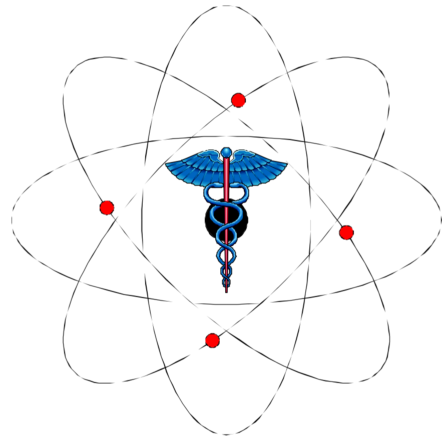
PARAPET



Specification of Common Building Blocks

Document: D/ GENEVA/LABBE-ZAIDI-MOREL/
HAMMERSMITH/THIELEMANS/
BRUNEL/HAGUE/

4.1b/20-01-1999/1/1.4

FINAL DRAFT

Approved by:

C. Labbé — (Author)

F. Wray — (Project manager)

I. INTRODUCTION **6**

II. DEDICATED IMPLEMENTATION OF ANALYTICAL RECONSTRUCTION ALGORITHMS **6**

II.1. PROMIS 7

II.2. FAVOR 7

II.3. FORE+FBP 7

II.4. SSRB 7

III. BUILDING BLOCKS **8**

III.1. PROGRAMMING FRAMEWORK 8

III.2. GENERAL CONVENTIONS USED TO DESCRIBE FUNCTIONS 9

IV. GENERAL CONVENTIONS **10**

IV.1. BASIC TYPES 10

IV.2. UNITS 10

IV.3. COORDINATE SYSTEM FOR IMAGE DATA 10

IV.4. NAMING CONVENTIONS 11

IV.5. FILE CONVENTIONS 11

IV.6. ERROR HANDLING 11

IV.7. CODE TYPE SETTING 11

V. DOCUMENTATION CONVENTIONS **12**

VI. CLASSES FOR PATIENT DATA **12**

VI.1. CLASS STUDYINFO 12

VI.2. CLASS SCANINFO 12

VI.3. CLASS PETSTUDY 13

VI.4. CLASS PETIMAGEDATA 14

VI.5. CLASS PETACQUISITIONDATA 14

VI.6. CLASS PETTRANSMISSIONDATA 14

VI.7. CLASS PETBLANKDATA 15

VII. CLASSES FOR IMAGE DATA **15**

VII.1. CLASS FOR 3D IMAGES (CLASS PETIMAGEOFVOLUME) 15

VII.2. CLASS FOR 2D IMAGE (CLASS PETPLANE) 17

VIII. CLASSES FOR PROJECTION DATA **18**

VIII.1. DESCRIPTION 18

VIII.2. CLASS PETSINOGRAM 20

VIII.3. CLASS PETVIEWGRAM 22

VIII.4. CLASS PETSEGMENT 23
VIII.5. CLASS PETSEGMENTBYVIEW 26
VIII.6. CLASS PETSEGMENTBYSINOGRAM 27
VIII.7. CLASS PETSINOGRAMOFVOLUME 29

IX. TRUNCATING, TRIMMING, OFFSETTING, MASHING AND ZOOMING 34
IX.1. DESCRIPTION: 34
IX.2. LOCATION: 35
IX.3. ON PROJECTION DATA 35
IX.4. ON IMAGE DATA 36
IX.5. ALGORITHM 36

X. CLASSES FOR SCANNER INFORMATION 37
X.1. CLASS PETSCANNERINFO 37
X.2. CLASS PETSCANINFO 39

XI. CLASSES FOR RECONSTRUCTION 41
XI.1. DESCRIPTION 41
XI.2. LOCATION 41
XI.3. CLASS PETRECONSTRUCTION 41
XI.4. CLASS PETANALYTICRECONSTRUCTION 42
XI.5. CLASS PETITERATIVERECONSTRUCTION 42
XI.6. CLASS PETRECONSTRUCTION2D 42
XI.7. CLASS PETANALYTICRECONSTRUCTION2D 43
XI.8. CLASS RECONSTRUCT2DFBP 43

XII. CLASSES FOR TENSORS 44
XII.1. VECTORS WITH INDEX NOT STARTING AT 0 (CLASS VECTORWITHOFFSET) 44
XII.2. CONSTRUCTORS 45
XII.3. OPERATORS 45
XII.4. DATA ACCESS METHODS 46
XII.5. PROTECTED MEMBERS 47
XII.6. VECTORS CONTAINING NUMERIC ELEMENTS (CLASS NUMERICVECTORWITHOFFSET) 47
XII.7. CONSTRUCTORS 48
XII.8. OPERATORS 48
XII.9. NUMERIC VECTORS WITH SOME EXTRA OPERATIONS (CLASS TENSOR1D<T>) 49
XII.10. CONSTRUCTORS 51
XII.11. DATA ACCESS METHODS 51
XII.12. ARITHMETIC METHODS 51

XII.13. I/O METHODS 52

XII.14. A BASE CLASS FOR HIGHER DIMENSIONAL VECTORS (CLASS TENSORBASE<T, NUMBER>) 53

XII.15. CONSTRUCTORS 54

XII.16. ARITHMETIC METHODS 54

XII.17. 2D TENSORS (TENSOR2D<NUMBER>) 55

XII.18. CONSTRUCTORS 56

XII.19. OPERATORS 56

XII.20. DATA ACCESS METHODS 57

XII.21. I/O METHODS 58

XII.22. IMPLEMENTATION DETAILS 58

XII.23. 3D TENSORS (CLASS TENSOR3D<NUMBER>) 58

XII.24. 4D TENSORS (CLASS TENSOR4D<NUMBER>) 60

XII.25. SOME FUNCTIONS TO MANIPULATE TENSORS 61

XII.26. CONVERSION OF TENSOR OBJECTS : CONVERT 63

XII.27. CLASSES WITH INFORMATION ON THE (BUILT-IN) NUMERIC TYPES 64

XII.28. TEMPLATE <CLASS NUMBER> CLASS NUMERICINFO 65

XII.29. A CLASS FOR SPECIFYING BYTE ORDER (CLASS BYTEORDER) 66

XIII. CLASSES FOR FILTERS 67

XIII.1. DESCRIPTION 67

XIII.2. LOCATION 67

XIII.3. SOFTWARE IMPLEMENTATION 67

XIII.4. 1D FILTERS 68

XIII.5. 2D FILTERS 70

XIV. CLASSES FOR PROJECTION OPERATORS 71

XIV.1. BACKPROJECTION UTILITIES 71

XIV.2. FORWARD PROJECTION UTILITIES 88

XV. INTERFILE SUPPORT 96

XV.1. CLASSES FOR PARSING A FILE WITH INTERFILE-TYPE KEYS (CLASS KEYPARSER AND KEYARGUMENT) 96

XV.2. A CLASS FOR THE COMMON KEYWORDS IN THE INTERFILE HEADER (CLASS 98

XV.3. A CLASS FOR INTERFILE HEADERS CONTAINING IMAGES (CLASS INTERFILEIMAGEHEADER) 98

XV.4. A CLASS FOR INTERFILE HEADERS CONTAINING PROJECTION DATA (CLASS INTERFILEPSOVHEADER) 99

XV.5. INTERFILE FUNCTIONS 99

XVI. MISCELLANEOUS UTILITY FUNCTIONS 100

XVI.1. FAST FOURIER TRANSFORM UTILITIES 100

XVI.2. VARIOUS UTILITY FUNCTIONS 102

XVII. REFERENCES 106

XVIII. ANNEXES 108

ANNEX 1 : LIST OF ALL PARAPET SOURCE CODES NEEDED FOR PET IMAGE RECONSTRUCTION 108

ANNEX 2: SYMMETRIC VOXELS AND CORRESPONDING V PARAMETERS FOR POSITIVE RING INDEX DIFFERENCES Δ 112

ANNEX 3: SYMMETRIC VOXELS AND CORRESPONDING V PARAMETERS FOR NEGATIVE RING INDEX DIFFERENCES Δ 113

I. INTRODUCTION

The objective of this document is to identify, develop and describe common building blocks which can be used in the reconstruction of the algorithms to be implemented in tasks 4.2 (reprojection algorithm - PROMIS) and 4.3 (Fourier rebinning algorithm - FORE). These building blocks can be used for both analytical reconstruction algorithms (WP4), and iterative reconstruction algorithms (WP5).

II. DEDICATED IMPLEMENTATION OF ANALYTICAL RECONSTRUCTION ALGORITHMS

To illustrate the building blocks we need for most reconstruction algorithms, we list as examples some analytic algorithms. Various analytical, iterative or rebinning methods have been proposed to carry out three-dimensional (3D) reconstruction of data acquired by multi-ring clinical PET tomographs. Details of the reconstruction algorithms will not be presented here, as it was already described in the deliverable D1.3 or elsewhere [Colsher, 1980; Defrise et al., 1989; Kinahan et al., 1989; Comtat et al., 1994; Herman, 1995]. Whereas iterative algorithms are not considered here, the reprojection algorithm of Kinahan and Rogers [1989], also dubbed PROMIS (Project Missing Data), the FAVOR algortihm (Fast VOlume Reconstrcution) [Defrise et al., 1992), the FORE (FOurier REbinning) [Defrise et al., 1995], and SSRB (Single-Slice ReBinning) [Daube-Witherspoon et al., 1987] algorithms are briefly summarized in order to highlight the common routines generally used in analytical algorithms.

The PROMIS algorithm is thoroughly used in clinical 3D PET. Oblique projection data that are not accessible within the finite axial extent of the scanner and hence remain unmeasured, are estimated by forward-projecting through a low-statistics first-pass image reconstructed from transaxial projections only (using 2D FBP). The Colsher filter and 3D backprojection is then used to recover the image from the completed planar projections.

The FAVOR algorithm, is based on 3D FBP but the utilisation of an appropriate filter, the FAVOR filter, obviates the need for complete 2D parallel projections. Avoiding the forward projection step, the FAVOR filter reduces to the 1D Ramp filter for all oblique projections and 3D FBP can be calculated even though the projection are truncated.

The FORE algorithm is a method to rebin oblique projection data into a transaxial, 2D data set, with one sinogram for each transaxial slice. FORE involves rebinning of sinogram elements in frequency space. After the Fourier rebinning step (FORE), the rebinned data set may be reconstructed with any 2D reconstruction method. Conventional filtered backprojection is used in this implementation, resulting in the FORE+FBP algorithm.

The SSRB algorithm rebins 3D data to 2D slices as well, but with an approximation. 2D FBP can then be used on each slice (SSRB+FBP).

The steps for each one of the algorithms described above are summarized below.

II.1. PROMIS

- 2D FBP
  - Read direct sinograms
  - Extract 1D transaxial projections
  - Filter 1D transaxial projections
  - Backproject 1D filtered projections with Ramp filter => initial 2D image estimate
- 3D FBP
  - Read sinograms
  - Extract 2D projections
  - Forward project missing data estimated from initial 2D image estimate
  - Filter completed 2D projections with Colsher filter
  - Backproject filtered 2D projections

II.2. FAVOR

- Direct sinograms
  - Read direct sinograms
  - Extract 2D direct projections
  - Filter direct 2D projections with FAVOR filter
  - Backproject filtered 2D projections
- Oblique sinograms
  - Read oblique sinograms
  - Extract 1D transaxial projections
  - Filter transaxial 1D projections with Ramp filter
  - Backproject filtered 1D projections

II.3. FORE+FBP

- Rebin sinograms
  - Read sinograms
  - 2D Fourier transform sinograms
  - Rebin Fourier transformed sinograms
  - Inverse Fourier transform direct rebinned sinograms
- 2D FBP
  - Extract 1D transaxial projections
  - Filter 1D transaxial projections with Hamming filter
  - Backproject 1D filtered projections

II.4. SSRB

- Rebin sinograms
  - Read sinograms
  - Rebin oblique sinograms onto direct sinograms

• 2D FBP
- Extract 1D transaxial projections
- Filter 1D transaxial projections with Ramp filter
- Backproject 1D filtered projections

In general, analytic algorithms usually consist of three main steps: reading and writing data (images or sinograms), filtering of the projections, and 3D backprojection of the filtered projections into the image volume. The latter step represents a considerable part of the total reconstruction time of data acquired on usual cylindrical, multi-ring positron tomographs [Egg96,Egg97]: from about 45% using the reprojection algorithm [Kin89] to over 60% using the FAVOR algorithm [Def92], or more, depending on the particular implementation. Iterative algorithms use the backprojection and forward projection operations repeatedly, and are particularly sensitive to their accuracy and efficiency.

III. BUILDING BLOCKS

III.1. PROGRAMMING FRAMEWORK

The building blocks needed for the image reconstruction algorithms are shown in Figure III.1 and can be summarized as follows:

- Information about the data to be reconstructed (PET scanner characteristics, algorithm type, ...)
- Memory allocations of multi-dimensional arrays or tensors (1D, 2D, 3D or 4D)
- Reconstruction algorithm building blocks including
    - Filtering
    - Backprojection
    - Forward projection
    - 2D FBP
    - Zooming
    - Fast Fourier Transform

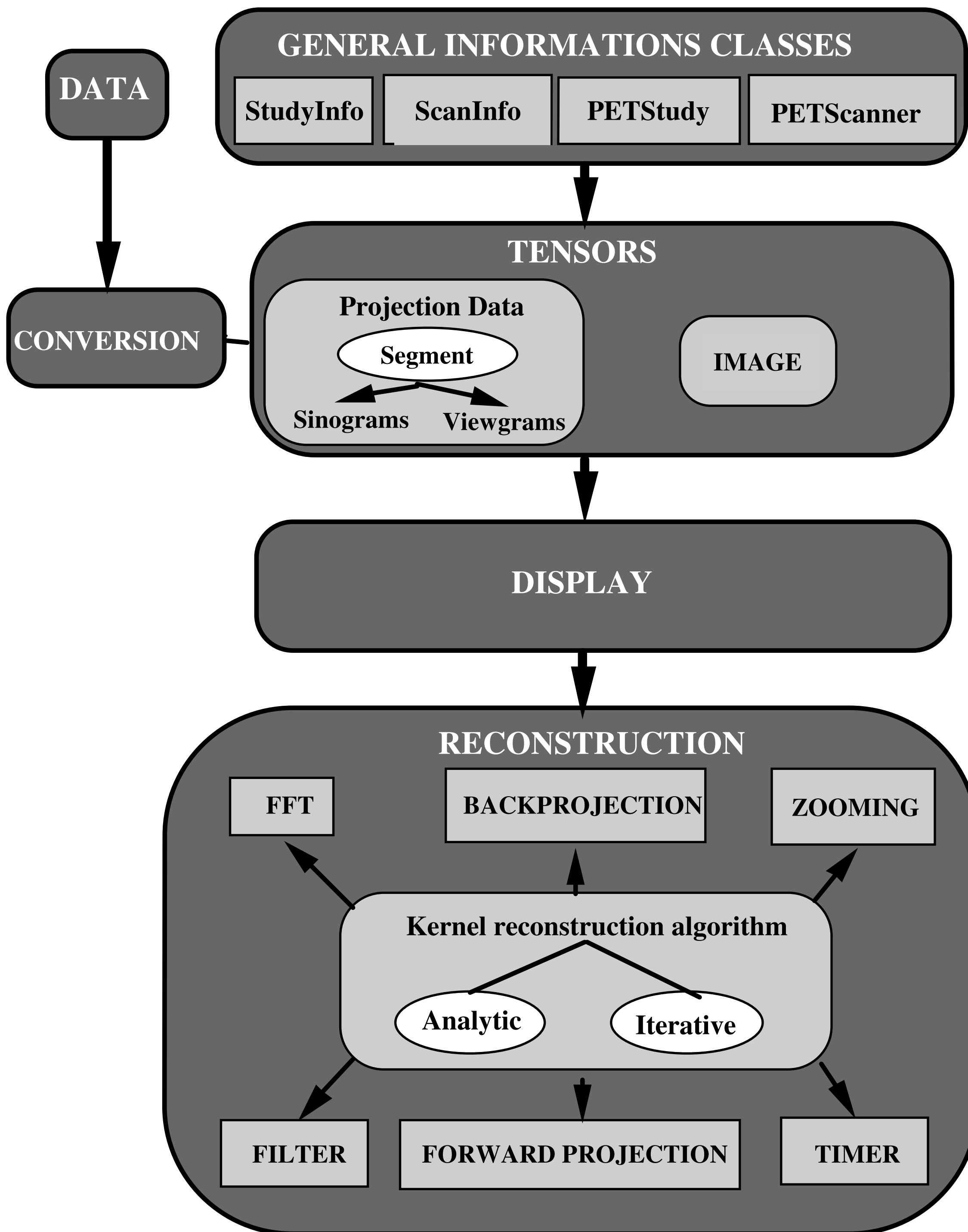


Figure III.1: *Chart showing the major building blocks needed for image reconstruction.*

III.2. GENERAL CONVENTIONS USED TO DESCRIBE FUNCTIONS

The documentation is organized as follows:

- description of the class or function

- location of source code (See annex 1 for the complete list of files and locations)

- name of classes and functions with the list of their arguments,

If it is a class,

    - content of the class

- constructors
- data access methods
- derived methods
- software implementation

If it is a function,

- content of the function with the list of its arguments and argument types
- algorithm
- software implementation

IV. GENERAL CONVENTIONS

IV.1. BASIC TYPES

We use Real and Int (currently typedef'ed to float and int respectively in *pet_common.h*) at places where we could want a different type later. This convention is not very well established yet.

IV.2. UNITS

Distances are in millimetre.

Relative times are in milliseconds.

Volumes are in millilitre.

IV.3. COORDINATE SYSTEM FOR IMAGE DATA

x-axis : horizontal axis, pointing right when looking from the bed into the gantry

y-axis : vertical axis, pointing upwards

z-axis : the scanner axis, pointing from the gantry towards the bed

The origin of the X and Y axes are located on the central axis of the PET scanner and the Z origin is located in the middle of the first ring (i.e at the opposite side of the bed).

Conventional axes and units within one transaxial slice of the target image are shown in figure IV.1.

