_{(nr-2)},\ rr_{(nr-1)}$$

$$\text{where } rr_k > r_{min} > rr_{k-1} \qquad \text{[IV.53]}$$

$$nk = nr - k$$

$$nrl = 2 \bullet nk - 1$$

On Figure IV.24 it can be seen that the intersection points of line ℓ_{xy} with the radial boundaries form a set of overlapping right triangles, this relationship is used to calculate the lengths to the radial boundaries.

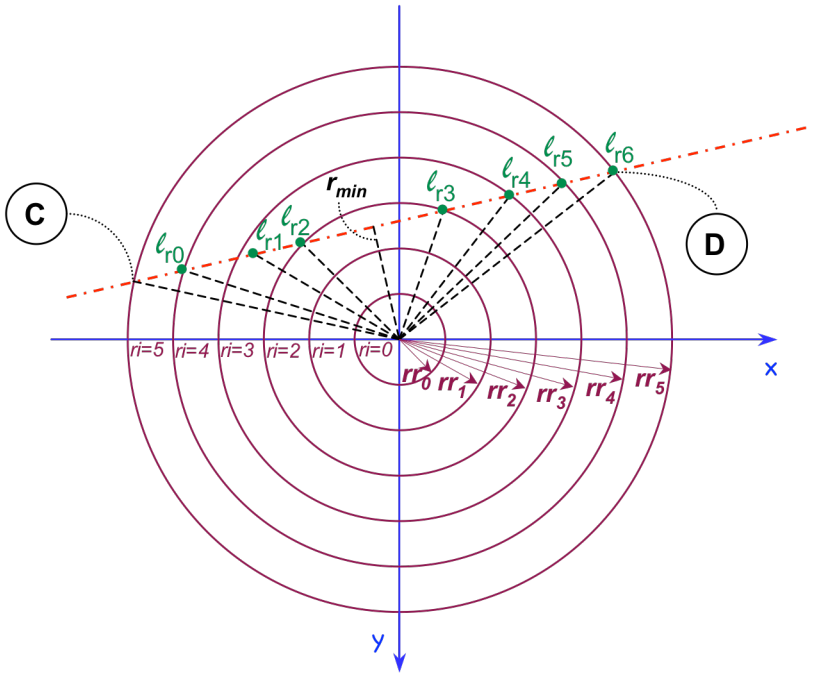


Figure IV.24 – Radial Lengths

By examing Figure IV.25 it can be seen that all the right triangles have a common side of r_{min}, and right triangles on either side of r_{min} are mirror images of each other. So the first step is to determine the lengths b_i.

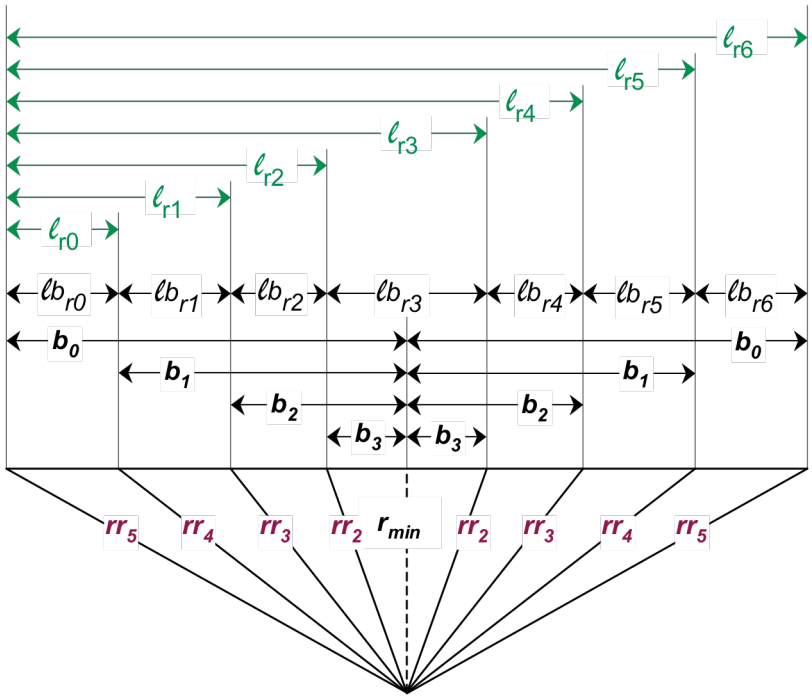


Figure IV.25 – Radial Length Calculation

$$\boldsymbol{b_i} = \sqrt{\left(\boldsymbol{rr}_{(\boldsymbol{nr}-1-i)}\right)^2 - \left(\boldsymbol{r_{min}}\right)^2} \qquad for \quad i = 0,...nk \qquad [IV.54]$$