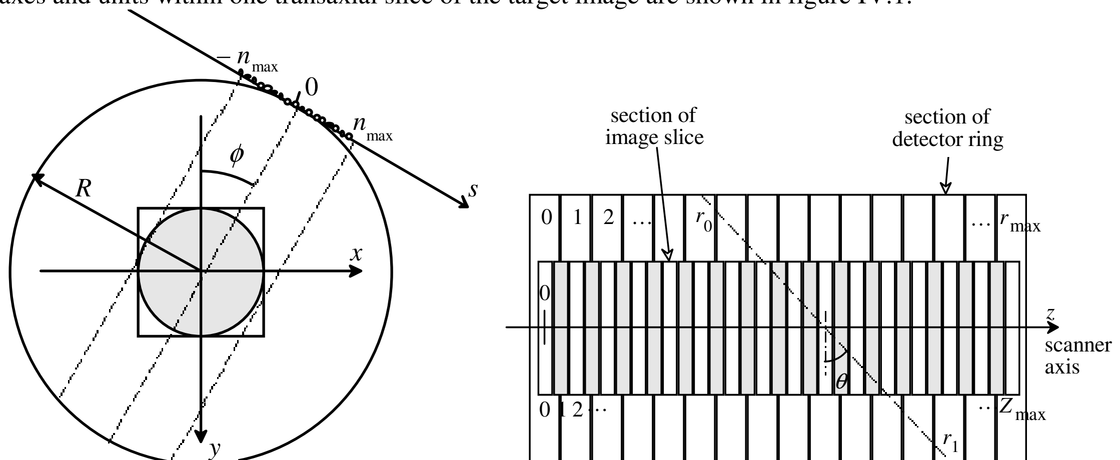


Figure IV.1: (Left) *Axes and units within one transaxial slice of the target image. The transaxial section of the field of view is shown as a grey circle. The number of measured projection elements along the s-axis is odd, so that s =0 is positioned at the center of the central projection element. The angles* θ *and* ϕ *define the direction of the line-of-response* **(Right)** *Sketch of main axes, units and angles used in the cylindrical scanner geometry (axial section, not to scale);*

IV.4. NAMING CONVENTIONS

. Types start with capitals, every word is capitalised, no underscores, e.g. *StudyInfo*.

. Most class names start for the moment with PET. This is a bit tedious though , but the current version of *gcc* does not support C++ namespaces.

. Variables, methods and members are lower case, underscores between different words, e.g. *voxel_size*.

. Variables , methods and members indicating

- a number of things start with *num*, e.g. num_gates.
- the number of an item in a sequence end with *num*, e.g. gate_num.
- a relative time (normally with respect to the scan start) end with *rel_time*, e.g. *tracer_injection_rel_time*.

. Variables quoted in this task are expressed in italic and classes in bold.

IV.5. FILE CONVENTIONS

. Most classes have their own *.h* and *.cxx* files. File names of such classes are simply *ClassName.h* and *ClassName.cxx* (preserving capitals).

IV.6. ERROR HANDLING

. It was originally planned to use C++ exceptions for error handling, allowing the caller of the routines to catch the error, and to provide a sensible handler. However, the current version of *gcc* does not support exceptions very well when optimising the code. So, currently when an error occurs, the program is simply aborted (after writing a diagnostic message).

. In many places, validatity of input arguments, or of the state of an object is checked by *assert* macros. This code is only compiled when the *_DEBUG* preprocessor macro is defined, such that a "production" version of the programs will not be slowed down.

IV.7. CODE TYPE SETTING

All the code which has been extracted from source code and included in this task is written in font Courier 8 as the example below shows:

```
int tracer_injection_time;         // relative to time, in seconds
Real injected_volume;              // ml
```

Comments (started with double slash "//") at the end has been added in order to explain the argument or the variable.

V. DOCUMENTATION CONVENTIONS

Every class has its own section. The title of the section indicates from which classes it is derived (using C++ syntax). There are subsections for public members, public methods, constructors, protected members, protected methods and implementation details. Note that protected members and methods are only available in the class itself and its derived classes.

VI. CLASSES FOR PATIENT DATA

These are the classes containing general information, like the patient information, how many scans, etc. The reconstruction methods do not need to know about them. Instead they would be interested in **PETSinogramOfVolume** etc.

VI.1. CLASS STUDYINFO

VI.1.1. Description

The first part of the Interfile header. Contains administrative information on the study. More details of the variables are described in Task 1.1.

VI.1.2. Location

- *include/study.h*

VI.1.3. public

```
string institution;
string contact_person;
string data_description;
string study_ID;
string patient_name;
string patient_ID;
string patient_DOB; // date of birth
string patient_sex;
string patient_dexterity;
string examination_type;
string isotope;
string radiopharmaceutical;
string date;
string time;
int tracer_injection_time;        // relative to time, in seconds
Real injected_volume;             // ml
string patient_orientation;
string patient_rotation;
bool decay_corrected;
string process_status;
```

VI.2. CLASS SCANINFO

VI.2.1. Description

This class contains all the information for one scan (or image).

VI.2.2. Location

- *include/study.h*

VI.2.3. public

```
Real duration;
Real start_rel_time;
Real bed_rel_position;
int frame_num;
int gate_num;
int data_num;
int bed_num;
// other gating info should be inserted here
int energy_window_num;
// other energy window info should be inserted here
```

VI.3. CLASS PETSTUDY

VI.3.1. Description

This is the base class that contains everything common to all study classes. We define a ‘study’ as a collection of acquisitions (all of the same type), or images, of one patient, where the scans are taken in one sequence. One could have related studies of one patient, e.g. made on two different days. This organisation is not part of PETStudy, but the task of some database system. Much of this organisation is inspired by the proposed Interfile standard for PET (see also deliverable D1.1).

VI.3.2. Location

- *include/study.h*

VI.3.3. Terminology

‘Frames’ are time frames.

‘Data types’ have a fixed meaning for acquired data. Possible values are then ‘prompts’, ‘delayed’, ‘multiples’, ‘corrected prompts’. For images, the data type is essentially free, and can be used for output of kinetic models.

VI.3.4. public

```
StudyInfo info;
PETScannerInfo scanner;
int num_gates;
// here should be more info on gates
int num_acquisitions;
// different bed positions and/or time frames
int num_data_types; /
char **data_type_info;
int num_energy_windows;
vector<float> lower_energy_limit;
vector<float> upper_energy_limit;
vector<string> energy_info;
float start_bed_position;
float end_bed_position;
bool stepped_bed;
ScanInfo& scan_info(int frame = 1, int gate = 1, int data = 1, int bed = 1) throw
(ScanInfoNotFound);
```

VI.3.5. protected

```
vector<ScanInfo*> scan_infos;
find_element(int frame, int gate, int data, int bed, int energyw) throw (ScanInfoNotFound);
```

This function is used to get the index into the vector scan_info which corresponds to the desired scan. It is also used in the derived classes PETImageData and PETAcquisitionData to retrieve the corresponding image or acquisition.

VI.4. CLASS PETIMAGEDATA

VI.4.1. Description

A study containing the images and all its information.

VI.4.2. Location

- *include/study.h*

VI.4.3. Class

Class PETImageData : public PETStudy{}

VI.4.4. public

```
PETImageOfVolume& volume(int frame = 1, int gate = 1, int data = 1, int bed = 1) throw
(ScanInfoNotFound);
```

VI.4.5. private

```
vector<PETImageOfVolume* > volumes;
```

VI.5. CLASS PETACQUISITIONDATA

VI.5.1. Description

A study containing one acquisition sequence and all its information.

VI.5.2. Location

- *include/study.h*

VI.5.3. Class

Class PETAcquisitionData : public PETStudy{}

VI.5.4. public

```
PETSinogramOfVolume& sino(int frame = 1, int gate = 1, int data = 1, int bed = 1, int energyw
= 1) throw (ScanInfoNotFound);
```

VI.5.5. private

```
vector<PETSinogramOfVolume * > sinos;
```

VI.6. CLASS PETTRANSMISSIONDATA

VI.6.1. Description

This class contains information specific to transmission acquisitions (like the type of external sources, their activity, etc.)

VI.6.2. Location

- *include/study.h*

VI.6.3. Class

Class PETTransmissionData : public PETAcquisitionData{}

VI.7. CLASS PETBLANKDATA

VI.7.1. Description

This class contains information specific to transmission acquisitions (like the type of external sources, their activity, etc.)

VI.7.2. Location

- *include/study.h*

VI.7.3. Class

Class PETBlankData : public PETAcquisitionData{}

VII. CLASSES FOR IMAGE DATA

VII.1. CLASS FOR 3D IMAGES (CLASS PETIMAGEOFVOLUME)

VII.1.1. Description

This class is used to represent a reconstructed volume and contains information specific to images such as voxel size, origin of the image volume, scale factor. Some methods are available in order to extract or set a plane of the volume (stack of *N* planes)

VII.1.2. Location

- *include/imagedata.h*

VII.1.3. Class

```
class PETImageOfVolume : public Tensor3D<float> {
public:
  PETImageOfVolume(const Tensor3D<float> & v, const Point3D& origin, const Point3D& voxel_size)
PETImageOfVolume::PETImageOfVolume(const PETScanInfo& scan_info,
                                                   const float zoom,
                                                   const float Xoff, const float Yoff,
                                                   const int xy_size)


PETImageOfVolume::PETImageOfVolume(
                                                   const PETScanInfo& scan_info,
                                                   const float zoom,
                                                   const float Xoff, const float Yoff,
                                                   const bool make_xy_size_odd)

PETImageOfVolume PETImageOfVolume::get_empty_copy() const
```

```
  int get_x_size() const                 // Returns the length along x-axis
  int get_y_size() const                 // Returns the length along y-axis
  int get_z_size() const                 // Returns the length along z-axis
  int get_min_x() const                  // Returns the first index value along x-axis
  int get_min_y() const                  // Returns the first index value along y-axis
  int get_min_z() const                  // Returns the first index value along z-axis
  int get_max_x() const                  // Returns the last index value along x-axis
  int get_max_y() const                  // Returns the last index value along z-axis
  int get_max_z() const                  // Returns the last index value along y-axis

  Point3D get_origin()  const            // Get the values x, y and z of the origin point along
             the three axes.
  Point3D get_voxel_size()  const        // Get the value of  x, y and z for the the voxel size
             along the three axes.

  Tensor2D<float>& get_plane(int z)
  const Tensor2D<float>& get_plane(int z) const
  void set_plane(Tensor2D<float>& p, int z)

  Point3D origin;                        // N.B. Point3D is a class containing the coordinates of
             x, y and z for the origin
  Point3D voxel_size;

  void set_origin(Point3D &origin_v);    // Set the new coordinate (x,y,z) of the origin

  void set_voxel_size(Point3D &voxel_size_v); // Set the new voxel size along x,y and z axes
}
```

VII.1.4. Constructors

VII.1.4.1. PETImageOfVolume(const Tensor3D<float> & v, const Point3D& origin, const Point3D& voxel_size)

This constructor takes a Tensor3D<float> object, and origin (a displacement vector in physical units with respect to the origin of the coordinate system) and voxel_size information.

VII.1.5. Data access members

VII.1.5.1. int get_x_size() const, int get_y_size() const, int get_z_size() const

These methods return the length along respectively the x, y and z axis

VII.1.5.2. int get_min_x() const, int get_min_y() const, int get_min_z() const, int get_max_x() const, int get_max_y() const, int get_max_z() const

These methods return the ranges of the indices to access the data following the conventions in the Tensor classes: *min* and *max* give the first and the last index in the range.

VII.1.5.3. Point3D get_origin() const, Point3D get_voxel_size() const

These methods return the value for the origin and voxel size along the three axes in the volume.

VII.1.5.4. Tensor2D<float>& get_plane(int plane_num), const Tensor2D<float>& get_plane(int plane_num) const

Two methods return a plane of the volume.

VII.1.5.5. void set_plane(Tensor2D<float>& p, int plane_num)

This method replaces a plane of the volume *p*.

VII.1.5.6. void set_origin(Point3D &origin_v)

```
This method set the new coordinate of the origin which is belonged to Point3D class
```

VII.1.5.7. void set_voxel_size(Point3D &voxel_size_v)

```
void set_voxel_size(Point3D &voxel_size_v); // Set the new voxel size along x,y and z axes,
voxel_size_v belonged to Point3D class
```

VII.1.6. Implementation details

This class is derived from *Tensor3D<float>* which means that all data has to be stored in memory. A 95 planes volume of the Ecat 966, with 128 pixels in each plane, needs about 6MB. Restricting access to a plane by plane basis would mean completely rewriting the forward and backward projection routines, and making them much less efficient.

VII.2. CLASS FOR 2D IMAGE (CLASS PETPLANE)

VII.2.1. Description

This class is used to represent a reconstructed image plane and is derived from Tensor2D<float>.

VII.2.2. Location

- *include/imagedata.h*

VII.2.3. Class

```
Class PETPlane : public Tensor2D<float>public {
  int plane_num;                               // Plane number of image
  Point3D origin;
  Point3D voxel_size;
  PETPlane( const Tensor2D<float> &p, const int p_num,
            const Point3D& origin, const Point3D& voxel_size)

  int get_x_size() const
  int get_y_size() const
  int get_min_x() const
  int get_min_y() const
  int get_max_x() const
  int get_max_y() const

  Point3D get_origin()  const
  Point3D get_voxel_size()  const

};
```

VII.2.4. Constructors

```
PETPlane(const Tensor2D<float> &p, const int p_num, const Point3D& origin, const Point3D&
voxel_size)
```

This constructor takes a Tensor2D<float> object, the plane number, origin (a displacement vector in physical units with respect to the origin of the coordinate system) and voxel_size information.

VII.2.5. Data access members

VII.2.5.1. int get_x_size() const, int get_y_size() const

Three methods which return the number of voxels in the volume.

VII.2.5.2. int get_min_x() const, int get_min_y() const, const, int get_max_x() const, int get_max_y() const

These methods return the ranges of the indices to access the data following the conventions in the Tensor classes: *min* and *max* give the first and the last index in the range.

VII.2.5.3. Point3D get_origin() const, Point3D get_voxel_size() const

Two methods returning the origin and voxel size.

VIII. CLASSES FOR PROJECTION DATA

VIII.1. DESCRIPTION

The projection data for 3D PET form a fairly complicated structure, thus needing a number of classes to be able to represent the data in C++. Confusingly, the data set for a 3D PET scan is four dimensional. Here are the four coordinates used in the classes below:

segment : (this is CTI terminology, CTI PET Systems, Knoxville, TE) before axial compression, this is the ring difference between the rings detecting the LOR, hence it is related to the angle between the LOR and z-axis.

view : angle of LOR projected on a plane perpendicular to the z-axis, runs anticlockwise looking along the z-axis

ring : a LOR is between detectors on two rings. **ring** is the number of the ring at smallest **z.** Unfortunately, when axial compression is used, the number of 'virtual' rings is (2*num_scanner_rings - 1), see below.

bin : for a given segment, angle and z, there are a number of (approximately) parallel and coplanar LORs. The bins are assumed to lie on an arc of a circle, unless arc correction is performed.

The data can be stored in different ways on disk, depending on the order of the coordinates (see PETSinogramOfVolume::StorageOrder below and the implementation issues of this subsection). This gives us a hierarchy of data structures as presented in the Figure IV.1. Note that this is not a hierarchy of class derivation, but of containment. We use here the abuse of terminology to call a ***PETSegment*** the 3D data structure, and not the coordinate. A ***PETSegment*** comes in two flavours (derived classes), depending on the order of the coordinates. See below for details.

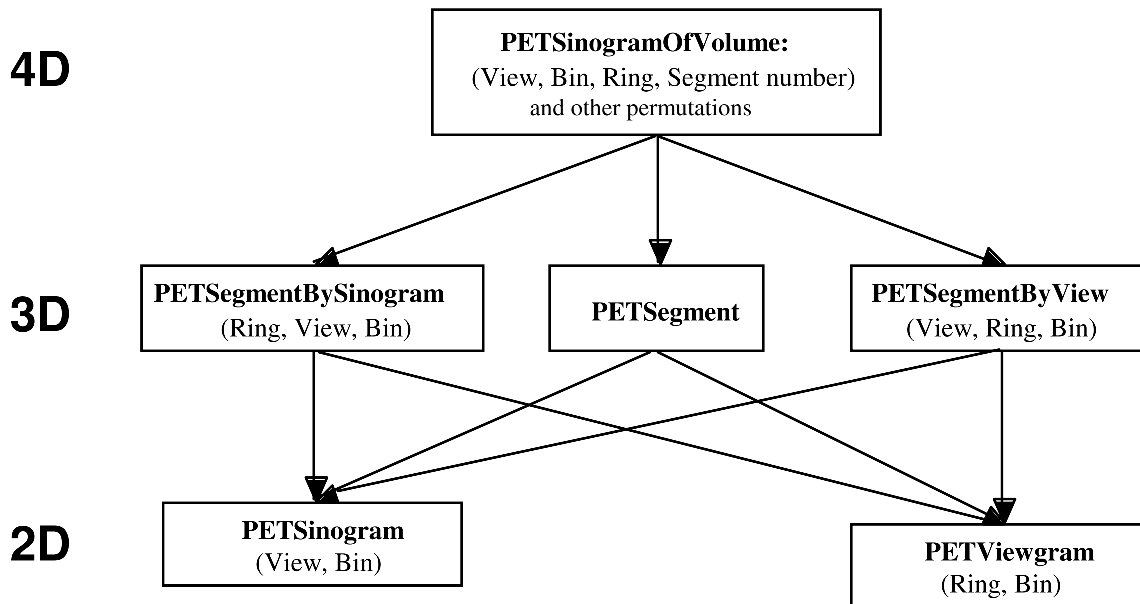


Figure VIII.1: *Hierarchical view of the projection data. Note that at the top, three combinations of storage are supported (**SegmentRingViewBin**, **SegmentViewRingBin** or **ViewSegmentRingBin**. The latter order is especially used in the case of GE Advance sinogram data.*

The whole 3D dataset has always an ODD number of segments. The segment identification numbers for a sinogram with $2N+1$ segments are: 0, –1, +1, –2, +2 ...., $-N$, $+N$

VIII.1.1. Axial compression

To reduce data size, many scanners (including the ECAT 966, ECAT ART and GE Advance scanner) normally do not store a 2D sinogram for every ring and ring difference, but data of close ring differences are added together (Figure IV.2). To keep as high an axial resolution as possible, some of these 2D sinograms are assigned to positions halfway between two rings. This is an extension of the standard 2D PET practice of adding sinograms of ring difference +1 and –1 to give a sinogram for the "cross-plane". Currently, our classes keep a *ring_num* which runs over these "virtual rings". Furthermore, to every segment there are two parameters *min_ring_difference* and *max_ring_difference* associated which give the range of (scanner) ring differences added to make up that segment. To illustrate axial compression, a michelogram of ART data for span of 7 and a ring difference of 17 (16 rings) is shown in Annex 3 of D4.1a.

VIII.1.2. Location

- *include/sinodata.h*

VIII.1.3. Implementation issues

A well-known fact in C or C++ programming is worth repeating here: care must be taken when choosing the order of indices in multidimensional arrays – the fastest varying index last, the slowest first. Due to the large number of memory accesses this has a considerable effect: in a previous implementation [*Egger, 1998*], it was found that using sinogram arrays indexed by [*Ring*][*View*][*Bin*] and an image array indexed by [*plane*][*y*][*x*] results in the backprojection executing more than 15% faster than when these indices are used back to front (*View* is the azimuthal angle of the

projection (*angle*), bin the radial co-ordinate on the projection plane (bin element), and *x* and *y* transaxial image co-ordinates). This implementation uses viewgrams as input for the projectors as ring and bin are the coordinates in the inner loops. Data is stored as float.

We now list the classes in "down-to-top" order.

VIII.2. CLASS PETSINOGRAM

VIII.2.1. Description

The class to represent the 2D dataset one gets when fixing the *segment_num* and *ring_num* coordinates of the whole projection set. Similar to ***PETViewgram***.

VIII.2.2. Location

- *include/sinodata.h*

VIII.2.3. Class

```
Class PETSinogram: public Tensor2D <float>
{
private:
  int segment_num;
  int ring_num;
  int min_ring_difference;
  int max_ring_difference;

public:
 const PETScanInfo* scanner;             // scanner points to all design parameters of the scanner
  Real scale_factor; // scale factor ignored for the moment
  PETSinogram(const Tensor2D<float>& data, const PETScanInfo &scan_info,
               const int ring_num, const int segment_num,
               const int min_ring_diff, const int max_ring_diff)

  PETSinogram(const Tensor2D<float>& data, const PETScanInfo &scan_info,
               const int ring_num, const int segment_num)

  const PETScannerInfo* scanner;         // Pointer to scanner information

  int get_segment_num() const            // Returns the segment number to which the sinogram
        belongs
  int get_min_ring_difference() const    // Returns the minimum ring difference
  int get_max_ring_difference() const    // Returns the maximum ring difference
  float get_average_ring_difference() const     // Returns the average ring difference

  int get_ring_num() const               // Returns the ring number to which the sinogram belongs
  int get_num_views() const              // Returns the number of views
  int get_num_bins() const               // Returns the number of bin elements
  int get_min_view() const               // Returns the first index along view axis
  int get_max_view() const               // Returns the last index along view axis
  int get_min_bin() const                // Returns the first index along bin axis
  int get_max_bin() const                // Returns the last index along bin axis
};
```

VIII.2.4. Constructors

VIII.2.4.1. General constructor

```
PETSinogram(const Tensor2D<float>& data, const PETScanInfo &scan_info,
             const int ring_num, const int segment_num,
             const int min_ring_diff, const int max_ring_diff)
```

The *data* argument contains the ‘real’ data, to be accessed as *data[view_num][bin_num].* The *ring_num* and *segment_num* arguments say what the ‘missing’ coordinates are in the 3D PET dataset. The last two arguments are discussed in the note on axial compression above.

VIII.2.4.2. Constructor without ring differences

```
PETSinogram(const Tensor2D<float>& data, const PETScanInfo &scan_info,
           const int ring_num, const int segment_num)
```

This constructor does not take ring_difference arguments, assuming that there is no axial compression. In effect, this means that the general constructor is called with max_ring_difference = max_ring_difference = segment_num.

VIII.2.4.3. Copy constructor PETSinogram(const PETSinogram &s)

This constructor creates a new ***PETSinogram*** copied from the one passed as an argument.

VIII.2.5. Public members

VIII.2.5.1. const PETScanInfo* scan_info

Points to a data structure giving information on the scanner.

VIII.2.6. Data access methods

VIII.2.6.1. int get_segment_num() const, int get_min_ring_difference() const, int get_max_ring_difference() const, float get_average_ring_difference() const

These methods get information on the segment to which the 2D sinogram belongs. *get_average_ring_difference)* returns the average (scanner) ring difference for the segment.

VIII.2.6.2. int get_ring_num() const

This methods returns the number of the ‘virtual’ ring to which the 2D sinogram belongs.

VIII.2.6.3. int get_num_views() const, int get_num_bins() const

These two methods return the dimensions of the 2D sinogram.

VIII.2.6.4. int get_min_view() const, int get_max_view() const, int get_min_bin() const, int get_max_bin() const

The four methods provide the range of the indices. These ranges follow the conventions of the *Tensor* classes: *min* and *max* give the first and the last index in the range.

VIII.2.7. Derived methods

As this is a derived class from Tensor2D<float>, all its methods can be used. The most important one of course are the *operator[int i]()* methods. As an example:

```
PETSinogram sino = segment.get_sinogram(2);
// set the element at view_num 2,  bin_num 1 to a new value
sino[3][1] = 1.F;
```

VIII.3. CLASS PETVIEWGRAM

VIII.3.1. Description

The class to represent the 2D dataset one gets when fixing the *segment_num* and *view_num* coordinates of the whole projection set. Similar to **PETSinogram**.

VIII.3.2. Location

This class can be found in *include/sinodata.h*

VIII.3.3. Class

```
Class PETSViewgram: public Tensor2D <float>
{
private:
  int segment_num;
  int view_num;
  int min_ring_difference;
  int max_ring_difference;

public:
  const PETScanInfo scan_info;
  Real scale_factor;

  PETViewgram(const Tensor2D<float>& data, const PETScanInfo &scan_info,
              const int view_num, const int segment_num,
              const int min_ring_diff, const int max_ring_diff)

  PETViewgram(const Tensor2D<float>& data, const PETScanInfo &scan_info,
              const int view_num, const int segment_num)

  const PETScanInfo* scan_info;          // Pointer to scanner information

  int get_segment_num() const            // Returns the segment number to which the sinogram
        belongs
  int get_min_ring_difference() const    // Returns the minimum ring difference
  int get_max_ring_difference() const    // Returns the maximum ring difference
  float get_average_ring_difference() const     // Returns the average ring difference

  int get_view_num() const               // Returns the view number to which the sinogram belongs
  int get_num_rings() const              // Returns the number of rings
  int get_num_bins() const               // Returns the number of bin elements
  int get_min_ring() const               // Returns the first index along ring axis
  int get_max_ring() const               // Returns the last index along ring axis
  int get_min_bin() const                // Returns the first index along bin axis
  int get_max_bin() const                // Returns the last index along bin axis
};
```

VIII.3.4. Constructors

VIII.3.4.1. General constructor

```
PETViewgram(const Tensor2D<float>& data, const PETScanInfo &scan_info,
             const int view_num, const int segment_num,

             const int min_ring_diff, const int max_ring_diff)
```

The *data* argument contains the ‘real’ data, to be accessed as *data[view_num][bin_num]*. The *view_num* and *segment_num* arguments say what the ‘missing’ coordinates are in the 3D PET dataset. The last two arguments are discussed in the note on axial compression above.

VIII.3.4.2. Constructor without ring differences

```
PETViewgram(const Tensor2D<float>& data, const PETScanInfo &scan_info,
            const int view_num, const int segment_num)
```

This constructor does not take ring_difference arguments, assuming that there is no axial compression. In effect, this means that the general constructor is called with max_ring_difference = max_ring_difference = segment_num.

VIII.3.4.3. Copy constructor PETViewgram(const PETViewgram &v)

This constructor creates a new **PETViewgram** copied from the one passed as an argument.

VIII.3.5. Public members

VIII.3.5.1. const PETScanInfo* scan_info

Points to a data structure giving information on the scanner.

VIII.3.6. Data access methods

VIII.3.6.1. int get_segment_num() const, int get_min_ring_difference() const, int get_max_ring_difference() const, float get_average_ring_difference() const

These methods get information on the segment to which the 2D viewgram belongs. *get_average_delta()* returns the average (scanner) ring difference for the segment.

VIII.3.6.2. int get_view_num() const

This methods returns the number of the 'virtual' ring to which the 2D viewgram belongs.

VIII.3.6.3. int get_num_views() const, int get_num_bins() const

These two methods return the dimensions of the 2D viewgram.

VIII.3.6.4. int get_min_view() const, int get_max_view() const, int get_min_bin() const, int get_max_bin() const

The four methods provide the range of the indices. These ranges follow the conventions of the *Tensor* classes: *min* and *max* give the first and the last index in the range.

VIII.3.7. Derived methods

As this is a derived class from Tensor2D<float>, all its methods can be used. The most important one of course are the *operator[int i]()* methods. As an example:

```
PETViewgram view = segment.get_viewgram(2);
// set the element at ring_num 2,  bin_num 1 to a new value
view[3][1] = 1.F;
```

VIII.4. CLASS PETSEGMENT

VIII.4.1. Description

A base class to represent the 3D datasets one gets when fixing the *segment_num co*ordinates of the whole projection set. This class serves as a base for **PETSegmentBySinogram** and **PETSegmentByViewgram** as documented below. As this class contains pure virtual functions, it is an 'abstract' class, *i.e.* no objects of class **PETSegment** can be constructed. One can pass references or pointers to a **PETSegment** though. In that case, the virtual function mechanism will make sure that appropriate versions of the members will be called.

VIII.4.2. Location

- *include/sinodata.h*

VIII.4.3. Class

```
Class PETSegment{
public:
enum StorageOrder{ StorageByView, StorageBySino };

protected:
  int segment_num;
  int min_ring_difference;
  int max_ring_difference;

public:
  const PETScanInfo scan_info;

  virtual StorageOrder get_storage_order() const = 0;

  PETSegment(  const PETScanInfo& sc_info, const int s_num,
               const int min_rd, const int max_rd) :
               segment_num(s_num),
               min_ring_difference(min_rd),
               max_ring_difference(max_rd),
               scan_info(sc_info)

  int get_segment_num() const
  int get_min_ring_difference() const
  int get_max_ring_difference() const
  float get_average_ring_difference() const

  virtual int get_min_ring() const = 0;
  virtual int get_max_ring() const = 0;
  virtual int get_min_view() const = 0;
  virtual int get_max_view() const = 0;
  virtual int get_min_bin() const = 0;
  virtual int get_max_bin() const = 0;
  virtual get_get_num_rings() const = 0;
  virtual get_get_num_views() const = 0;
  virtual get_get_num_bins() const = 0;

  virtual PETSinogram get_sinogram(int ring_num) const = 0;
  virtual PETViewgram get_viewgram(int view_num) const = 0;
  virtual void set_sinogram(const PETSinogram &s) = 0;
  virtual void set_viewgram(const PETViewgram &v) = 0;
  virtual void set_sinogram(const PETSinogram &s, int ring_num) = 0;
};
```

VIII.4.4. Constructors

VIII.4.4.1. General constructor

```
PETSegment(   const PETScanInfo& sc_info, const int s_num,
              const int min_rd, const int max_rd) :
              segment_num(s_num),
              min_ring_difference(min_rd),
              max_ring_difference(max_rd),
              scan_info(sc_info)
```

This constructor allows to pass either PETSegmentBySinogram or PETSegmentByView data and only the s_num argument tells you which sequence number of 3D PET dataset to process. The last two arguments are discussed in the note on axial compression above.

VIII.4.5. Public types

VIII.4.5.1. StorageOrder

This is a type used to distinguish between the two kinds of **PETSegment**. However, because most of the functions are virtual, it should be seldom necessary to use this type.

VIII.4.6. Public members

VIII.4.6.1. const PETScarInfo &scan_info

Points to a data structure giving information on the scanner.

VIII.4.7. Protected members

VIII.4.7.1. int segment_num, int min_ring_difference, int max_ring_difference

These members store the information on which segment this is.

VIII.4.8. Data access methods

VIII.4.8.1. StorageOrder get_storage_order() const

This methods returns an object of type **StorageOrder** and to tell which type the object is. This could be replaced by C++ run time type information, but this is a very recent addition to the ANSI C++ standard, and not many compilers support it (*gcc* does support RTTI).

VIII.4.8.2. int get_segment_num() const, int get_min_ring_difference() const, int get_max_ring_difference() const, float get_average_ring_difference() const

These methods get information on the segment. *get_average_ring_difference)* returns the average (scanner) ring difference for the segment.

VIII.4.8.3. int get_num_rings() const

This methods returns the number of the 'virtual' rings in the segment.

VIII.4.8.4. int get_num_views() const, int get_num_bins() const

These two methods return the other dimensions of the segment.

VIII.4.8.5. int get_min_ring() const, int get_max_ring() const, int get_min_view() const, int get_max_view() const, int get_min_bin() const, int get_max_bin() const

These methods provide the range of the indices. These ranges follow the conventions of the *Tensor* classes: *min* and *max* give the first and the last index in the range.

VIII.4.8.6. PETSinogram get_sinogram(int ring_num), PETViewgram get_viewgram(int view_num)

These methods extract part of the data and return objects of the appropriate class.

VIII.4.8.7. void set_sinogram(const PETSinogram &s), void set_viewgram(const PETViewgram &v)

These methods have to be used to change the data in a PETSegment.

VIII.4.8.8. void set_sinogram(const PETSinogram &s, int ring_num)

This method assigns the PETSinogram to a specific ring number

VIII.5. CLASS PETSEGMENTBYVIEW

VIII.5.1. Description

This class is derived from **PETSegment**. Most of the methods are just implementations of those discussed in the section on **PETSegment**. We document only the new methods below. See also the class **PETSegmentBySinogram**.

The class is also derived from Tensor3D<*float*>. Although this is currently a public derivation, this should not be used in any code. The reason is that a Tensor3D<*float*> object has to fit in memory, and one segment can be fairly large (maximum 16 MB for segment 0 of the Ecat 966 scanner when not using axial compression).

VIII.5.2. Location

- *include/sinodata.h*

VIII.5.3. Class

```
Class PETSegmentByView : public PETSegment, public Tensor3D <float>{
public:
  PETSegmentByView(const Tensor3D<float>& data, const PETScanInfo &scan_info,
             const int segment_num,
             const int min_ring_difference, const int max_ring_difference)

  PETSegmentByView(const Tensor3D<float>& data, const PETScanInfo &scan_info,
                 const int segment_num)

  PETSegmentByView(const PETSegmentBySinogram& );

  StorageOrder get_storage_order() const

  int get_num_rings() const
  int get_num_views() const
  int get_num_bins() const


  int get_min_ring() const
  int get_max_ring() const
  int get_min_view() const
  int get_max_view() const
  int get_min_bin() const
  int get_max_bin() const

  PETSinogram get_sinogram(int ring_num) const
  PETViewgram get_viewgram(int view_num) const;

  void set_sinogram(const PETSinogram &s);
  void set_viewgram(const PETViewgram &v);
  void set_sinogram(PETSinogram const &s, int ring_num);
};
```

VIII.5.4. Constructors

VIII.5.4.1. General constructor

```
PETSegmentByView(const Tensor3D<float>& data, const PETScanInfo &scan_info,
             const int segment_num,
             const int min_ring_difference, const int max_ring_difference)
```

The *data* argument contains the 'real' data, stored as *data[view_num][ring_num][bin_num]*. The *segment_num* arguments say what the 'missing' coordinates are in the 3D PET dataset. The last two arguments are discussed in the note on axial compression above.

VIII.5.4.2. Constructor without ring differences

```
PETSegmentByView(const Tensor3D<float>& data, const PETScanInfo &scan_info,
                 const int segment_num)
```

This constructor does not take ring_difference arguments, assuming that there is no axial compression. In effect, this means that the general constructor is called with max_ring_difference = max_ring_difference = segment_num.

VIII.5.4.3. Copy constructor PETSegmentByView(const PETSegmentByView &)

This constructor creates a new PETSegmentByView copied from the one passed as an argument.

VIII.5.4.4. Conversion constructor PETSegmentByView(const PETSegmentBySinogram&)

This constructor creates a PETSegmentByView copied from a PETSegmentBySinogram. This means that data are stored in a different order.

VIII.5.4.5. PETViewgram get_sinogram(int ring_num), PETViewgram get_viewgram(int view_num)

These methods extract part of the data and return objects of the appropriate class.

VIII.5.4.6. void set_sinogram(const PETSinogram &s), void set_viewgram(const PETViewgram &v)

These methods have to be used to change the data in a PETSegment.

VIII.5.4.7. void set_sinogram(const PETSinogram &s, int ring_num)

This method assign the PETSinogram to specific ring number

VIII.5.4.8. int get_num_rings() const

This methods returns the number of the 'virtual' rings in the segment.

VIII.5.4.9. int get_num_views() const, int get_num_bins() const

These two methods return the other dimensions of the segment.

VIII.5.4.10. int get_min_ring() const, int get_max_ring() const, int get_min_view() const, int get_max_view() const, int get_min_bin() const, int get_max_bin() const

These methods provide the range of the indices. These ranges follow the conventions of the *Tensor* classes: *min* and *max* give the first and the last index in the range.

VIII.6. CLASS PETSEGMENTBYSINOGRAM

VIII.6.1. Description

This class is derived from **PETSegment**. Most of the methods are just implementations of those discussed in the section on **PETSegment**. We document only the new methods below. Se also the class **PETSegmentByView**.

The class is also derived from **Tensor3D**<*float*>. Although this is currently a public derivation, this should not be used in any code. The reason is that a **Tensor3D**<*float*> object has to fit in memory, and one segment can be fairly large (maximum 16 MB for segment 0 of the Ecat 966 scanner when not using axial compression).