Then the ℓb_i values are calculated

$$\ell b_i = \boldsymbol{b_i} - \boldsymbol{b_{(i+1)}} \qquad \text{for} \quad i = 0,....(nk-2) \qquad [IV.55a]$$

$$\ell b_{(nk-2)} = 2 \bullet \boldsymbol{b_{(nk-1)}} \qquad [IV.55b]$$

$$\ell b_i = \ell b_{(nrl-i-1)} \qquad \text{for} \quad i = nk, \ldots (\boldsymbol{nrl}-1) \qquad [IV.55c]$$

Finally the $\boldsymbol{\ell_{ri}}$ values are calculated

$$\boldsymbol{\ell_{ri}} = \sum_{n=0}^{i} \ell b_n \qquad for \quad i = 0,...\left(\boldsymbol{nrl}-1\right) \qquad [IV.56]$$

The index, $\boldsymbol{rindex_i}$, which indicates the radial location corresponding to $\boldsymbol{\ell_{ri}}$ must also be determined. The values *ir* on Figure IV.24 show the radial location numbering convention, the outer most radii is ir_{max} and is equal to (*nr*-1). The actual index starts at ir_{max} then

decimates until the inner most index for the line is reach, then the index increments back to ir_{max}.

$$\boldsymbol{rindex_0} = ir_{max}$$

$$\boldsymbol{rindex_i} = \boldsymbol{rindex_{(i\text{-}1)}} - 1 \qquad \text{for} \quad i = 1, \ldots k \qquad \text{[IV.57]}$$

$$\boldsymbol{rindex_i} = \boldsymbol{rindex_{(i\text{-}1)}} + 1 \qquad \text{for} \quad i = (k+1), \ldots (\boldsymbol{nrl}-1)$$

c. Angular Lengths

The angular voxel boundaries vary by radii due to the graduated polar design, this variation could make the calculation of the angular boundaries complicated. A fairly simple way to deal with this is to calculate the lengths for the angular boundaries associated with the smallest angle possible, the angle associated with the outer radii. Depending on the radius many of these voxel boundaries will not be used however it is simpler to decide this when sorting the line lengths than to deal with it when calculating the angular voxel boundaries.

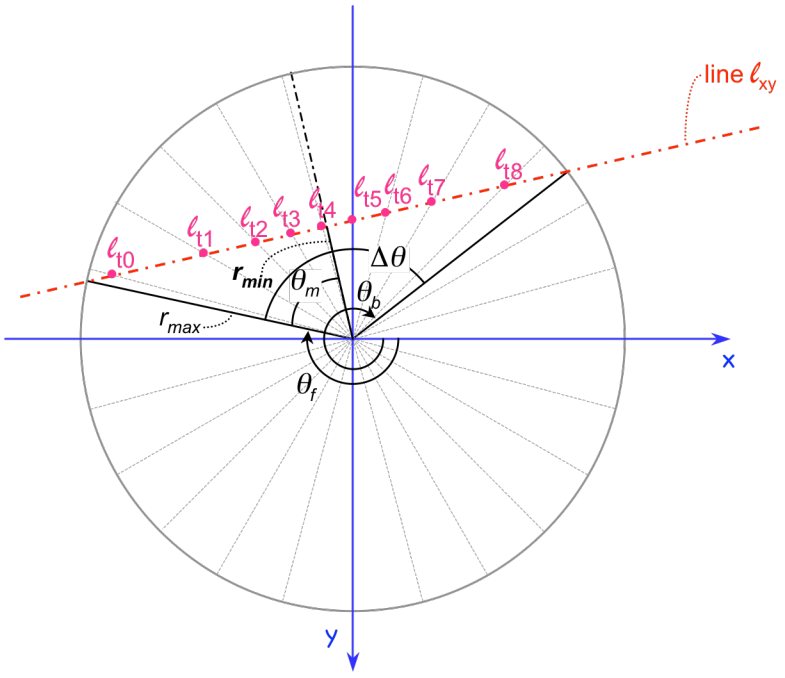


Figure IV.26 – Angular Line Intersects

The lengths on line ℓ_{xy} to the angular boundaries will be calculated in a similar fashion to the radial boundaries lengths as seen in Figure IV.26. However the right triangles in this case will not be symmetric about the $\boldsymbol{r_{min}}$. First the variable θ_m will be determined

$$\theta_m = cos^{-1}\left(\frac{|\boldsymbol{r_{min}}|}{r_{max}}\right) \qquad \text{[IV.58]}$$

Note that where as θ_f and θ_b are relative to the r-θ axis θ_m is just an absolute value, it is half the angle between θ_f and θ_b. It is calculated using $\boldsymbol{r_{min}}$ and r_{max} because it eliminates the issue of an angle greater than π. Another needed variable is $\Delta\theta$

$$\Delta\theta = |\theta_f - \theta_b| \qquad \text{[IV.59]}$$

The calculation of the ℓ_{ti} values involves stepping from θ_f to θ_b, whether this step is in the positive or negative direction angle depends on whether θ_f is greater or less than θ_b and whether the difference between θ_f and θ_b is greater or less than π. The variable *tdir* is used to indicate the direction. Figure IV.27 shows the four possible alignments of θ_f and θ_b and the resulting value of *tdir*.