VIII.6.2. Location

- *include/sinodata.h*

VIII.6.3. Class

```
Class PETSegmentBySinogram: public PETSegment, public Tensor3D <float>{
{
public:
  PETSegmentBySinogram(const Tensor3D<float>& data, const PETScanInfo &scan_info,
             const int segment_num,
             const int min_ring_difference, const int max_ring_difference)

  PETSegmentBySinogram(const Tensor3D<float>& data, const PETScanInfo &scan_info,
                 const int segment_num);

  PETSegmentBySinogram(const Tensor3D<float>& data, const PETScanInfo &scan_info,
                 const int segment_num, const int min_rd, const int max_rd);

  PETSegmentBySinogram (const PETSegmentByView& );

  StorageOrder get_storage_order() const
  int get_num_rings() const
  int get_num_views() const
  int get_num_bins() const

  int get_min_ring() const
  int get_max_ring() const
  int get_min_view() const
  int get_max_view() const
  int get_min_bin() const
  int get_max_bin() const

  PETSinogram get_sinogram(int ring_num) const
  PETViewgram get_viewgram(int view_num) const;

  void set_sinogram(PETSinogram const &s, int ring_num)
  void set_sinogram(const PETSinogram &s);
  void set_viewgram(const PETViewgram &v);
};
```

VIII.6.4. Constructors

VIII.6.4.1. General constructor

```
PETSegmentBySinogram(const Tensor3D<float>& data, const PETScanInfo &scan_info,
             const int segment_num,
             const int min_ring_difference, const int max_ring_difference)
```

The *data* argument contains the ‘real’ data, stored as *data[view_num][ring_num][bin_num]*. The *segment_num* arguments say what the ‘missing’ coordinates are in the 3D PET dataset. The last two arguments are discussed in the note on axial compression above.

VIII.6.4.2. Constructor without ring differences

```
PETSegmentBySinogram(const Tensor3D<float>& data, const PETScanInfo &scan_info,
                 const int segment_num)
```

This constructor does not take ring_difference arguments, assuming that there is no axial compression. In effect, this means that the general constructor is called with max_ring_difference = max_ring_difference = segment_num.

VIII.6.4.3. Constructor with ring differences

```
PETSegmentBySinogram(const Tensor3D<float>& data, const PETScanInfo &scan_info,
```

```
                    const int segment_num, const int min_rd, const int max_rd)
```

This constructor does not take ring_difference arguments, assuming that there is no axial compression. In effect, this means that the general constructor is called with max_ring_difference = max_ring_difference = segment_num.

VIII.6.4.4. Copy constructor PETSegmentBySinogram(const PETSegmentBySinogram &s)

This constructor creates a new PETSegmentBySinogram copied from the one passed as an argument.

VIII.6.4.5. Conversion constructor PETSegmentBySinogram(const PETSegmentByView &v)

This constructor creates a PETSegmentBySinogram copied from a PETSegmentByView. This means that data are stored in a different order.

VIII.6.4.6. PETSinogram get_sinogram(int ring_num), PETViewgram get_viewgram(int view_num)

These methods extract part of the data and return objects of the appropriate class.

VIII.6.4.7. void set_sinogram(const PETSinogram &s), void set_viewgram(const PETViewgram &v)

These methods have to be used to change the data in a PETSegment.

VIII.6.4.8. void set_sinogram(const PETSinogram &s, int ring_num)

This method assign the PETSinogram to specific ring number

VIII.6.4.9. int get_num_rings() const

This methods returns the number of the 'virtual' rings in the segment.

VIII.6.4.10. int get_num_views() const, int get_num_bins() const

These two methods return the other dimensions of the segment.

VIII.6.4.11. int get_min_ring() const, int get_max_ring() const, int get_min_view() const, int get_max_view() const, int get_min_bin() const, int get_max_bin() const

These methods provide the range of the indices. These ranges follow the conventions of the *Tensor* classes: *min* and *max* give the first and the last index in the range.

VIII.7. CLASS PETSINOGRAMOFVOLUME

VIII.7.1. Description

The **PETSinogramOfVolume** class represents PET data BEFORE reconstruction and is always organized into an ODD number of segments, where each segment contains a specific number of sinograms or viewgrams.

The segment identification numbers for a sinogram with N segments are: 0, –1, +1, –2, +2 ...., $-N/2$, $+N/2$

Because of the size of the whole set of projections (for the Ecat 966 scanner about 800 MB if no compression is used and data stored as float), we decided not to keep the whole data set in memory.

VIII.7.2. Location

- *include/sinodata.h*

- *buildblock/sinodata.cxx*

- *buildblock/viewdata.cxx*

VIII.7.3. Class

```
Class PETSinogramOfVolume
{
```

```
public:
  enum StorageOrder {
    SegmentRingViewBin, SegmentViewRingBin,
    ViewSegmentRingBin, // GE Advance format
    Unsupported };

private:
  iostream& sino_stream;
  long  offset;

  int min_segment;
  int max_segment;
  int min_view;
  int max_view;
  int min_bin;
  int max_bin;

  vector<int> segment_sequence;
  vector< int> min_ring_difference;
  vector< int> max_ring_difference;
  vector<int> min_rings;
  vector<int> max_rings;
  StorageOrder storage_order;

  int find_segment_index_in_sequence(const int segment_num) const;

  NumericType on_disk_data_type;
  Real scale_factor;

public:
  StorageOrder get_storage_order() const
  PETScanInfo scan_info;

PETSinogramOfVolume(const PETScanInfo &scr_info,
                 const vector<int>& segment_seq,
                 const vector<int>& min_r,
                 const vector<int>& max_r,
                 const int min_v, const int max_v,
                 const int min_b, const int max_b,
                 iostream& s, const long offs,
                 StorageOrder o,
                 NumericType data_type,
                 Real scale_factor = 1)

  PETSinogramOfVolume(const PETScannInfo& scr_info,
                 const vector<int>& segment_seq,
                 const vector<int>& min_ring_diff,
                 const vector <int>& max_ring_diff,
                 const vector<int>& min_r,
                 const vector<int>& max_r,
                 const int min_v, const int max_v,
                 const int min_b, const int max_b,
                 iostream& s, const long offs,
                 StorageOrder o,
                 NumericType data_type,
                 Real scale_factor = 1)

  PETSinogramOfVolume(PETScannerInfo& scanner, int span, int max_delta,
                 iostream& s, long offset_in_file,
```

```
                 PETSinogramOfVolume::StorageOrder storage_order,
                 NumericType data_type,
                 Real scale_factor = 1);

  int get_num_segments() const;
  int get_num_views() const;
  int get_num_bins() const;
  int get_min_bin() const
  int get_max_bin() const
  int get_min_view() const
  int get_max_view() const
  int get_min_segment() const
  int get_max_segment() const

  int get_min_ring(int segment_num) const
  int get_max_ring(int segment_num) const
  int get_min_ring_difference(int segment_num) const
  int get_max_ring_difference(int segment_num) const
  float get_average_ring_difference(int segment_num) const
  float get_max_average_ring_difference() const
  float get_min_average_ring_difference() const

  void show_max_rings() const
  void show_min_rings() const

  PETSegmentBySinogram get_segment_sino_copy(const int segment_num) const;
  PETSegmentByView get_segment_view_copy(const int segment_num) const;

  PETViewgram get_viewgram_copy(const int view, const int segment_num) const;

  PETSegmentBySinogram empty_segment_sino_copy(const int segment_num,
                        const bool make_num_bins_odd = true) const;

  PETSegmentByView empty_segment_view_copy(const int segment_num,
                                         const bool make_num_bins_odd = true)
const;

  PETViewgram empty_viewgram_copy(const int view, const int segment_num,
                                  const bool make_num_bins_odd = true) const;
};
```

VIII.7.4. Public types

StorageOrder is a type that gives the (supported) ways in which the four coordinates in the dataset are ordered on disk. For example, for ***SegmentRingViewBin***, segments are stored one after the other, while each segment contains a sequence of sinograms.

VIII.7.5. Constructors

VIII.7.5.1. General constructor

```
PETSinogramOfVolume(const PETScanInfo& scan_info,
                 const vector<int>& segment_sequence,
                 const vector<int>& min_ring_diff,
                 const vector <int>& max_ring_diff,
                 const vector<int>& min_ring,
                 const vector<int>& max_ring,
                 const int min_view_num, const int max_view_num,
                 const int min_bin, const int max_bin,
```

```
                iostream&, const long offset,
                StorageOrder,
                NumericType data_type,
                Real scale_factor = 1)
```

- This constructor takes a lot of arguments because the data structure is fairly complicated. The *segment_sequence* parameter is a vector saying in which order the segments occur in the file. The other arguments of type *vector<int>* contain information on the segments in the same order, i.e. *min_ring_diff[3]* should contain the minimum (scanner) ring difference which make up the segment with segment number *segment_sequence[3]*. Note that the *min_ring, max_ring* arguments have to be vectors, as in general these parameters vary per segment.

- The *iostream&* argument points a stream with the (binary) data. The *offset* argument specifies where the data starts in the stream. This allows for more than one projection dataset to be stored in one file. However, one dataset has to be stored contiguously (i.e. no headers between e.g. 2D sinograms).

- The *data_type* argument specifies in what format the data is written in the *iostream*. However, once in memory, the data is stored as floats.

Finally, the *scale_factor* can be used to specify a calibration factor for instance. Again, we do not allow for different scale factors per 2D sinogram.

VIII.7.5.2. Constructor assuming no axial compression

```
 PETSinogramOfVolume(const PETScannerInfo& scanner,
                const vector<int>& segment_sequence,
                const vector<int>& min_ring,
                const vector<int>& max_ring,
                const int min_view_num, const int max_view_num,
                const int min_bin, const int max_bin,
                iostream& s, const long offset,
                StorageOrder,
                NumericType data_type,
                Real scale_factor = 1)
```

This is essentially the same constructor as before, but without the *min_ring_diff*, *max_ring_diff* arguments, as they are not necessary of if no axial compression is used.

VIII.7.5.3. A shorthand constructor which uses knowledge about the scanner

```
PETSinogramOfVolume(PETScannerInfo& scanner, int span, int max_delta,
                iostream& s, long offset_in_file,
                PETSinogramOfVolume::StorageOrder storage_order,
                NumericType data_type,
                Real scale_factor = 1);
```

Different scanner manufacturers have their typical ways of storing data. This constructor infers from the *scanner* argument what most of the parameters are for the general constructor above. In particular, for CTI-Siemens scanners, the *span* argument specifies the axial compression (via a so-called 'michelogram' illustrated in Annex 3 of D4.1a). The GE Advance scanner stores its data always with the same axial compression, so the *span* argument of this constructor is ignored for that scanner. Finally, *max_delta* is the maximum stored ring difference in the data set (this parameter is set at acquisition time).

VIII.7.6. Public members

VIII.7.6.1. PETScanInfo scan_info

A data structure giving information on the scanner.

VIII.7.7. Data access methods

VIII.7.7.1. StorageOrder get_storage_order() const

This method returns the storage order of the data on disk.

VIII.7.7.2. int find_segment_index_in_sequence(const int segment_num) const

This internal function aims to make finding a segment easier.

VIII.7.7.3. int get_num_segments() const, int get_num_views() const, int get_num_bins() const

A few methods giving information on the size of the data.

VIII.7.7.4. int get_min_segment() const, int get_max_segment() const, int get_min_view() const, int get_max_view() const, int get_min_bin() const, int get_max_bin() const

These methods provide the range of the indices. These ranges follow the conventions of the *Tensor* classes: *min* and *max* give the first and the last index in the range.

VIII.7.7.5. int get_min_ring(int segment_num) const, int get_max_ring(int segment_num) const, int get_min_ring_difference(int segment_num) const, int get_max_ring_difference(int segment_num) const

These methods provide the range of the indices for a specific segment, respectively for the ring, the ring difference

VIII.7.7.6. float get_average_ring_difference(int segment_num) const

This method returns the average ring difference for a specific segment

VIII.7.7.7. float get_max_average_ring_difference() const

This method returns the maximum ring difference of the whole volume of sinograms

VIII.7.7.8. float get_min_average_ring_difference() const

This method returns the minium ring difference of the whole volume of sinograms

VIII.7.7.9. PETSegmentBySinogram get_segment_by_sino_copy(const int segment_num) const

A method which returns a copy of a particular segment as a PETSegmentBySinogram.

VIII.7.7.10. PETSegmentByView get_segment_by_view_copy(const int segment_num) const;

A method which returns a copy of a particular segment as a PETSegmentByView.

VIII.7.7.11. PETSegmentBySinogram empty_segment_sino_copy(const int segment_num, const bool make_num_bins_odd = true) const

This method creates an empty PETSegmentBySinogram with appropriate sizes, scan_info etc. for this segment_num. If make_num_bins_odd==false, the result is the same as calling get_segment_by_sino_copy(segment_num), followed with a fill(0). If make_num_bins_odd==true and get_num_bins() is an even number, the number of bins is increased with 1.

VIII.7.7.12. PETSegmentByView empty_segment_view_copy(const int segment_num, const bool make_num_bins_odd = true) const

This method creates an empty PETSegmentByView with appropriate sizes etc. See empty_segment_sino_copy() for more details.

VIII.7.7.13. PETViewgram empty_viewgram_copy(const int view, const int segment_num, const bool make_num_bins_odd = true) const

This method creates an empty PETViewgram with appropriate sizes etc. See empty_segment_sino_copy() for more details.

VIII.7.8. Implementation details

Because of the size of the whole set of projections (for the Ecat 966 scanner about 800MB if the data is stored as float and no compression is used), it is normally unfeasible to store the whole projection data in memory. Therefore we decided not to provide direct access to the data, but only via the get_segment_by_sino(),get_segment_by_view(), get_viewgram() methods. As the data are accessed only via the iostream object passed to the constructor, one can use an object of type fstream (for data on disk) or stringstream (for data in memory).

IX. TRUNCATING, TRIMMING, OFFSETTING, MASHING AND ZOOMING FUNCTIONS

IX.1. DESCRIPTION:

The zooming function includes both truncation, trimming, offseting, and zooming in and out. The impact of these four possibilties depends both on the initial and final 2D data array sizes, as well as on the initial and final voxel sizes determined by the zoom factor.

- Truncation :

Data truncation makes the matrix size smaller without changing the pixel size

initial_pixel_size = final_pixel_size and

initial_dimension > final_dimension

- Trimming (padding with zeroes) :

Padding data makes bigger matrix without changing the pixel size

initial_pixel_size = final_pixel_size and

initial_dimension < initial_dimension

- Zoom in :

Magnification 2D data matrix makes the pixel size smaller and occurs when zoom > 1

initial_pixel_size > final_pixel_size whatever the dimensions of the 2D data array

- Zoom out :

Shrinking 2D data matrix makes the pixel size bigger and occurs when zoom < 1

initial_pixel_size < final_pixel_size whatever the dimensions of the 2D data array

- Offsetting :

if positive,

- the X offset shifts image left
- the Y offset shifts image down

if negative,

- the X offset shifts image right
- the Y offset shifts image up

Of course, these zooming utilities can be applied on both sinograms and images data.

So, to determine which case has to be applied during zooming, the user has to define three parameters :

- the zoom factor (magnification effect) (case of zoom > 1)
- the new image size,
- the offsets in X, Y (where the negative value have the effect to shift image up for Y offset, and left for X offset).

A few consistency conditions have to be applied on projection data. Before zoom:

original _bin_size*original_num_bins == FOV_radius (Eq. IX.1)

When zoom >1 , it is not useful to have :

new_bin_size*new_num_bins > FOV_radius, (Eq. IX.2)

On projection data, choosing a zoom factor means to rebin sinogram after filtering so that :

new_pixel_size == initial_pixel_size/zoom == initial pixel size (Eq. IX.3)

Arbitrarily, zoom=1 has been chosen to correspond to image pixel size equal to the physical sampling size in the middle of the scanner (i.e the bin size). Then, a zoom of 2 corresponds to image pixel size half of the bin size and so forth.
The number of image pixel should not exceed the dimension of the scanner so that:

bin size * number of image pixels / zoom <= FOV diameter = binsize * num_bins (Eq. IX.4)

where num_bins is the number of measured sinogram bins.

IX.2. LOCATION:

- *recon_buildblock/zoom.cxx*
- *include/recon_buildblock/zoom.h*

IX.3. ON PROJECTION DATA

IX.3.1. Along X axis direction (Trimming, zooming in/out, offset)

On projection data, the `zoom_segment` acts only on the bin direction.

```
void zoom_segment (
      PETSegmentBySinogram &segment,      // (in/out) Segment
      const float zoom,                   // (in) zoom factor value (magnification effect)
      const float Xoff,                   // (in) displacement along the x-axis of the image (X
                                            offset in cm)
      const float Yoff,                   // (in) displacement along the x-axis of the image (Y
                                            offset in cm)
      const int size,                     // (in) Final bin size
      const float itophi);                // (in) Value of itophi since this value changes
                                            according to the type of algorithm.
                                            In PROMIS, itophi= π / num_views;
                                            In FORE, itophi = 2* π / num_views_pow2 with
                                            num_views_pow2 is the power of 2 of nviews

void zoom_segment (   PETSegmentByView& segment, const float zoom,
                      const float Xoff, const float Yoff, const int size, const float itophi);
```

This method implement the zooming from PETSegmentByView projections data along the bin elements direction.

```
void zoom_segment (   PETSegmentBySinogram& segment, const float zoom,
                      const float Xoff, const float Yoff, const int size, const float itophi);
```

This method implement the zooming from PETSegmentBySinogram projections data along the bin elements direction..

```
void zoom_viewgram (   PETViewgram& in_view, const float zoom,
                       const float Xoff, const float Yoff, const int size, const float itophi);
```

This method implement the zooming from PETViewgram data along the bin elements direction.