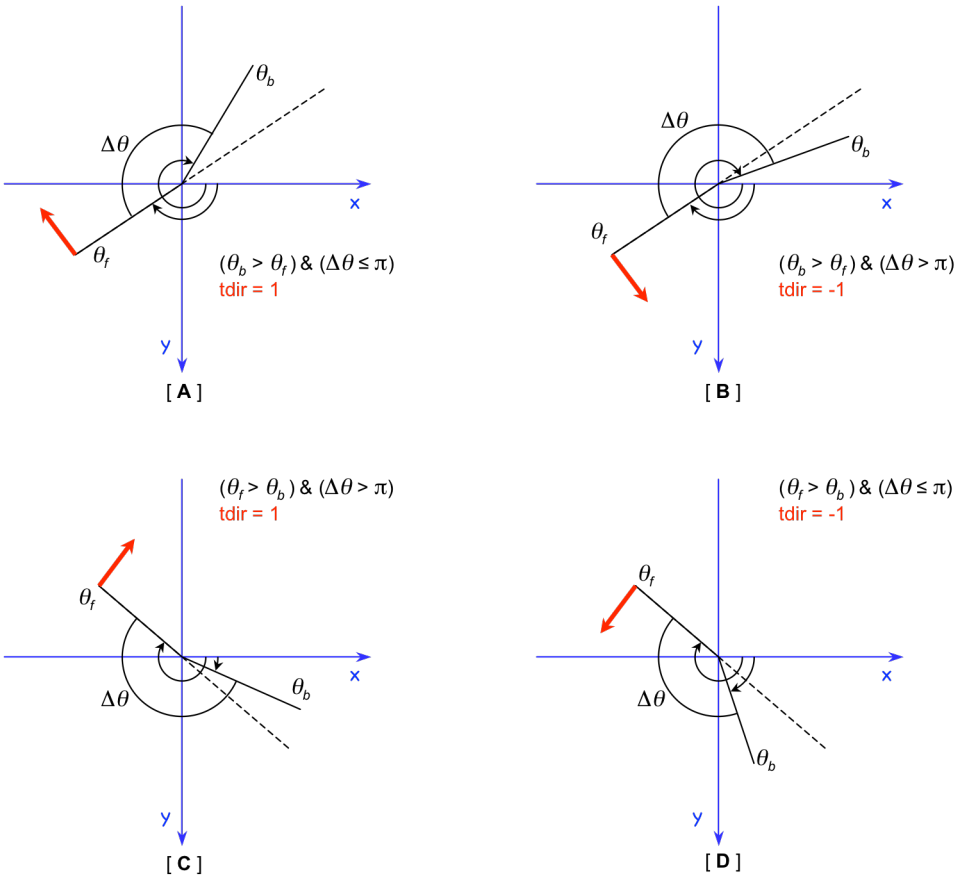


Figure IV.27 – Angular Direction

Other variables are now determined and the calculation of some of them depend on the value of *tdir*. An example for the case for *tdir* = 1 is shown in Figure IV.28. First the angular voxel indices where the line enters and exits the outer radius are calculated.

$$tnum = (int)\left(\frac{\theta_f}{\boldsymbol{xdtheta}}\right)$$
$$tlast = (int)\left(\frac{\theta_b}{\boldsymbol{xdtheta}}\right) \qquad \text{[IV.60]}$$