IX.3.2. Along Y-direction of a sinogram (Mashing)

Whereas, `average_views` acts only on view direction, but in that case, it is no question of a zooming but rather to sum views by 2, 4, 2N...

```
void average_views(PETSegmentByView& segment, const int num_averaged_views);

void average_views(PETSegmentBySinogram& segment, const int num_averaged_views);

void average_views(PETSinogram& sino,const int num_averaged_views);
```

IX.4. ON IMAGE DATA

```
void zoom_image(       PETImageOfVolume &image, const float zoom,
                       const float Xoff, const float Yoff, const int new_size);
```

This method implement the zooming from a 3D image volume in both directions (along x and y-axes)..

```
void zoom_image(       const PETPlane &image2D, PETPlane &new_image2D);
```

This method implement the zooming from a image plane in both directions (along x and y-axes)..

IX.5. ALGORITHM

The zooming algorithm preserves the statistics before and after zooming (i.e the total counts are equal before and after zooming) except in the case of truncation. As an example, positive zoom (zoom in) applied to 1D data is described beloz. Figure IX.1 shows the indexes used in both the initial and the final image where :

. x1 = [1, dim1];

. x2 = [1, dim2];

. pix1 corresponds to the initial pixel size;

. pix2 corresponds to the final pixel size.

Let denote also the current voxel of the initial image V_i , and the one of the final image V_f.

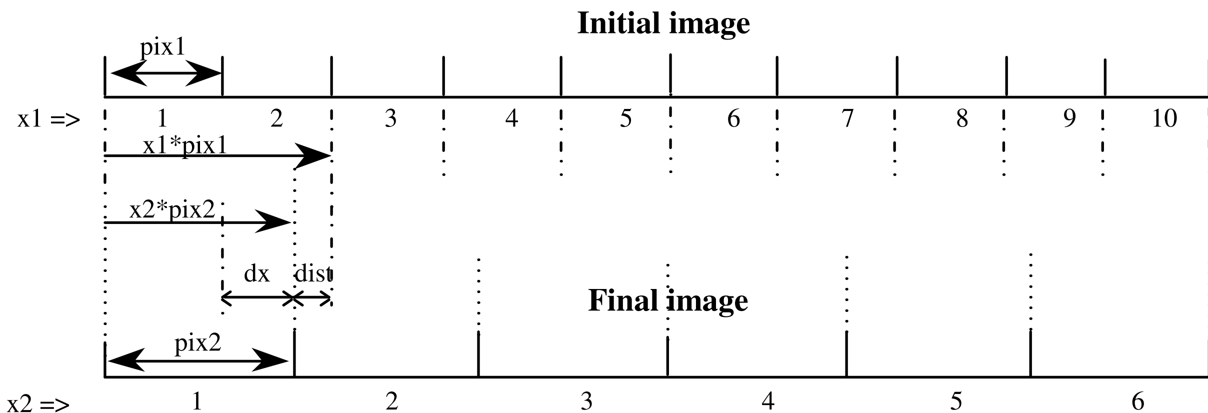


Figure IX.1: Positive zoom (zoom in) applied to 1D data with zoom = pix1/pix2.

IX.5.1. Case of zoom <= 1

```
x2 =1;
For (x1= 1 to dim1) {
        dx = x1. x2/zoom;
        if (dist <=0),
          Vf (x2+1) += zoom*Vi(x1)
        else {
          Vf (x2+1) += zoom*Vi(x1) *(1-dx);
           x2++;
        Vf (x2) += zoom*Vi(x1) *dx;
        }
}
```

IX.5.2. Case of zoom > 1

```
x1 =1;
For (x2= 1 to dim2) {
        dx = zoom*x1 - x2;
        if (dx >=0),
          Vf (x2) +=  Vi(x1) ;
        else
          Vf (x2) += Vi(x1) *(zoom-dx);
          Vf (x2 + 1) += Vi(x1)*dx;
}
```

IX.5.3. Offsetting

Offsets can be applied both in x and y-axes while rescaling. However, shifting images or sinogram by a given offset differ. Indeed, in projection data, as each point is associated with a corresponding LOR in the FOV of the detector ring, the set of LORs through a fixed point within the FOV describes a sinusoidal curve on the sinogram, as shown in figure IX.2, hence the origin of the term sinogram. So, moving the centre of the FOV is equivalent to changing the radial coordinate offset by $x\cos(\phi) + y \sin(\phi)$.

X. CLASSES FOR SCANNER INFORMATION

X.1. CLASS PETSCANNERINFO

X.1.1. Description

This is the class that contains everything relating to the scanner used for the PET data acquisition. Also, as these scanners parameters cannot be changed, a similar class to this one has been designed, PETScanInfo in order to change

the scanner parameters when zooming, offsetting, trimming and mashing is used during the reconstruction (see section IX)

X.1.2. Location

These functions are declared in

- *buildblock/PETScannerInfo.cxx*
- *include/PETScannerInfo.h*

X.1.3. Class

```
class PETScannerInfo
{
    enum Scanner_type {E951,E953,E921,E925,E961,E962,E966,ART,RPT,Advance,Exact, GPET
Unknown_Scanner};

    PETScannerInfo(Scanner_type scanner_type = E953);
    PETScannerInfo(Scanner_type type, int num_rings, int num_bins, int num_views, float
ring_radius, float FOV_radius, float FOV_axial, float ring_spacing, float                bin_size,
float intrinsic_tilt));

    Scanner_type type;                        // Scanner type where the list of available PET
                                                scanner is described just above
    int num_rings;                            // number of rings
    int num_bins;                             // number of bins (or projection elements)
    int num_views;                            // number of views (or angles)
    float ring_radius;                        // detector radius (in mm)
    float FOV_radius;                         // FOV radius (in mm)
    float FOV_axial;                          // Axial FOV (in mm)
    float ring_spacing;                       // ring spacing (or plane separation in mm)
    float bin_size;                           // bin size (spacing of transaxial elements in mm)
    float view_offset;                        // angle of first view, in radians

 public:
      PETScannerInfo(Scanner_type scanner_type = E953); // default constructor set up for the
                  953 scanner
      PETScannerInfo(Scanner_type, int num_rings, int num_bins, int num_views, float
ring_radius, float FOV_radius, float FOV_axial, float ring_spacing, float                bin_size,
float intrinsic_tilt));                 // General constructor
      void show_params(); // This function allows you to display all the information related to
                  the used PET scanner
};
```

X.1.4. Constructors

X.1.4.1. General constructor

```
PETScannerInfo(Scanner_type type, int num_rings, int num_bins, int num_views, float ring_radius,
      float FOV_radius, float FOV_axial, float ring_spacing, float bin_size, float intrinsic_tilt)
```

X.1.4.2. Constructor assuming "standard" 953 scanner

```
scannerPETScannerInfo(Scanner_type scanner_type = E953);
```

X.1.5. Data access method

X.1.5.1. void show_params()

This method displays all the informations related to the used PET scanner

X.1.6. PET scanner parameters list

The PET scanners currently defined are listed below, with parameters as in the general constructor :

```
RPT      =>set_params(RPT,   16, 128,   96,   380,   200,   108,  6.75, 3.108,    0);
E953     =>set_params(E953,  16, 160,  192, 382.5, 253.3,   108,  6.75, 3.108,    0);
Advance  =>set_params(Advance,18,281, 336, 469.5,   275,   153,  8.50,  1.96,    0);
ART      =>set_params(ART,   24, 192,  192,   205,   250,   162,  6.75,  2.44,    0);
E966     =>set_params(E966,  48, 288,  288, 206.2,   324,232.8,  4.85,  2.25,    0);
GPET     =>set_params(GEA,    0,   0,    0,     0,     0,     0,     0     0,    0);
Exact    =>set_params(Exact, 24,   0,    0,   205,     0,     0,   6.5     0,    0);
```

These following scanners are old scanners and although they are not fully completed, they are nevertheless set up in the list.

```
E951     =>set_params(E951,  16, 192,  256,   255, 300.5,   108,  6.75, 3-129,    0);
E921     =>set_params(E921,  24, 192,  192, 412.5,   324,   162,  6.75,  3.37,   15);
E925     =>set_params(E925,  24, 192,  192, 412.5,   324,   162,  6.75,  3.37,   15);
E961     =>set_params(E961,  24, 336,  196,   410,   150,     0, 3.125,  1.65,   13);
E962     =>set_params(E962,  32, 288,  144, 412.5,     0, 2.425,  2.24,    0.,    0);
```

PETscanner parameters which are not fully filled in are GPET, Exact, E962, E961

All these parameters are fixed by assuming

- there is no mashing during the acquisition or during the reconstruction, which could reduce the number of views by a multiple of 2 (generally, 2 or 4),

X.2. CLASS PETSCANINFO

X.2.1. Description

This is the class that contains almost everything relating to the scanner used for the PET data acquisition. But contrary to PETScannerInfo class, PETScanInfo is a class where some access method allows you to change the parameters scanner values.

X.2.2. Location

These functions are declared in

- *buildblock/PETScanInfo.cxx*
- *include/PETScanInfo.h*

X.2.3. Class

```
class PETScanInfo
{
private:
  int num_rings;                        // number of direct planes
  int num_bins;                         // default number of bins
  int num_views;                        // default number of angles
  float ring_radius;                    // detector radius in mm
  float ring_spacing;                   // plane separation in mm*
  float bin_size;                       // bin size in mm (spacing of transaxial elements)
  float view_offset;                    /7 angle of first view, in radians

private:
  PETScannerInfo::Scanner_type scanner_type;

public:
  PETScanInfo (const PETScannerInfo& scanner =PETScannerInfo::E953);
```

```
  PETScanInfo(PETScannerInfo::Scanner_type type_v,
              int num_rings_v, int num_bins_v, int num_views_v,
              float ring_radius_v, float ring_spacing_v, float bin_size_v,
              float view_offset_v)
    :
    scanner_type(type_v),
    num_rings(num_rings_v),
    num_bins(num_bins_v),
    num_views(num_views_v),
    ring_radius(ring_radius_v),
    ring_spacing(ring_spacing_v),
    bin_size(bin_size_v),
    view_offset(view_offset_v)

  PETScanInfo& operator=(const PETScannerInfo::Scanner_type scanner_type);
  PETScannerInfo get_scanner() const;

  int get_num_rings() const;
  int get_num_bins() const;
  int get_num_views() const;

  float get_ring_radius() const;
  float get_ring_spacing() const;
  float get_bin_size() const;
  float get_view_offset() const;

  void set_num_rings(int num_rings_v);
  void set_num_bins(int num_bins_v);
  void set_num_views(int num_views_v);
  void set_bin_size(float bin_size);
  void set_view_offset(float view_offset_v);

  void show_params() const;
};
```

X.2.4. Constructors

X.2.4.1. General constructor

```
  PETScanInfo(PETScannerInfo::Scanner_type type_v,
              int num_rings_v, int num_bins_v, int num_views_v,
              float ring_radius_v, float ring_spacing_v, float bin_size_v,
              float view_offset_v)
```

Note that in this constructor two parameters from PETScanner parameters list have been removed which are FOV_radius and FOV_axial

X.2.4.2. Constructor assuming "standard" 953 scanner

```
PETScanInfo (const PETScannerInfo& scanner =PETScannerInfo::E953)IV.5.8.1.5. Data access method
```

X.2.4.3. void show_params()

This method displays all the informations related to the used PET scanner

X.2.5. Data access method

X.2.5.1. void set_num_rings(int num_rings_v); void set_num_bins(int num_bins_v); void set_num_views(int num_views_v); void set_bin_size(float bin_size); void set_view_offset(float view_offset_v);

These methods set up the new value of the scanner parameters

XI. CLASSES FOR RECONSTRUCTION

XI.1. DESCRIPTION

These are the classes that provides the basic interface for all reconstruction processing(either analytic or iterative). This provides only the framework for reconstruction methods. Actual implementations are described in other deliverables.

XI.2. LOCATION

- *include/Reconstruction.h*

XI.3. CLASS PETRECONSTRUCTION

XI.3.1. Description

This is the base class providing the general interface reconstruction. It is an abstract class as no implementation of the reconstruct method is provided.

XI.3.2. Class

```
Class PETReconstruction
{
      virtual string method_info()
      virtual string parameter_info()
      virtual void reconstruct(PETSinogramOfVolume &s, PETImageOfVolume &i) = 0
      virtual void reconstruct(PETSinogramOfVolume &s,PETSinogramOfVolume &a, PETImageOfVolume &v)
}
```

XI.3.3. Data access method

XI.3.3.1. virtual string method_info()

This method returns the type of the reconstruction algorithm

XI.3.3.2. virtual string parameter_info()

This method returns all the parameters used for the reconstruction algorithm. This will display

- the common parameters being used for filtering (fc, alpha, transaxial extension for FFT,...),
- the parameters being used for the specific reconstruction algorithm such as for FORE, the smallest angular frequency, the delta maximum for small omega...

XI.3.3.3. virtual void reconstruct(PETSinogramOfVolumes &s, PETImageOfVolume &i) = 0

This method performs the reconstruction by giving as input the emission sinogram data *s* corrected for attenuation, dead time (see detail in D3.2 section Correction Info in emission, transmission...), and returns the output reconstructed image *i*.

XI.3.3.4. virtual void reconstruct(PETSinogramOfVolume s,PETSinogramOfVolume a, PETImageOfVolume v)

This constructor uses as input the emission data sinogram not yet corrected for attenuation *s*, the attenuation data sinogram *a*, and returns the reconstructed image *i*

XI.4. CLASS PETANALYTICRECONSTRUCTION

XI.4.1. Description

This class is designed for analytical algorithms and is derived from **PETReconstruction**.

XI.4.2. Class

```
Class PETAnalyticReconstruction: public PETReconstruction
{
      int delta_min;                    // defines the lowest limit of the "obliqueness". In
        normal use, it should be 2
      int delta_max;                    // defines the uppest limit of the "obliqueness". In
        normal use, it should be 2
      Filter filter;                    // Ramp filter with or without Hamming apodized window
      virtual string parameter_info();
      PETAnalyticReconstruction(int min, int max, const Filter1D<float>& f);
};
```

XI.5. CLASS PETITERATIVERECONSTRUCTION

XI.5.1. Description

This is the class for iterative algorithms and is derived from **PETReconstruction**

XI.5.2. Class

```
Class PETIterativeReconstruction: public PETReconstruction
{
private:
      int max_iterations;                     // Maximal number of iterations to be used
      // other stopping criterion
public:
};
```

XI.5.3. Constructors

XI.5.3.1. PETIterativeReconstruction(int max);

This constructor parameter takes as input the maximal number of iterations to be used. Some other stopping criterion could be added in the **PETIterativeReconstruction** class such as penalties, constraints which ensure that the image has certain desirable properties and allow the iterative reconstruction to converge faster and therefore to save useless computing time.

XI.6. CLASS PETRECONSTRUCTION2D

XI.6.1. Description

This is the base class for 2D reconstruction algorithms.

XI.6.2. Class

```
Class PETReconstruction2D
{
      virtual string method_info()
      virtual string parameter_info()
      virtual void reconstruct(const PETSinogram &sino2D, PETPlane &image2D)=0;
      virtual void reconstruct(const PETSegment &sino, PETImageOfVolume &image)=0;
      virtual void reconstruct(const PETSegment& s, const PETSegment& a, PETImageOfVolume& v)
};
```

XI.6.3. Data access methods

See PETReconstruction class

XI.7. CLASS PETANALYTICRECONSTRUCTION2D

XI.7.1. Description

This class is designed for analytical algorithms and is derived from **PETReconstruction**

XI.7.2. Class

```
Class PETAnalyticReconstruction2D : public PETReconstruction2D
{
      Filter1D<float> filter;
public:
      virtual string parameter_info();
      PETAnalyticReconstruction2D(Filter1D<float> &f);
};
```

XI.7.3. Constructor

XI.7.3.1. PETAnalyticReconstruction2D(Filter1D<float> &f);

This constructor uses as argument a 1D filter to be applied on sinogram data (for each view).

XI.7.4. Data access method

XI.7.4.1. virtual string method_info()

This method returns the type of the reconstruction algorithm during the reconstruction to be implemented

XI.8. CLASS RECONSTRUCT2DFBP

XI.8.1. Description

This is the class which reconstructs images using classical 2D filtered backprojection (FBP) which is the analytic solution to the two-dimensional inversion problem of recovering a 2D image from the set of its one-dimensional projections. As its name suggests, FBP consists of a filtering step followed by backprojection, or smearing back of the projection data into the image matrix. The details of the algorithm can be found in deliverable D1.3 Section 2.

XI.8.2. Class

```
Class PETReconstruction2DFBP : public PETAnalyticReconstruction2D
{
      virtual string method_info();

      Reconstruct2DFBP(const Filter1D<float>& f): PETAnalyticReconstruction2D(f);
      void reconstruct(const PETSinogram &sino2D, PETPlane &image2D);
      void reconstruct(const PETSegment &sino, PETImageOfVolume &image);
};
```

XI.8.3. Constructors

XI.8.3.1. Reconstruct2DFBP(Filter1D<float> &f)

This constructor takes a 1D float filter vector in order to filter each 2D parallel projection and to give the filtered projection. Then these 2D filtered projections are backprojected by redistributing their values uniformly along the LOR so as to form the reconstructed image.

XI.8.4. Data access method

XI.8.4.1. virtual void reconstruct(const PETSegment &sino, PETImageOfVolume &image);

This method takes as input a segment in order to return the 2D reconstructed image by backprojecting the filtered direct sinograms from sino.

XI.8.4.2. virtual void reconstruct(const PETSinogram &sino2D, PETPlane &image2D;

This method takes as input a 2D sinogram in order to return only one plane of 2D reconstructed image by backprojecting the filtered sinograms from sino2D.

XII. CLASSES FOR TENSORS

This section describes a series of classes that implement multi-dimensional arrays (1D, 2D, 3D, 4D).

XII.1. VECTORS WITH INDEX NOT STARTING AT 0 (CLASS VECTORWITHOFFSET)

XII.1.1. Description

This class provides basic functionality for vectors where indexing does not start from 0. It is used as a base class for the tensor classes. This is a template class: **VectorWithOffset<T>** defines a vector with elements of type **T**. The only assumptions made on the class **T** is that it provides assignment (**operator=**) and comparison (**operator==**).

XII.1.2. Location

- *include/VectorWithOffset.h*

XII.1.3. Class

(All methods are **inline** for efficiency).

```
template <class T>
class VectorWithOffset
{
protected:
  Int length;                           // length of matrix (in cells)
  Int start;                            // vertical starting index
  T *num;   // array to hold elements indexed by start
  T *mem;   // pointer to start of memory
  void Init();                          // default member settings
  void check_state() const;             // check if object is in a valid state
                                        // only non-empty when _DEBUG is #defined

public:
  VectorWithOffset();                   // default constructor
  VectorWithOffset(const Int hsz)       // construct a vector of given length
  VectorWithOffset(const Int hfirst, const Int hlast);
                                        // construct a vector of elements
                                        // with offsets hfirst and hlast
  VectorWithOffset(const VectorWithOffset&il)
                                        // copy constructor
```

```
   ~VectorWithOffset();                   // destructor
   void Recycle();                        // free memory and make object as if
                                          // default-constructed

   void set_offset(const Int hfirst);     // Set the offset from index hfirst
   virtual void grow(Int hfirst, Int hlast);
                                          // grow the length range of the tensor,
                                          // new elements are set to T()
   VectorWithOffset & operator= (const VectorWithOffset& il)
                                          // Assignment operator
   bool operator== (const VectorWithOffset&iv) const
                                          // comparison
   Int get_length() const;                // return length of vector
   Int get_min_index() const;             // return lowest possible index
   Int get_max_index() const;             // return highest possible index

   T& operator[] (Int i) ;                // element access
   const T& operator[] (Int i) const;// element access of 'const' vector
   void fill(const T &n);                 // set all elements equal to 'n'
};
```

XII.1.4. Future extensions

Iterators à la STL will be added. This will enable using all generic algorithms included in the STL. It will also allow using faster loops through the vector (effectively by using pointer arithmetic).

XII.2. CONSTRUCTORS

XII.2.1. Default Constructor - VectorWithOffset()

This constructor creates an empty vector with no elements.

XII.2.2. Destructor - ~VectorWithOffset()

The destructor releases all memory associated with the vector and its elements.

XII.2.3. Copy Constructor - VectorWithOffset(const VectorWithOffset&)

This constructor creates a new vector copied from the one passed as an argument.

XII.2.4. Sized Constructor - VectorWithOffset(const Int hfirst, const Int hlast)

This constructor creates a vector with indices ranging from **hfirst** to **hlast**. All elements are initialised with the default constructor of the type **T**. Note that the built-in types (like **int**) have *no* default constructor, so in **VectorWithOffset<int>(0,2)**, elements will be undefined.

XII.2.5. Sized Constructor - VectorWithOffset(const Int size)

This constructor creates a vector with indices ranging from **0** to **size–1**. See the two argument constructor for details.

XII.3. OPERATORS

XII.3.1. Assignment Operator - VectorWithOffset& operator= (const VectorWithOffset &)

This operator replaces the contents of an existing object with a copy of the vector passed as an argument, freeing and allocating memory within the object as necessary.

XII.3.2. Equality Operator - bool operator== (const VectorWithOffset &) const

This operator returns **true** if, and only if, the dimension and offset of the vector passed as an argument are the same as the current object, and all the vector elements are equal in value. The operator returns **false** otherwise.

XII.3.3. Element Access Operator - T& operator[] (const Int)

This operator returns a reference to the element specified by the index passed as an argument. Note that since a reference is returned, changing the result will modify the **VectorWithOffset**, e.g.

```
VectorWithOffset<int> vec(1,3);          // construct a vector with indices 1..3
vec[2] = 4;                              // set 2nd element to '4'
```

XII.3.4. const Element Access Operator - const T& operator[] (const Int) const

This operator is as above, but will be used when the object is **const**, e.g.

```
void f(const VectorWithOffset<int>& vec)
{
  cerr << vec[2];                        // this will print the element at index '2'
}
```

XII.4. DATA ACCESS METHODS

XII.4.1. void Recycle()

This method reclaims all memory and resets the object to the same state as an object constructed with the default constructor.

XII.4.2. void set_offset(Int hfirst)

This method sets the index of the first element in the vector to **hfirst**.

XII.4.3. virtual void grow(Int hfirst, Int hlast)

This method sets the first and last indices, changing the size of the vector if necessary. The new range of indices has to be 'larger' than the original range, except when the vector was constructed with the default constructor. New elements of the vector are initialised with the default constructor of the type **T** of the elements. Note that the built-in types (like **int**) have *no* default constructor, so new elements will be undefined.

This is a **virtual** function, which means that if a derived class overrides **grow**, the new method will be called under all circumstances. This will be used in the Tensor classes.

XII.4.4. Int get_length() const

This method returns the number of elements in the vector.

XII.4.5. Int get_min_index() const

This method returns the first valid index in the vector.

XII.4.6. Int get_max_index() const

This method returns last valid index in the vector.

XII.4.7. void fill(const T &n)

This method sets all vector elements to the value of **n**.

XII.5. PROTECTED MEMBERS

These members can be accessed by methods of this class and its derived classes only. They are irrelevant for using object of type VectorWithOffset, but are important if you want to use this class as a base class.

XII.5.1. Int length

The length of the vector. Derived classes should use get_length() instead, as this member will become **private** in the next version.

XII.5.2. Int start

The value of the first index of the vector. Derived classes should use get_min_index() instead, as this member will become **private** in the next version.

XII.5.3. T *num

This is a pointer 'into' the array of elements. It is shifted such that **num[start]** accesses the first element of the vector.

XII.5.4. T *mem

This is a pointer 'into' the array of elements. It is shifted such that **num[0]** accesses the first element of the vector. This member will be made private in the next version.

XII.5.5. void Init()

Resets above members to values indicating that the vector is empty. It does not free memory first, so should be used with care. This member will be made **private** in the next version.

XII.5.6. void check_state() const

This method checks if the object is in a valid state. Its definition is only non-empty when _DEBUG is #defined before *VectorWithOffset.h* is included. It is useful for debugging this class and any derived classes.

XII.6. VECTORS CONTAINING NUMERIC ELEMENTS (CLASS NUMERICVECTORWITHOFFSET)

XII.6.1. Description

This class provides functionality for vectors with elements of a numeric type, where indexing does not start from 0. It is derived from **VectorWithOffset**, and is used as a base class for the tensor classes. This is a template class: **NumericVectorWithOffset<T, NUMBER>** defines a vector with elements of type **T**. In addition, its elements can be modified by numeric operations with objects of type **NUMBER.** For example, matrices could be implemented as objects of type **NumericVectorWithOffset< NumericVectorWithOffset<NUMBER, NUMBER>, NUMBER>.**
In addition to assumptions made by **VectorWithOffset** on the class **T**, it is required that the class **T** defines operators +=, -=, *=, /= for arguments of type **T** and **NUMBER**.
As this is a derived class, all methods of its base class are also available, and not listed in the documentation below.

XII.6.2. Location

- *include/NumericVectorWithOffset.h*

XII.6.3. Class

(All methods are **inline** for efficiency).

```
template <class T, class NUMBER>
class NumericVectorWithOffset : public VectorWithOffset<T>
{
public:
  NumericVectorWithOffset();
  NumericVectorWithOffset(const Int hsz);
  NumericVectorWithOffset(const Int hfirst, const Int hlast);

  NumericVectorWithOffset operator+ (const NumericVectorWithOffset &iv) const;
  NumericVectorWithOffset operator- (const NumericVectorWithOffset &iv) const;
  NumericVectorWithOffset operator* (const NumericVectorWithOffset &iv) const;
  NumericVectorWithOffset operator/ (const NumericVectorWithOffset &iv) const;

  NumericVectorWithOffset operator+ (const NUMBER &iv) const;
  NumericVectorWithOffset operator- (const NUMBER &iv) const;
  NumericVectorWithOffset operator* (const NUMBER &iv) const;
  NumericVectorWithOffset operator/ (const NUMBER &iv) const;

  NumericVectorWithOffset &operator+= (const NumericVectorWithOffset &iv);
  NumericVectorWithOffset &operator-= (const NumericVectorWithOffset &iv);
  NumericVectorWithOffset &operator*= (const NumericVectorWithOffset &iv);
  NumericVectorWithOffset &operator/= (const NumericVectorWithOffset &iv);

  NumericVectorWithOffset &operator+= (const NUMBER &iv);
  NumericVectorWithOffset &operator-= (const NUMBER &iv);
  NumericVectorWithOffset &operator*= (const NUMBER &iv);
  NumericVectorWithOffset &operator/= (const NUMBER &iv);
};
```

XII.6.4. Future extensions

ANSI C++ (and gcc since version 2.8) allows the use of member templates. This would allow rewriting the operations with class **NUMBER**, such that this class could be defined as *template <class T> NumericVectorWithOffset.*. This would be more logical, and would allow using faster code in some cases (for example in adding an **int** to a matrix of **float**s).

A possible alternative is to implement the eight methods using class **NUMBER** as functions (not methods).

Using member templates would also allow defining addition of **NumericVectorWithOffset** objects with different types **T1**, **T2**.

XII.7. CONSTRUCTORS

Constructors and the destructor are exactly as in **VectorWithOffset**. (Note that the copy constructor and the destructor are automatically generated by the compiler.) It is important to remember that if **T** is a built-in type, new elements will be undefined.

XII.8. OPERATORS

XII.8.1. Arithmetic Assignment Operators - NumericVectorWithOffset& operator+= (const NumericVectorWithOffset &) and -=, *=, /=

These work element by element and call the respective operators of class **T**, for instance *T::operator+= (const T &).* If the vector at the left hand side of the assignment has a 'smaller' range than the vector at the right hand side, it is first **grow**n, e.g.

```
NumericVectorWithOffset<complex, double> v1(1,3);
NumericVectorWithOffset<complex, double> v2(0,3);
// initialise v1 and v2 somehow
v1 += v2;
assert(v1.get_min_index() == 0);
```

However, this **grow** feature is useless when **T** is a built-in type as new elements will be undefined. For such cases, use **Tensor1D<T>** instead.

As usual, the operators return a reference to the (modified) object, such that they can be used in somewhat cryptic ways, e.g.

```
some_function( v1+= v2 );
// is equivalent to
v1 += v2;
some_function( v1 );
```

XII.8.2. Arithmetic Assignment Operators - NumericVectorWithOffset& operator+= (const NUMBER &) and -=, *=, /=

These work element by element and call the respective operators of class **T**, for instance *T::operator+= (const NUMBER &),* e.g.

```
v1 += 3.14159;                          // adds pi to each element of v1
```

XII.8.3. Arithmetic Operators - NumericVectorWithOffset& operator+ (const NumericVectorWithOffset &) and -, *, /

These operators perform the arithmetic operations element by element, and return a new **NumericVectorWithOffset** object. Its range will be the larger of the two vectors in the operation. The original objects are not modified.

XII.8.4. Arithmetic Operators - NumericVectorWithOffset& operator+ (const NUMBER & n) and -, *, /

These operators perform the arithmetic operations on each element, always with the same **NUMBER** *n*.. They return a new **NumericVectorWithOffset** object. The original objects are not modified.

XII.9. NUMERIC VECTORS WITH SOME EXTRA OPERATIONS (CLASS TENSOR1D<T>)

XII.9.1. Description

This class provides extra functionality for vectors with elements of a numeric type, where indexing does not start from 0. It is derived from **NumericVectorWithOffset<T,T>**. This is a template class: **Tensor1D<T>** defines a vector with elements of type **T**.

In addition to assumptions made by **NumericVectorWithOffset** on the class **T**, assignment by **0** has to be defined. **find_min, find_max** use **T::operator<(const T)**.

As this is a derived class, all methods of its base class are also available, and not listed in the documentation below.

WARNING: for optimal performance we use **memcpy** to copy the elements of the vector, which does not call the copy constructors of the class **T**. Similarly, **read_data**, **write_data** use stream bases **read**, **write**. This is of course no problem for the built-in numeric types.

XII.9.2. Location

- *include/Tensor1D.h.* Two methods (versions of **readdata**, **writedata**) are implemented in *buildblock/Tensor1D.cxx.*

XII.9.3. Instantiation

Currently, three types are instantiated in *buildblock/Tensor1D.cxx:* **NUMBER** is either **short, unsigned short** or **float**. Using other types should give no link problems, except when using the three argument versions of **read_data**, **write_data.**

XII.9.4. Class

(All methods are **inline** for efficiency, except the two last ones).

```
template <class NUMBER>
class Tensor1D : public NumericVectorWithOffset<NUMBER, NUMBER>
{
public:
  Tensor1D();
  Tensor1D(Int size);
  Tensor1D(Int first, Int last);
  Tensor1D(const Tensor1D &il);
  Tensor1D(const NumericVectorWithOffset<NUMBER, NUMBER>& t);
                                       // conversion of base type

  Tensor1D & operator= (const Tensor1D &il);

  virtual void grow(Int hfirst, Int hlast);
                                       // initialises new elements to 0

  NUMBER& operator[] (const Int i);
  const NUMBER operator[] (const Int i);
  NUMBER get(Int i) const;             // return nth element of vector
  NUMBER set(Int i, NUMBER val);       // set nth element to value i

  Tensor1D & concat (const Tensor1D &il);
  void Add(NUMBER it);
  void Del(Int it) ; // Delete nth-indexed element from list
  int Exists(NUMBER i) const;          // does i exist in the vector?

  NUMBER sum() const;                  // add up all elements in the vector
  NUMBER sum_positive() const;         // add up all positive numbers
  NUMBER find_max() const;             // return maximum value of all elements
  NUMBER find_min() const;             // return minimum value of all elements

 // read binary data from stream
  void read_data(    istream& s,
                     const ByteOrder byte_order = ByteOrder::native);
/ write binary data to stream
  void write_data(   ostream& s,
                     const ByteOrder byte_order = ByteOrder::native) const; /
  // read from different type
```

```
  void read_data(    istream& s, NumericType type, Real& scale,
                     const ByteOrder byte_order = ByteOrder::native);
// write as different type
  void write_data(   ostream& s, NumericType type, Real& scale,
                     const ByteOrder byte_order = ByteOrder::native) const;
```

XII.10. CONSTRUCTORS

Syntax of constructors and the destructor are exactly as in **NumericVectorWithOffset**. (Note that the destructor is automatically generated by the compiler.) However, in **Tensor1D**, all new elements are initialised with 0. There is one new constructor, listed below.

XII.10.1. Tensor1D(const NumericVectorWithOffset<NUMBER, NUMBER>& t);

This constructor allows conversion of objects of the base type to **Tensor1D**.

XII.11. DATA ACCESS METHODS

XII.11.1. virtual void grow(Int hfirst, Int hlast)

As in **VectorWithOffset**, but initialises new elements to 0.

XII.11.2. Element Access - NUMBER& operator[] (const Int I)

As in **VectorWithOffset.**

XII.11.3. const Element Access - const NUMBER operator[] (const Int I)

As in **VectorWithOffset**, but does not return a reference, as objects of type **NUMBER** are supposed to be small.

XII.11.4. const Element Access - NUMBER get(Int i) const

Another syntax for accessing element **i**.

XII.11.5. const Element Access - NUMBER set(Int i, NUMBER val)

Another syntax for modifying element **i**, as opposed to *a[i] = 1*

XII.11.6. Tensor1D & concat (const Tensor1D &il)

Concatenates (appends) **il** to the vector.

XII.11.7. void Add(NUMBER it);

Appends just one number to the vector.

XII.11.8. void Del(Int n)

Delete nth-indexed element from the vector.

XII.11.9. int Exists(NUMBER i) const

Returns **0** if the number **i** occurs in the vector, non-zero otherwise.

XII.12. ARITHMETIC METHODS

XII.12.1. NUMBER sum() const

This method returns the sum of all the elements in the tensor.

XII.12.2. NUMBER sum_positive() const

This method returns the sum of all the positive numbers in the tensor.

XII.12.3. NUMBER find_max() const

This method returns the value of the largest element in the tensor.

XII.12.4. NUMBER find_min() const

This method returns the value of the smallest element in the tensor

XII.13. I/O METHODS

XII.13.1. void read_data(istream&)

Initialises the elements of the current tensor from the stream. This uses **istream::read**, so the stream should be opened in binary mode. Only the data itself is read, no information about the dimensions etc, e.g.

```
Tensor1D<float> t(-3,3);                  // a tensor with 7 elements
ifstream in("myfile.dat", ios::in | ios::binary);
t.read_data(in);                          // will read 7 floats from in
```

If byte_order is neither ByteOrder::native or equal to ByteOrder::get_native_order(), the data is swapped after reading (see also the ByteOrder class).

Error handling is currently very crude. If the stream state is bad, or if it is attempted to read past the EOF, an error message is written and **Abort()** is called.

XII.13.2. void write_data(ostream&)

Writes the elements of the current tensor out to the stream. This uses **ostream::write**, so the stream should be opened in binary mode. Only the data itself is written, no information about the dimensions etc, e.g.

```
Tensor1D<float> t(-3,3);                  // a tensor with 7 elements
t.fill(1);
ofstream out("myfile.dat", ios::out | ios::binary);
t.write_data(in);                         // will write 7 floats to out
```

If byte_order is neither ByteOrder::native or equal to ByteOrder::get_native_order(), the data is swapped before writing (see also the ByteOrder class).

Error handling is currently very crude. If the stream state is bad, an error message is written and **Abort()** is called.

XII.13.3. void read_data(istream&, NumericType ntype, Real& scale_factor)

This function allows to read data stored on disk in a different format as you want to have it in memory, e.g. data is stored as short, but you need to read it into an object of class **Tensor1D<float>**. The type of the data in the stream is given by **ntype**. The third argument **scale_factor** will be set such that (ignoring types) data_in_tensor == data_on_disk

* scale_factor If **scale_factor** is initialised to 0, the maximum range of type **NUMBER** is used. If **scale_factor** is non-zero, **read_data** attempts to use the given **scale_factor**, unless the **NUMBER** range does not fit. In that case, the same **scale_factor** is used as in the 0 case.

There is an effective threshold at 0 currently (i.e. negative numbers in the input are stored as zero) when **NUMBER** is an unsigned type.