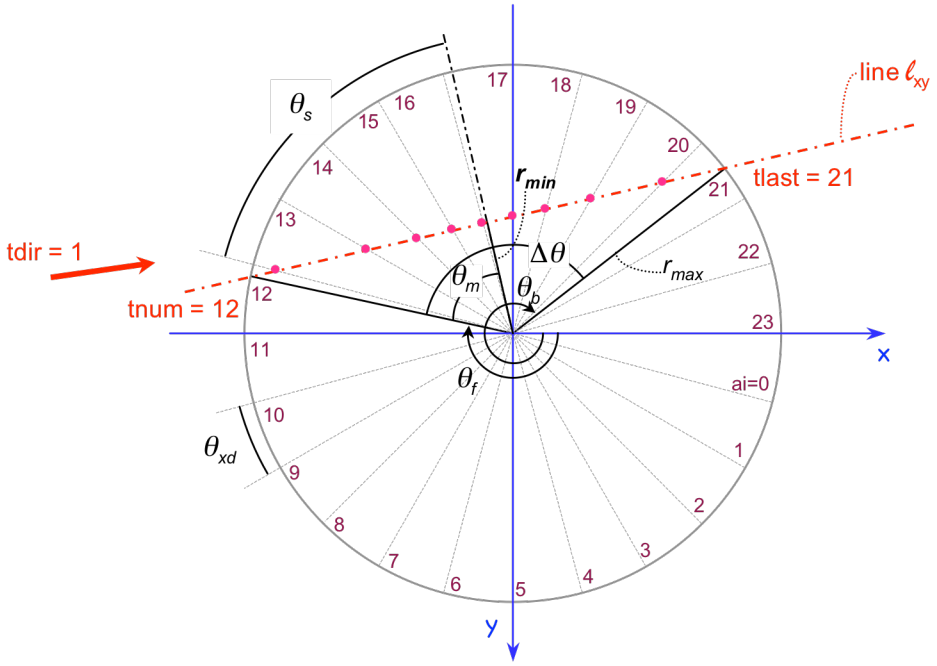


Figure IV.28 – Angular Indices

Then θ_s is determined, this is the angle between θ_m and the first angular voxel edge before or after, depending on *tdir*, the voxel *tnum*. The value *tnum2* is a directional version of the angular location index *ai*. If *tdir* = 1, *tnum2* is clockwise; positive from the x-axis or θ = 0. If *tdir* = -1, *tnum2* is counter-clockwise; negitive direction from the x-axis.

$$tnum2 = tnum$$
$$\theta_s = \theta_m - \left|\theta_f - tnum \cdot \boldsymbol{xdtheta}\right| \qquad \text{for} \quad tdir = -1 \qquad \text{[IV.61a]}$$

$$tnum2 = \boldsymbol{nangles}_{(\boldsymbol{nr-1})} - tnum - 1$$
$$\theta_s = \theta_m - \left|\theta_f - \left(tnum + 1\right) \cdot \boldsymbol{xdtheta}\right| \qquad \text{for} \quad tdir = 1 \qquad \text{[IV.61b]}$$

tnum2 is used to determine the radial dependent variable $\boldsymbol{tstart}_{ri}$. $\boldsymbol{tstart}_{ri}$ is used to determine which angular voxel boundaries should not be used on each radius.

$$\mathbf{tstart}_i = \left((int)\left(\frac{tnum2}{\mathbf{toff}_i}\right)\right)\cdot \mathbf{toff}_i \qquad for \quad i = 0,1,\ldots\left(\mathbf{nr}-1\right) \qquad [IV.62]$$

tnum2 and $\mathbf{tstart}_{ri}$ are shown in Figure IV.29 for *tdir* = -1 and *tnum* = 12.

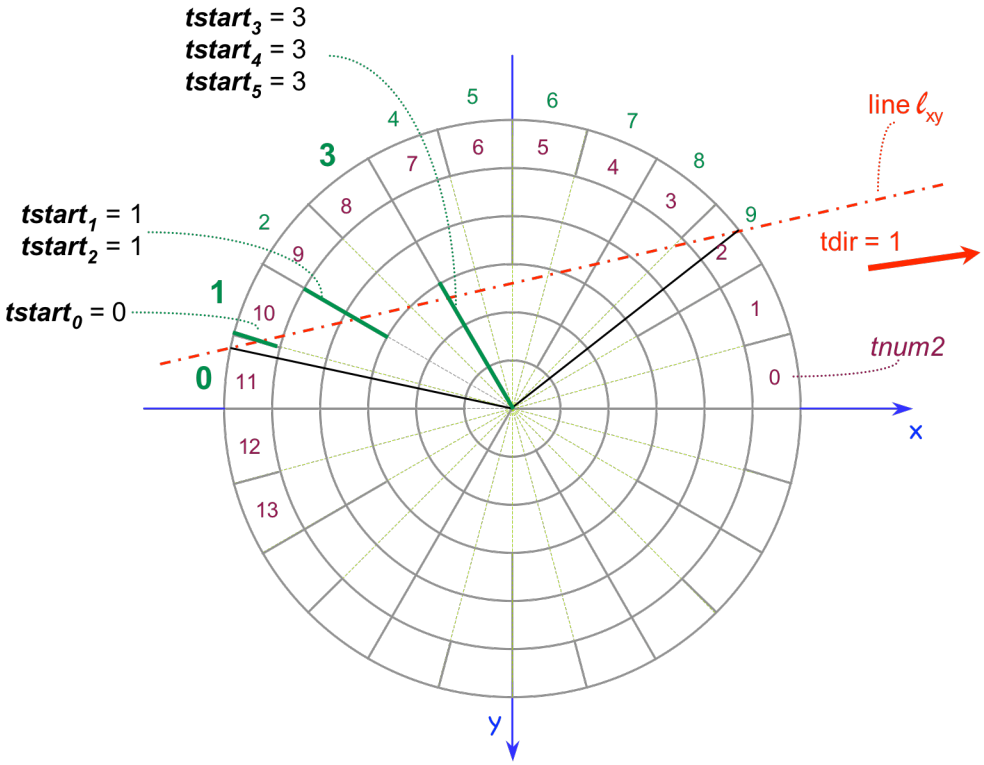


Figure IV.29 – Angular tstart

On Figure IV.26 it can be seen that the angular index will step from *tnum* to *tlast*, so the total number of angular lengths is *ntl*.

$$if \quad \left|tnum - tlast\right| > \frac{\mathbf{nangles}_{(\mathbf{nr}-1)}}{2} \quad then \quad ntl = \frac{\mathbf{nangles}_{(\mathbf{nr}-1)}}{2} - \left|tnum - tlast\right| \qquad [IV.63]$$

$$otherwise \quad ntl = \left|tnum - tlast\right|$$

Now the angular lengths can be calculated. As seen in Figure IV.30 the length *lt0* from θ_t to θ_m is

$$lt0 = \left|\mathbf{r}_{min}\right| \cdot tan\left(\theta_m\right) \qquad [IV.64]$$

Using θ_s the angles between θ_m and each of the angular boundaries is determined and from these angles the lengths ℓ_{ti} are calculated.

$$\theta_i = \theta_s - i \cdot \textbf{xdtheta} \qquad \text{for i = 0,1, ….(}nrl\text{-1)} \qquad \text{[IV.65a]}$$

$$\ell_{ti} = lt0 - \left|\textbf{r}_{min}\right| \cdot tan\left(\theta_s\right) \qquad \text{[IV.65b]}$$

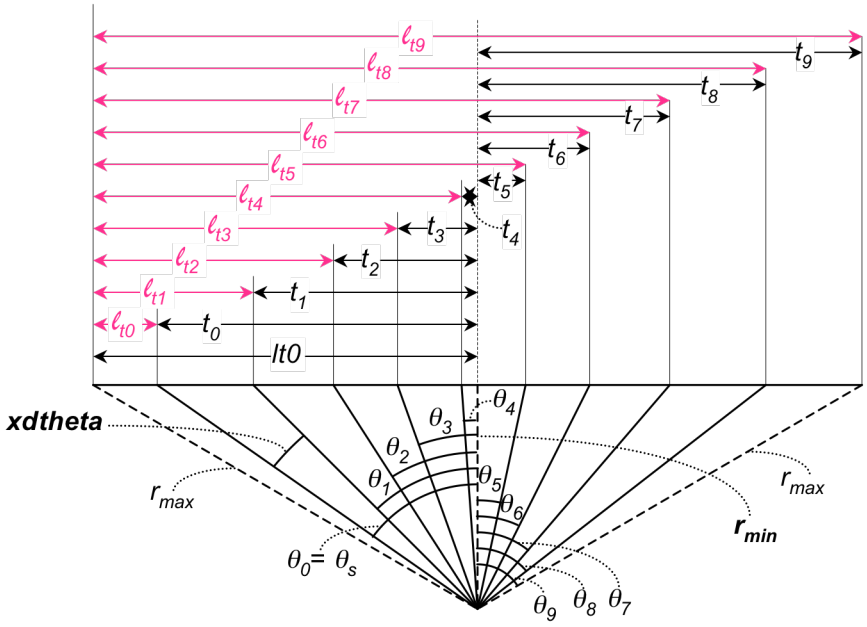


Figure IV.30 – Angular Length Calculation

d. Final Lengths and Indices

Once all the lengths ℓ_{ti}, ℓ_{ti}, and ℓ_{ti} and the initial voxel indices *izf*, *tnum1,* and $\textbf{rindex}_0$ are calculated the actually voxel lengths and indices along the line can be determined. Start at the first voxel and determine which initial length is the smallest. That becomes the length through the first voxel and that is the component (radial, angular or z) that has its voxel index increased or decreased depending on the line direction. This processed is continued through the entire object.

4. Ray Sum

The ray sum for **pcone** is different than the rectangular systems. With the rectangular systems each ray-path is unique. In the polar system the calculated ray-path elements are valid for all the views. So for each pixel on the detector all the views will be determined at

once. The voxel memory indices are for the starting view, the next memory voxel will be shifted in memory by $\boldsymbol{pfactor_{ir}}$. So instead of calculating an entire pixel value at one time the pixel values will be accumulated as each element of the ray-path is processed.