If byte_order is neither ByteOrder::native or equal to ByteOrder::get_native_order(), the data is swapped after reading (see also the ByteOrder class).

See also **convert()**.

```
Tensor1D<float> float_tensor(-3,3);      // a tensor with 7 elements
ifstream in("myfile.dat", ios::in | ios::binary);
Real scale = Real(1);
// read data from myfile.data, which contains ints
float_tensor.read_data(in, NumericType::INT, scale);
// as the range of float is larger than the range of int,
// scale will not be modified
assert(scale == 1);

Tensor1D<short> short_tensor(-3,3);      // a tensor with 7 elements
scale = Real(0);
short_tensor.read_data(in, NumericType::INT, scale);
// now, scale will depend on the maximum of the data read

// check if data in the 2 tensors are equal, up to rounding errors
assert( fabs(short_tensor[2]*scale / float_tensor[2] - 1) < 1E-5 );
```

XII.13.4. void write_data(ostream&, NumericType ntype, Real& scale)

This function allows to write data to disk in a different format as you have it in memory, e.g. data is in the tensor are short, but you need to write it as float. The type of the data in the stream is given by **ntype**. The third argument **scale_factor** will be set such that (ignoring types) data_in_tensor == data_on_disk * scale_factor. If **scale_factor** is initialised to 0, the maximum range of type **NUMBER** is used. If **scale_factor** is non-zero, **read_data** attempts to use the given **scale_factor**, unless the **NUMBER** range does not fit. In that case, the same **scale_factor** is used as in the 0 case.

There is an effective threshold at 0 currently (i.e. negative numbers in the input are written as zero) when **ntype** is an unsigned type.

If byte_order is neither ByteOrder::native or equal to ByteOrder::get_native_order(), the data is swapped after reading (see also the ByteOrder class).

See also **convert()**.

XII.14. A BASE CLASS FOR HIGHER DIMENSIONAL VECTORS (CLASS TENSORBASE<T, NUMBER>)

XII.14.1. Description

This class is the base class that will be inherited by all the tensors 2D, 3D and 4D. Multi-dimensional tensors are simply NumericVectors of tensors of a lower dimension. This class is thus very short. It only adds a few members where

recursion is used. For example, sum() simply computes the sum of sum() called for each element. This all works because Tensor1D has appropriate definitions for each of these 'recursive' members.

Objects of class **Tensorbase** should never be used directly. However, its methods are usable for objects of all the higher dimensional tensors.

XII.14.2. Location

- *include/Tensorbase.h*

XII.14.3. Class

(All methods are **inline** for efficiency).

```
template <class T, class NUMBER>
class Tensorbase : public NumericVectorWithOffset<T, NUMBER>
{
  Tensorbase()                           // construct an empty Tensorbase
  Tensorbase(const Int hsz)              // construct a Tensorbase of given length
  Tensorbase(const Int hfirst, const Int hlast)
                                         // construct a Tensorbase of elements with
                                         // offsets hfirst, hlast

  NUMBER sum() const;                    // return sum of all elements
  NUMBER sum_positive() const;           // return sum of all positive elements
  NUMBER find_max() const;               // return the maximal value in the tensor
  NUMBER find_min() const;               // returns the minimal value of the tensor
  void fill(const NUMBER &n);            // Fill elements with value n
};
```

XII.15. CONSTRUCTORS

Syntax of constructors and the destructor are exactly as in **NumericVectorWithOffset**. (Note that the defaults constructor and the destructor are automatically generated by the compiler.)

XII.16. ARITHMETIC METHODS

XII.16.1. NUMBER sum() const

This method returns the sum of all the elements in the tensor.

XII.16.2. NUMBER sum_positive() const

This method returns the sum of all the positive numbers in the tensor.

XII.16.3. NUMBER find_max() const

This method returns the value of the largest element in the tensor.

XII.16.4. NUMBER find_min() const

This method returns the value of the smallest element in the tensor.

XII.17. 2D TENSORS (TENSOR2D<NUMBER>)

XII.17.1. Description

This is the class containing everything for the creation and manipulation of a 2D tensor. As it is derived from **Tensorbase**, all its methods (and hence those from **NumericVectorWithOffset** and **VectorWithOffset**) can be used.

XII.17.2. Location

- *include/Tensor2D.h*

- *buildblock/Tensors.cxx*

XII.17.3. Class

```
template <class NUMBER>
class Tensor2D : public Tensorbase<Tensor1D<NUMBER>, NUMBER>
{
private:
  Int width;
  Int widstart;
  void Init();
  void check_state() const;
public:

  Tensor2D();
  Tensor2D(Int hsz, Int wsz);
  Tensor2D(Int hfirst, Int hlast, Int wfirst, Int wlast);
  Tensor2D(const Tensor2D &il);
  Tensor2D(const Tensorbase<Tensor1D<NUMBER>, NUMBER> &il);

  Tensor2D & operator= (const Tensor2D &il);
  Tensor2D & operator= (const Tensorbase<Tensor1D<NUMBER>, NUMBER> &il);


  virtual void grow(Int first2, Int last2);
  void grow_height(Int first2, Int last2);
  void grow_width(Int wfirst, Int wlast);
  void grow(Int hfirst, Int hlast, Int wfirst, Int wlast);

  Int get_length2() const;
  Int get_min_index2() const;
  Int get_max_index2() const;
  Int get_length1() const;
  Int get_min_index1() const;
  Int get_max_index1() const;

  Int get_width() const;                 // return width of matrix
  Int get_height() const ;               // return height of matrix
  Int get_h_min() const;
  Int get_w_min() const;
  Int get_h_max() const;
  Int get_w_max() const;

  void set_offsets(const Int hfirst, const Int wfirst);

  NUMBER set(Int y, Int x, NUMBER value);
  NUMBER get(Int y, Int x) const;

  Tensor2D operator+ (const Tensor2D &iv) const;
  Tensor2D operator- (const Tensor2D &iv) const;
  Tensor2D operator+ (const Tensor2D &iv) const;
```

```
  Tensor2D operator/ (const Tensor2D &iv) const;
  void read_data(istream& s,
         const ByteOrder byte_order = ByteOrder::native);
  void write_data(ostream& s,
         const ByteOrder byte_order = ByteOrder::native) const;
  void read_data(istream& s, NumericType type, Real& scale,
         const ByteOrder byte_order = ByteOrder::native);
  void write_data(ostream& s, NumericType type, Real& scale,
         const ByteOrder byte_order = ByteOrder::native) const;
};
```

XII.18. CONSTRUCTORS

Syntax of constructors and the destructor are exactly as in **NumericVectorWithOffset**. (Note that the destructor is automatically generated by the compiler.) However, in **Tensor2D**, all new elements are initialised with 0. There is one new constructor, listed below.

XII.18.1. Tensor2D(const Tensorbase<Tensor1D<NUMBER>, NUMBER> & t);

This constructor allows conversion of objects of the base type to **Tensor2D**.

XII.19. OPERATORS

See **Tensorbase** for most operators. In particular, Tensor1D& operator[] returns a reference to the row vector (**Tensor1D**) specified by the index passed as an argument. This gives the **Tensor2D** class the appearance of being an array of row vectors. Note that since a reference is returned, anything done to the result vector is reflected in the matrix itself - in particular, if the **[] operator** is used on the vector, individual matrix elements can be accessed and changed, e.g.

```
Tensor2D<int> matrix(1,3,1,3);          // construct 3 by 3 matrix
matrix[2][3] = 4;                       // set 2nd row, 3rd column to ë4í
```

XII.19.1. Assignment operator - Tensor2D & operator= (const Tensor2D &il)

Assigns a new tensor. The old object is deallocated first.

XII.19.2. Assignment operator - Tensor2D & operator= (const Tensorbase<Tensor1D<NUMBER>, NUMBER>&il)

This method allows assignment of objects of the base type without the need of a conversion (which would copy the data).

XII.19.3. Arithmetic Operators - Tensor2D & operator+ (const Tensor2D & t2) and -, *, /

These operators perform the arithmetic operations element by element, and return a new **Tensor2D** object. Its range will be the larger of the two vectors in the operation. The original objects are not modified.

The operators simply call the corresponding versions of **NumericVectorWithOffset**, but are repeated in this class for efficiency.[1]

---

[1] Implementation of these methods work by constructing a temporary object, initialised by ***this**, and then calling **operator+=(t2)**. If **operator+()** of the base class would be used, the temporary object would be of type **NumericVectorWithOffset**, and an extra conversion would be needed to **Tensor2D**.

XII.20. DATA ACCESS METHODS

XII.20.1. void set_offsets(Int row, Int column)

This method sets the row and column index of the top-left element of the matrix.

XII.20.2. void grow_height(Int first, Int last) and void grow(Int first, Int last)

These equivalent methods set the first and last row indices, changing the size of the matrix if necessary. This essentially calls **VectorWithOffset::grow**, but initialises new elements to **Tensor1D** objects (filled with 0) of the correct dimensions.

XII.20.3. void grow_width(Int first, Int last)

This method sets the first and last column indices, changing the size of the matrix if necessary.

XII.20.4. void grow(Int hfirst, Int hlast, Int wfirst, Int wlast)

This method sets the first and last row (**h**) and column (**w**) indices, changing the size of the matrix if necessary.

XII.20.5. NUMBER set(Int row, Int col, NUMBER value)

This method sets the element specified by **row** and **col** to the NUMBER **value**, returning this value. Equivalent syntax is *a[row][col] = value;*

XII.20.6. NUMBER get(Int row, Int col) const

This method returns the value of the element at the specified **row** and **col.** Equivalent syntax is *a[row][col].*

XII.20.7. Int get_width() const

This method returns the number of columns in the matrix.

XII.20.8. Int get_height() const

This method returns the number of rows in the matrix.

XII.20.9. Int get_h_min(), Int get_w_min(), Int get_h_max(), Int get_w_max()

These four methods return the indexes of the first row, first column, last row and last column, respectively.

XII.20.10. Int get_length1(), Int get_min_index1(), Int get_max_index1()

Equivalent to get_width(), get_w_min() and get_w_max(), respectively.

XII.20.11. Int get_length2(), Int get_min_index2(), Int get_max_index2()

Equivalent to get_height(), get_h_min() and get_h_max(), respectively.

XII.21. I/O METHODS

XII.21.1. void read_data(istream&, const ByteOrder byte_order = ByteOrder::native)

Initialises the elements of the current tensor from the stream. See Tensor1D::read_data for details. Data are read by calling **read_data** recursively on each element. This corresponds to the usual C convention that in multidimensional arrays, the last index runs fastest.

XII.21.2. void write_data(ostream&, const ByteOrder byte_order = ByteOrder::native)

Writes the elements of the current tensor out to the stream. . See Tensor1D::write_data for details. Data are written by calling **write_data** recursively on each element. This corresponds to the usual C convention that in multidimensional arrays, the last index runs fastest.

XII.21.3. void read_data(istream&, NumericType ntype, Real& scale_factor, const ByteOrder byte_order = ByteOrder::native)

See Tensor1D::read_data for details.

XII.21.4. void write_data(ostream&, NumericType ntype, Real& scale, const ByteOrder byte_order = ByteOrder::native)

XII.22. IMPLEMENTATION DETAILS

The function **check_state()** is again used for debugging purposes only. It is only non-empty when **_DEBUG** is #defined to 2 or higher. This is because it checks on uniformity of the tensor ranges, and is fairly CPU intensive (especially as it is called very often). For lower values of **_DEBUG**, **VectorWithOffset::check_state()** is called by all methods of **Tensorbase**.

Because of the class hierarchy, each element in a **Tensor2D** is a **Tensor1D.** This implies a memory overhead (as size/offset of the rows are stored more than once), and means that data is not stored contiguously. This is roughly like using float ** for 2D arrays. One consequence is that for higher dimensional tensors, memory allocation/deallocation overhead can become significant. However, it makes **grow**ing of the outer index ranges much more efficient. Also, it means we had to write less code because inheritance does most of the job.

The class keeps **widstart**, **width** as private members, specifying the size/offset of the rows. These parameters could be inferred from one of the rows instead, but it was hoped that the current solution is slightly more efficient.

Users of this class should not rely on the class hierarchy we used. Proper programming style would have made **Tensorbase** a private (or protected) base class, where relevant operators would be made public. However, this requires a lot of extra lines, and was felt to be over-the-top for our purposes.

XII.23. 3D TENSORS (CLASS TENSOR3D<NUMBER>)

XII.23.1. Description

This is the class containing everything for the creation and manipulation of 3D tensors. As it is derived from **Tensorbase**, all its methods (and hence those from **NumericVectorWithOffset** and **VectorWithOffset**) can be used. The set-up of this class is identical to **Tensor2D**. So we only give the class definition below.

XII.23.2. Location

- *include/Tensor3D.h*
- *buildblock/Tensors.cxx*

XII.23.3. Class

```
template <class NUMBER>
class Tensor3D : public Tensorbase<Tensor2D<NUMBER>, NUMBER>
{
private:
  Int width;
  Int height;
  Int widstart;
  Int hgtstart;
  void Init();
  void check_state() const;

public:
  Tensor3D();
  Tensor3D(Int npl, Int hsz, Int wsz);
  Tensor3D(Int first3, Int last3, Int hfirst, Int hlast, Int wfirst, Int wlast);
  Tensor3D(const Tensor3D &il);
  Tensor3D(const Tensorbase<Tensor2D<NUMBER>, NUMBER> &il);

  Tensor3D & operator= (const Tensor3D &il);
  Tensor3D & operator= (const Tensorbase<Tensor2D<NUMBER>, NUMBER> &il);

  Int get_length3() const;
  Int get_min_index3();
  Int get_length2() const;
  Int get_min_index2() const;
  Int get_max_index2() const;
  Int get_length1() const;
  Int get_min_index1() const;
  Int get_max_index1() const;

  virtual void grow(Int first3, Int last3);
  void grow_dim3(Int first3, Int last3);
  void grow_height(Int hfirst, Int hlast);
  void grow_width(Int wfirst, Int wlast);
  void grow(Int first3, Int last3, Int hfirst, Int hlast, Int wfirst, Int wlast);

  NUMBER set(Int z, Int y, Int x, NUMBER value);
  NUMBER get(Int z, Int y, Int x) const;
  Tensor3D operator+ (const Tensor3D &iv) const;
  Tensor3D operator- (const Tensor3D &iv) const;
  Tensor3D operator* (const Tensor3D &iv) const;
  Tensor3D operator/ (const Tensor3D &iv) const;

  void read_data(istream& s,
         const ByteOrder byte_order = ByteOrder::native);
  void write_data(ostream& s,
         const ByteOrder byte_order = ByteOrder::native) const;
  void read_data(istream& s, NumericType type, Real& scale,
         const ByteOrder byte_order = ByteOrder::native);
  void write_data(ostream& s, NumericType type, Real& scale,
         const ByteOrder byte_order = ByteOrder::native) const;
```

```
};
```

XII.24. 4D TENSORS (CLASS TENSOR4D<NUMBER>)

XII.24.1. Description

This is the class containing everything for the creation and manipulation of 4D tensors. As it is derived from **Tensorbase**, all its methods (and hence those from **NumericVectorWithOffset** and **VectorWithOffset**) can be used. The set-up of this class is identical to **Tensor2D**. So we only give the class definition below.

XII.24.2. Location

- *include/Tensor4D.h*

- *buildblock/Tensors.cxx*

XII.24.3. Class

```
template <class NUMBER>
class Tensor4D : public Tensorbase<Tensor3D<NUMBER>, NUMBER>
{
private:
  Int width;
  Int height;
  Int size3;
  Int widstart;
  Int hgtstart;
  Int start3;

  void Init();
  void check_state() const;

public:
  Tensor4D();
  Tensor4D(Int sz4, Int npl, Int hsz, Int wsz)
  Tensor4D (Int first4, Int last4, Int first3, Int last3,
             Int hfirst, Int hlast, Int wfirst, Int wlast);
  Tensor4D(const Tensor4D &il);
  Tensor4D(const Tensorbase<Tensor2D<NUMBER>, NUMBER> &il);

  Tensor4D & operator= (const Tensor4D &il);
  Tensor4D & operator= (const Tensorbase<Tensor3D<NUMBER>, NUMBER> &il);

  virtual void grow(Int first4, Int last4);
  void grow_dim4(Int first4, Int last4);
  void grow_dim3(Int first3, Int last3);
  void grow_height(Int hfirst, Int hlast);
  void grow_width(Int wfirst, Int wlast);
  void grow(Int first4, Int last4, Int first3, Int last3,
             Int hfirst, Int hlast, Int wfirst, Int wlast);


  NUMBER set(Int t, Int z, Int y, Int x, NUMBER value);
  NUMBER get(Int t, Int z, Int y, Int x) ;

  Tensor4D operator+ (const Tensor4D &iv) const;
  Tensor4D operator- (const Tensor4D &iv) const;
  Tensor4D operator* (const Tensor4D &iv) const;
```

```
  Tensor4D operator/ (const Tensor4D &iv) const;

  void read_data(istream& s,
         const ByteOrder byte_order = ByteOrder::native);
  void write_data(ostream& s,
         const ByteOrder byte_order = ByteOrder::native) const;
  void read_data(istream& s, NumericType type, Real& scale,
         const ByteOrder byte_order = ByteOrder::native);
  void write_data(ostream& s, NumericType type, Real& scale,
         const ByteOrder byte_order = ByteOrder::native) const;
};
```

XII.25. SOME FUNCTIONS TO MANIPULATE TENSORS

XII.25.1. Description

These functions provide some additional tools for manipulating multi-dimensional arrays:

- functions which work on all Tensor*n*D objects, and which change every element of the tensor. They all work element-wise.

These following functions are in sublevel

- in_place_log,
- in_place_exp (these work only well when elements are float or double,
- in_place_abs,
- in_place_apply_function
- functions specific to Tensor1D: inner_product, norm, angle
- functions specific to Tensor2D: matrix_transpose, matrix_multiply

All the functions are templated such that they work (if possible) on tensors of any numeric type.