Note: There are a number of special issues regarding the radii less than ***irc****. Due to the lack of time these issues will not be covered in this report. There are also a few special cases in the calculation of the various lengths not discussed, they are however described in the code.*

Appendix A –SCT File

```
!Info needed for simulation
!-simflag y


!Radiograph info
-pfilegeom SINOGRAM
-pfile p2100
-volumeout y
-arange 360.0
-rfile r2100

-nrays 512
-nangles 180
-nslices 1
-pxsize  0.25
-pzsize  0.25
-pxdist  0.25
-pzdist  0.25
-pxcenter 255.5
-pxsrcctr 255.5
-pzcenter 0.0
-pzsrcctr 0.0
-sod  2000.
-sdd  4000.


!Image info
-rxelements  512
-ryelements  512
-rzelements  1
-rxsize  0.125
-rysize  0.125
-rzsize  0.125
-rxorigin  -31.9375
-ryorigin  -31.9375
-rzorigin  -0.0

!CCG info
-rmaxiters 10
-saveiter ALL
-initialize ZERO
!-initial_file o2100
-constraints NONNEG
-fixedvars n
-min_const_file min_const_file
-max_const_file max_const_file
-fix_mask_file fix_mask_file
-fix_image_file fix_image_file
```

Appendix B – RECON Start Up Page

debug status = off
run status = 0
* indicates default value

-NAME	[VALUE]	(RANGE)	DESCRIPTION
-sctfile	[p2100]		SCT filename
-simflag	*[n]	(y/yes or n/no)	Do you want to simulate radiographs? For simulation set initialize = FROMFILE and set initial_file = file to be simulated
-pfile	[p2100]		Projection filename
-pfilegeom	[SINOGRAM]		Storage of ray-sums in the projection file(s):
		(SINOGRAM)	1) Sinogram or sequence of sinograms
		(RADIOGRAPH)	2) Sequence or volume of radiographs
-nrays	*[512]	(1,10000)	Number of rays per projection
-nslices	[1]	(1,10000)	Number of planar slices per projection
-nangles	[180]	(1,1000000)	Number of projection angles
-arange	[360]	(-720,720)	Range of projection angles, (+ is CW, looking down)
-pxdist	[0.25]	(1e-30,1e+30)	Distance between detectors in x (ray spacing) (mm)
-pzdist	[0.25]	(-1e+30,1e+30)	Distance between detectors in z (slice spacing) (mm)
-pxstart	*[0]	(0,9999)	Starting ray number (from 0) to process
-pzstart	*[0]	(0,9999)	Starting slice number (from 0) to process
-pxcenter	[255.5]	(-1e+30,1e+30)	Distance (in pixels) from center of left-most pixel to location of axis-of-rotation on projection
-pzcenter	[0]	(-1e+30,1e+30)	Distance (in pixels) from center of top-most pixel to location center of z-axis on projection
-pxsrcctr	[255.5]	(-1e+30,1e+30)	Distance (in pixels) from center of left-most pixel to center of beam on x-axis
-pzsrcctr	[0]	(-1e+30,1e+30)	Distance (in pixels) from center of top-most pixel to center of beam on z-axis
-sod	[2000]	(1e-30,1e+30)	Distance between source and origin (i.e., center of rotation) (mm)
-sdd	[4000]	(1e-30,1e+30)	Distance between source and center of detector (mm)
-rfile	[r2100]		Reconstructed (transmission) image filename
-volumeout	[y]	(y/yes or n/no)	Do you want to save the image as a volume?
-rxelements	[512]	(1,10000)	Number of reconstruction elements (pixels) in x
-ryelements	[512]	(1,10000)	Number of reconstruction elements (pixels) in y
-rzelements	[1]	(1,10000)	Number of slices to reconstruct in z
-rxsize	[0.125]	(1e-30,1e+30)	Reconstructed pixel size in x (mm)
-rysize	[0.125]	(1e-30,1e+30)	Reconstructed pixel size in y (mm)
-rzsize	[0.125]	(1e-30,1e+30)	Reconstructed pixel size in z (mm)
-rxorigin	[-31.9375]	(-1e+30,1e+30)	Physical location of center of first voxel in x (mm)
-ryorigin	[-31.9375]	(-1e+30,1e+30)	Physical location of center of first voxel in y (mm)
-rzorigin	[-0]	(-1e+30,1e+30)	Physical location of center of first voxel in z (mm)
-rmaxiters	[10]	(0,10000)	Maximum number of iterations

-saveiter	[ALL]		Save interium information in view volumes
		(NONE)	1) None
		(IMAGE)	2) Image only
		(ALL)	3) Image and interium values
-cost_type	*[LST_SQ]		Cost function type
		(LST_SQ)	1) Least Squares
		(POISSON)	1) Poisson
		(LST_SQ_LN)	1) Least Squares with ln
-l1penalty	*[0]	(0,1)	L1 Penalty parameter (0 = no penalty)
-l2penalty	*[0]	(0,1)	L2 Penalty parameter (0 = no penalty)
-matcol	*[n]	(y/yes or n/no)	Use matcol?
-initialize	[ZERO]		Initialization?
		(ZERO)	1) None (all zeros)
		(FROMFILE)	2) From an image file
-constraints	[NONNEG]		What type of constraints?
		(NONE)	1) Unconstrained
		(NONNEG)	2) Non-negative
		(FROMFILE)	3) From constraint files
		(FIXED)	4) Fixed lower and upper constraints
-fix_low_constr	*[1e-10]	(-1e+30,1e+30)	Fixed low constraint
-fix_high_constr	*[1e+25]	(-1e+30,1e+30)	Fixed high constraint
-fixedvars	[n]	(y/yes or n/no)	Do you want to fix any variables?
-noisesig	*[0]	(0,1e+30)	Noise variance
-initial_file	*[]		Initial image filename
-min_const_file	[min_const_file]		Minimum constraint image filename
-max_const_file	[max_const_file]		Maximum constraint image filename
-fix_mask_file	[fix_mask_file]		Fix mask image filename
-fix_image_file	[fix_image_file]		Fix image filename

Enter parameter to be changed and value [-parameter value]
Enter <return> to exit edit mode

Appendix C – RECON Directories, Files, Headers, Subroutines

recon/src Directories	Files in Directory	Headers Used in File	Subroutines in File
nwin			
	main_nw.c		
		global.h recon.h recon_err.h get_ctp.h util_pt.h	
util			
	binary_pt.h get_ctp.h global.h recon_err.h recon.h util_pt.h		
	binary_file.c		
		global.h recon.h recon_err.h binary_pt.h	open_datafile create_datafile append_datafile close_datafile setloc_datafile get_bin_datafile put_bin_data
	ctp_file.c		
		global.h recon.h recon_err.h get_ctp.h util_pt.h	open_ctp_file_read open_ctp_file_write ctp_file_exists get_line ctp_copy is_eol
	error_check.c		
		global.h recon.h recon_err.h	error_check error_set error_reset debug_msg
	get_ctp.c		
		global.h recon.h recon_err.h get_ctp.h util_pt.h io_data.h	read_sctfile read_idlfile edit_ctp get_out_name write_sctfile get_ctp_string get_ctp_value get_ctp_src set_ctp_src check_swi_range

recon/src Directories	Files in Directory	Headers Used in File	Subroutines in File
			check_int_range check_float_range check_double_range check_choice_range
	other_file.c		
		global.h recon.h recon_err.h get_ctp.h util_pt.h	add_ext remove_ext get_mem
	parse_cmd.c		
		global.h recon.h recon_err.h get_ctp.h util_pt.h	parse_cmd_line find_arg
	print_ctp.c		
		global.h recon.h recon_err.h get_ctp.h util_pt.h	print_ctp help
	timing.c		
		global.h recon.h recon_err.h get_ctp.h util_pt.h	get_date mark_timer stop_timer
view			
	io_data.h view_pt.h		
	view_file.c		
		global.h recon.h recon_err.h binary_pt.h get_ctp.h util_pt.h io_data.h view_pt.h	view_file_exists find_first find_seq view_error vspr_error get_dim_from_view get_yslice get_zslice get_volume put_view_spr put_volume put_zslice
	view_io.c		
		global.h recon.h recon_err.h io_data.h view_pt.h	get_data_info read_volume read_volume_ftod write_volume write_volume_dtof write_one_slice

recon/src Directories	Files in Directory	Headers Used in File	Subroutines in File
			read_radio_ftod read_ratio read_sino_ftod read_one_radio read_one_sino find_first_file
apps			
app-util	app-util_pt.h		
	center.c		
	center2.c		
	fft842.c		
	next_file.c		
	rho.c		
cbp			
	cbp_pt.h		
	cbp.h		
	cbp_2d_real.c		
	cbp.c		
ccg			
	ccg_pt.h		
	ccg.h		
	clsrch.c		
		global.h	clsrch
		ccg.h	
		ccg_pt.h	
	getsol.c		
		global.h	getsol
		ccg.h	alloc_mem
		ccg_pt.h	l12penalty_1
			l12penalty_2
			l12penalty_3
			l12penalty_4
			print_progress
	ivecops.c		
		global.h	init_xn
		ccg.h	init_directionn
		ccg_pt.h	calc_dx_xnm
		binary_pt.h	check_varboundar
			calc_newdirection
			restart_wgradient
	update.c		
		global.h	update
		ccg.h	update_matcol

recon/src Directories	Files in Directory	Headers Used in File	Subroutines in File
		ccg_pt.h	update_nomatcol
	vector.c		
		ccg.h	dot_product
		ccg_pt.h	mag
			l_2_norm
			l_1_norm
			add_const
			axpy
			dot_product_ivec
			sum_ivec
			countem
fkrecl			
	fkrecl_pt.h		
	fkrecl.h		
	fkrecl.c		
lcone			
	lcone_ccg_pt.h		
	lcone_ccg.h		
	check_time.c		
		global.h	check_time
	linear_ccg.c		
		global.h	yconst
		recon.h	forward
		recon_err.h	calc_yt
		ccg.h	calc_r
		ccg_pt.h	calc_res
		get_ctp.h	gradient
		util_pt.h	direction1
		io_data.h	direction2
			dd_gnd
			dd_rv
			least_squares
			cost_function
			save_ccg
jproj			
	jproject.h		
	jproject_pt.h		
	jmatrix.c		
		global.h	matrix *
		recon.h	matcol *
		recon_err.h	
		ccg.h	
		jproject.h	
		lcone_ccg.h	
		jproject_pt.h	

recon/src Directories	Files in Directory	Headers Used in File	Subroutines in File
	jproject.c		
		global.h	project
		recon.h	minmax
		recon_err.h	first_point
		ccg.h	length_loop
		jproject.h	doit
		lcone_ccg.h	
		jproject_pt.h	
	ljcone_ccg.c		
		global.h	init_ctp *
		recon.h	init_proc *
		recon_err.h	proc_app *
		get_ctp.h	close_proc *
		util_pt.h	init_info
		binary_pt.h	init_data
		io_data.h	init_constraints
		ccg.h	init_reconradius
		ccg_pt.h	init_image
		jproject.h	init_fixed
		lcone_ccg.h	
		lcone_ccg_pt.h	
	niter_out.c		
		global.h	niter_out *
		recon.h	
		recon_err.h	
		ccg.h	
		jproject.h	
		lcone_ccg.h	
		io_data.h	
nproj			
	nproject.h		
	nproject_pt.h		
	lncone_ccg.c		
		global.h	init_ctp *
		recon.h	init_proc *
		recon_err.h	proc_app *
		get_ctp.h	close_proc *
		util_pt.h	init_info
		binary_pt.h	init_data
		io_data.h	init_constraints
		ccg.h	init_reconradius
		ccg_pt.h	init_image
		nproject.h	init_fixed
		lcone_ccg.h	
		lcone_ccg_pt.h	
	niter_out.c		
		global.h	niter_out *
		recon.h	
		recon_err.h	
		ccg.h	

recon/src Directories	Files in Directory	Headers Used in File	Subroutines in File
		nproject.h	
		lcone_ccg.h	
		io_data.h	
	nmatrix.c		
		global.h	matrix *
		recon.h	matcol *
		recon_err.h	
		ccg.h	
		nproject.h	
		lcone_ccg.h	
		nproject_pt.h	
	nproject.c		
		global.h	nproject
		recon.h	nproject1
		recon_err.h	doproj
		ccg.h	
		nproject.h	
		lcone_ccg.h	
		nproject_pt.h	
pproj			
	pproject.h		
	pproject_pt.h		
	gp2r.c		
		global.h	gp2r
		recon.h	
		recon_err.h	
		pproject.h	
	lpcone_ccg.c		
		global.h	init_ctp *
		recon.h	init_proc *
		recon_err.h	proc_app *
		get_ctp.h	close_proc *
		util_pt.h	init_info
		binary_pt.h	init_polar
		io_data.h	init_data
		ccg.h	init_constraints
		ccg_pt.h	init_image
		pproject.h	init_fixed
		lcone_ccg.h	
		lcone_ccg_pt.h	
	niter_pout.c		
		global.h	niter_out *
		recon.h	
		recon_err.h	
		ccg.h	
		pproject.h	
		lcone_ccg.h	
		io_data.h	
	pmatrix.c		
		global.h	matrix *

recon/src Directories	Files in Directory	Headers Used in File	Subroutines in File
		recon.h	matcol *
		recon_err.h	
		ccg.h	
		pproject.h	
		lcone_ccg.h	
		pproject_pt.h	
	pproject.c		
		global.h	pproject
		recon.h	ray_sum_fwd
		recon_err.h	ray_sum_scl
		ccg.h	ray_sum_bck
		pproject.h	
		lcone_ccg.h	
		pproject_pt.h	
	r2gp.c		
		global.h	r2gp
		recon.h	
		recon_err.h	
		pproject.h	