XII.25.2. Location

- *include/TensorFunction.h*

XII.25.3. Prototypes

```
template <class NUMBER>
Tensor1D<NUMBER>& in_place_log(Tensor1D<NUMBER>& v);

template <class T, class NUMBER>
Tensorbase<T, NUMBER>& in_place_log(Tensorbase<T, NUMBER>& v);

template <class NUMBER>
Tensor1D<NUMBER>& in_place_exp(Tensor1D<NUMBER>& v);

template <class T, class NUMBER>
Tensorbase<T, NUMBER>& in_place_exp(Tensorbase<T, NUMBER>& v);

template <class NUMBER>
Tensor1D<NUMBER>& in_place_abs(Tensor1D<NUMBER>& v);

template <class T, class NUMBER>
Tensorbase<T, NUMBER>& in_place_abs(Tensorbase<T, NUMBER>& v);

template <class NUMBER, class FUNCTION>
```

```
Tensor1D<NUMBER>& in_place_apply_function(Tensor1D<NUMBER>& v, FUNCTION f);

template <class T, class NUMBER, class FUNCTION>
Tensorbase<T, NUMBER>& in_place_apply_function(Tensorbase<T, NUMBER>& v, FUNCTION f);

// functions specific for Tensor1D

template<class NUMBER>
NUMBER
inner_product (const Tensor1D<NUMBER> & v1, const Tensor1D<NUMBER> &v2);

template<class NUMBER>
double
norm (const Tensor1D<NUMBER> & v1);

template<class NUMBER>
NUMBER
angle (const Tensor1D<NUMBER> & v1, const Tensor1D<NUMBER> &v2)

// functions specific for Tensor2D

template <class NUMBER>
Tensor2D<NUMBER>
matrix_transpose (const Tensor2D<NUMBER>& matrix);

template <class NUMBER>
Tensor2D<NUMBER>
matrix_multiply(const Tensor2D<NUMBER> &m1, const Tensor2D<NUMBER>& m2);
```

XII.25.3.1. in_place_log

Modify each element of the tensor by replacing it with its **log**(). This works only well when **log(NUMBER)** returns a **NUMBER,** for instance with **float** or **double** elements.

XII.25.3.2. in_place_exp

Modify each element of the tensor by replacing it with its **exp**(). This works only well when **log(NUMBER)** returns a **NUMBER,** for instance with **float** or **double** elements.

XII.25.3.3. in_place_abs

Modify each element of the tensor by replacing it with its absolute value. This uses **NUMBER::operator<(NUMBER)**, so will not work for e.g. complex numbers. Adding this functionality requires a C++ compiler that allows template specialisation (as gcc 2.8.1).

XII.25.3.4. in_place_apply_function

Modify each element of the tensor by replacing it with the result of **f** applied on it. To work properly, **NUMBER f(const NUMBER)** must be defined. Because of the template mechanism, current compilers seem to have trouble with overloaded functions.

XII.25.3.5. inner_product

Returns the inner (or scalar) product of two vectors as a **NUMBER**: $\sum_i v1_i * v2_i$. The vectors can have different sizes and/or offsets.

XII.25.3.6. norm

Returns the norm of the vector, here defined as the **sqrt** of the inner product of the vector with itself. This definition is unsuitable for complex numbers.

XII.25.3.7. angle

Returns the angle between two vectors, here defined as the **acos** of (the inner product of the two vector divided by the two norms) . This definition is unsuitable for complex numbers

XII.25.3.8. matrix_transpose

Returns a new **Tensor2D** object with columns and rows interchanged.

XII.25.3.9. matrix_multiply

Returns a new Tensor2D object, containing the matrix product of the two arguments. The index range of the rows of m1 must exactly match the range of the columns of m2.

XII.26. CONVERSION OF TENSOR OBJECTS : CONVERT

XII.26.1. Description

A templated set of functions to convert Tensor objects of different numeric types. The functions are implemented in *buildblock/convert.cxx*. However, because these are templated functions,. this presents a number of problems for the linker, i.e. how to generate a new template instantiation at link time. We circumvented this problem by explicitly instantiating the functions for the types we use (currently conversions between float and signed/unsigned short Tensors).

The functions are templated for generality. In the text below, we use the following names for the templated types:

NUMBER is the type of elements in the Tensor3D or Tensor2D object.

SCALE is the type of the scale factors

CHARP is the type of the text strings (char * or String)

If the input-type is a signed type, while the output type is unsigned, there is an effective threshold at 0 (i.e. negative numbers are cut out).

XII.26.2. Location

- include/convert.h

- buildblock/convert.cxx.*cxx*

XII.26.2.1. Definition

The definition for a Tensor1D object is as follows:

```
template <class T1, class T2, class SCALE>
Tensor1D<T2>
convert(const Tensor1D<T1>& data_in,
      const NumericInfo<T2> info2,
      SCALE& scale_factor);
```

Similar definitions exist for Tensor2D, Tensor3D and Tensor4D objects.

XII.26.2.2. Usage

```
data_out = convert(data_in, NumericInfo<T2>(), scale_factor)
```

Where the arguments are:

data_in :

some Tensor object, elements are of some numeric type T1

2nd parameter :

T2 is the desired output type

scale_factor :

a reference to a (float or double) variable which will be set to the scale factor such that (ignoring types)

data_in == data_out * scale_factor

If scale_factor is initialised to 0, the maximum range of T2 is used. If scale_factor != 0, convert attempts to use the given scale_factor, unless the T2 range doesn't fit. In that case, the same scale_factor is used as in the 0 case.

The return value is a Tensor object (of the same dimension as *data_in)* whose elements are of numeric type T2.

XII.26.2.3. Implementation details

The implementation of these functions is rather primitive, and taken from an old library by KT. This library runs on any Xwindows system, but also on PCs in MSDOS mode (a number of (older) graphics cards for higher resolution are supported). This implementation is only intended for testing purposes.

XII.27. CLASSES WITH INFORMATION ON THE (BUILT-IN) NUMERIC TYPES

Two classes providing a standard way of obtaining information on a numeric type. This replaces the use of ANSI C headers like */usr/include/limits.h.*

XII.27.1. Class NumericType

XII.27.1.1. Description

A class with names for the built-in numeric types and some properties. Objects of this type can be used to pass type information to functions.

XII.27.1.2. Location

- *include/NumericInfo.h*

XII.27.1.3. Class

```
{
  enum Type { BIT, SCHAR, UCHAR, SHORT, USHORT, INT, UINT, LONG, ULONG,
      FLOAT, DOUBLE, UNKNOWN_TYPE };

  NumericType(Type t);
  NumericType(const string number_format, const int bytes_per_pixel);

  bool operator==(NumericType type) const;
```

```
  size_t size_in_bytes() const;
  size_t size_in_bits() const;
}
```

XII.27.1.4. Public types

XII.27.1.5. Type

Enumerates the numeric types. No support of non-ANSI types like *long long int* or *long double* is included.

X1.4.2. Constructors

XII.27.1.5.1. NumericType(Type t)

This constructor essentially provides a conversion from an object of type NumericType::Type to a NumericType object.

XII.27.1.5.2. NumericType(const string number_format, const int bytes_per_pixel)

A constructor to work from named types á la Interfile. Possible contents for the number_format are "bit, "signed integer", "signed integer", "float". Exact types are determined via the bytes_per_pixel parameter.

XII. 26.1.4.3. Data access members

27.1.5.XII.3. size_t size_in_bytes() const

Returns the size of an object of the relevant type, as reported by sizeof(), which is on most (all?) machines the size in bytes.

27.1.5.XII.4. size_t size_in_bits() const

Returns the size of an object of the relevant type, in number of bits.

XII.28. TEMPLATE <CLASS NUMBER> CLASS NUMERICINFO

XII.28.1. Description

A template class that defines properties for the type NUMBER, which should be one of the types listed in NumericType::Type. Objects of such a class contain no data. This class is mainly useful to pass type information to a function. See the *convert* functions below for an example.

XII.28.2. Location

- *include/NumericInfo.h*

XII.28.3. Class

```
{
  bool signed_type() const;
  bool integer_type() const;
  size_t size_in_bits() const;
  size_t size_in_bytes() const;

  NUMBER max_value() const;
  NUMBER min_value() const;

  NumericType type_id() const;
};
```

XII.28.4. Data access members

XII.28.4.1. bool signed_type() const

A method that returns *true* if the NUMBER type is signed.

XII.28.4.2. bool integer_type() const

A method that returns *true* if the NUMBER type is integer valued (*signed char* is considered to be integer valued).

XII.28.4.3. size_t size_in_bytes() const, size_t size_in_bits() const

Methods that return the size of a NUMBER object. See also the corresponding methods of the *NumericType* class.

XII.28.4.4. NUMBER max_value() const, NUMBER min_value() const

Methods that return the maximum and minimum value that can be stored in an object of type NUMBER.

XII.28.4.5. NumericType type_id() const;

This methods returns an object of class *NumericType* corresponding to the NUMBER type.

XII.29. A CLASS FOR SPECIFYING BYTE ORDER (CLASS BYTEORDER)

XII.29.1. Description

This class provides facilities to specify a certain byte order (i.e. little or big endian) and compare it with the byte order native to the processor on which the program runs. It also provides methods to perform byte_order swapping.

XII.29.2. Location

- *include/NumericInfo.h*
- *buildblock/NumericInfo.cxx*

XII.29.3. Class

```
class ByteOrder
{
public:
enum Order { little_endian, big_endian, native, swapped };
inline static Order get_native_order();
template <class NUMBER>  void static swap_order(NUMBER& a);

ByteOrder(Order byte_order = native);

  bool operator==(ByteOrder order2) const;
  inline bool is_native_order() const;
  template <class NUMBER>  void swap_if_necessary(NUMBER& a) const;
};
```

XII.29.4. Public types

enum Order { little_endian, big_endian, native, swapped }

'little_endian' is like x86, MIPS, Alpha processors, 'big_endian' is like PowerPC, Sparc. 'native' means the same order as the machine the program is running on, 'swapped' means the opposite order.

XII.29.5. Constructors

ByteOrder(Order byte_order = native)
This constructs an object of the specified byte order. As it is a single argument constructor, it also provides an automatic conversion operator from a ByteOrder::Order object.

XII.29.6. Static members

inline static Order get_native_order()
This returns the order of the processor the program is running on.

```
if(ByteOrder::little_endian == ByteOrder::get_native_order())
 // do something appropriate for little endian machines
else
 // do something appropriate for big endian machines
```

template <class NUMBER> void static swap_order(NUMBER& a)
This 'converts' the number to the other byte order. It is currently only implemented for the built-in types.

XII.29.7. Data access members

bool operator==(ByteOrder order2) const
This checks if both orders are the same.

inline bool is_native_order() const
This checks if the object corresponds to the native order.

template <class NUMBER> void swap_if_necessary(NUMBER& a) const
This performs byte-order swapping on the number if the current object does not correspond to the native order.

XII.29.8. Implementation

For efficiency reasons, the get_native_order() method uses a static member. This is initialised in buildblock/NumericInfo.cxx.

On Windows compilers (except gcc), it is necessary to link with an specific Import Library:

- Windows NT or Windows 95/98: ws2_32.lib
- Win32s: wsock32.lib

This class uses member templates for convenience. If your compiler cannot handle it, you will have to write lots of (straightforward) lines for the built-in types.

XIII. CLASSES FOR FILTERS

XIII.1. DESCRIPTION

This contains all the filtering utilities including the 1D, 2D filter and the subclasses of these filters such as 1D Ramp and 2D Colsher filters).

XIII.2. LOCATION

- *include/Filter.h*
- *recon_buildingblock/filt1DRamp.cxx*

- *analytic/PROMIS/filtColsher.cxx*
- *analytic/PROMIS/filtColsher_view.cxx*

XIII.3. SOFTWARE IMPLEMENTATION

• The filtering step is performed using the FFT routines details in section XIII, which require the array dimensions to be integer powers of two. Symmetry properties of the Fourier transform of real signals are exploited to transform two real arrays at a time, by packing them into the real and imaginary parts of a complex array respectively, cutting the filtering time roughly in half.

• Aliasing in the discrete convolution is avoided by setting up a buffer zone of zeroes at one end of the projection; in order to guarantee artefact-free convolutions the number of zeroes must be greater than or equal to the maximum positive extension, or maximum negative extension, of the filter kernel. For an original array of 128x31 elements, the use or not of zero-padding results in a difference of a factor of over five in the time required for the Fourier transformations. While the usefulness of the technique is well established, it is also worthwhile examining in what cases it is indispensable, and in what other cases the increase in computational requirements is not justified.

• While the transaxial ramp filter can be shown analytically to decrease as $1/s^2$, the extension of the other filters (Colsher [Col80], Favor [Def92], with or without an apodising window), though somewhat larger, remains strongly peaked at $s = 0$ [Egg96]. Since ten or twenty padding zeroes may be enough, an array of width, say, 200 elements needs only be padded up to 256, not 512. Moreover, in many cases, such as brain studies, the object does not fill the entire transaxial field of view, and assuming good scatter and randoms corrections, the projection is naturally zero-padded by the absence of a signal outside the object. Additional transaxial padding may be dispensed with in such cases. This would not apply to studies of objects that fill the entire field of view, unless one knows *a priori* that one is not interested in the peripheral regions. Nor does it apply to the axial direction, the imaged object usually extending beyond the axial field of view of the tomograph.

• These considerations lead us to adopt the following padding scheme for the results shown below: zero-padding up to the next power of two in the transaxial direction, and up to the power of two greater than *double* the projection size in the axial direction.

XIII.4. 1D FILTERS

XIII.4.1. class Filter1D

XIII.4.1.1. Description

This is the base class for general 1D filter which contains two major parameters, the length of the filter and the value vector to be applied on data to be filtered

XIII.4.1.2. Location

- *include/Filter.h*

XIII.4.1.3. Class

```
Template <class T> class Filter1D
{
 protected:
      int length;                          // length of the filter
      Tensor1D <float> filter;             // vector containing the value of the 1D filter
 public:
      Filter1D(int length_v);              // Default constructor for Filter1D
      void apply(Tensor1D <T> &data);      // Member function for applying 1D filter to 1D
                                          data array
      void apply(Tensor2D <T> &data);      // Member function for applying 1D filter to the
                                          last index of 2D data array
      void apply(Tensor3D <T> &data);      // Member function for applying 1D filter to the
                                          last index of 3D data array
};
```

XIII.4.1.4. Constructors

I. general constructor

Filter1D(int length_v).

This is the default constructor where the length of the filter must be specified

XIII.4.1.5. Data access methods

void **apply**(Tensor1D <T> &ibuf), void apply(Tensor2D <T> &ibuf), void apply(Tensor3D <T> &ibuf)

I. void apply(Tensor1D <T> &data);

This method applies 1D filter to 1D data

II. void apply(Tensor2D <T> &data);

This method applies 1D filter to 2D data

III. void apply(Tensor3D <T> &data);

This method applies 1D filter to 3D data

XIII.4.1.6. Protected members

These members can be accessed by methods of this class and its derived classes only.

I. int length

The length of the filter.

II. Tensor1D <float> filter

The vector containing the values of the 1D filter to be applied on data. Also the filter is defined in frequency space

XIII.4.2. Ramp Filter (class Filter1DRamp)

XIII.4.2.1. Description

The Ramp filter is derived from 1D filter which contains three main parameters

- the cut-off frequency (fc)

- the alpha parameter in order to limit the noise amplification

- the sampling distance .

XIII.4.2.2. Location

- *recon_buildingblock/filtRamp.cxx*
- *include/Filter.h*

XIII.4.2.3. Class

This class is derived from **Filter1D** class.

class Filter1DRamp : public Filter1D<float>

```
{
public:
      float fc;                        // value of the cut-off frequency of the ramp filter
      float alpha;                     // value of the alpha parameter for Hamming window
      float sampledist;                // Sampling distance as it is the 1D filter is a
             transaxial filter in frequency space

      Filter1DRamp(float sampledist_v, int length_v , float alpha_v=1, float fc_v=.5);
      void show_params();              // member function to display all the Ramp filter
           parameters values
};
```

XIII.4.2.4. Constructors

I. constructor

```
Filter1DRamp(float sampledist_v, int length_v , float alpha_v=1, float fc_v=.5);
```

This constructor creates a 1D Ramp filter with a sampling distance, specified length, cut-off frequency, and alpha parameter . The defaults are designed to construct a pure Ramp filter without applying an apodized window such as Hamming filter (i.e alpha=1).

XIII.4.2.5. data access methods

I. void show_params()

This method allows to display all the informations related to the 1D Ramp filter.

XIII.4.2.6. Algorithm

The content of the 1D Ramp filter is as follows:Let assume that a vector $\vec{v}$ in $u^{\perp}$ may be defined by its modulus and an azimuthal angle ω, such that $\vec{v} = |\vec{v}|(\hat{a}\cos\omega + \hat{b}\sin\omega)$. The 1D transaxial ramp filter is then proportional to the magnitude of the transaxial component v_a of $\vec{v}$, i.e. $H_{ramp}(\hat{u},\vec{v}) \propto |\vec{v}|\cos\omega|$. We write the filter function as

$$H_{Ramp}(\vec{u},\vec{v}) = |v / sampling_{radial}| = |v / bin_size| \qquad \text{(Eq. XIII.1)}$$

where the frequency has been normalized to the radial sampling which is in our case the bin size.

When an apodized window is required, a generalized Hamming filter is used where the alpha parameter is aimed to control the smoothness of the transition from 1 to 0 just before the cut-off frequency. The Ramp filter apodized with Hamming window is expressed as follows :

$$H_{ApodizedRamp}(\vec{u},\vec{v}) = |H_{Ramp}(\vec{u},\vec{v})|(\alpha + (1-\alpha)\cos(\pi v / v_a)) \qquad \text{(Eq. XIII.2)}$$

So when $\alpha = 1$, $H_{ApodizedRamp} = H_{Ramp}$

When a Hamming filter is required as an apodized window, the typical value of α are 0.5-0.54.

XIII.5. 2D FILTERS

XIII.5.1. class Filter2D

XIII.5.1.1. Description

This 2D filter class contains three major parameters: the height, the width of the filter and the vector containing the value vector to be applied on data to be filtered

XIII.5.1.2. Location

The class of the 2D filter could be found in *include/Filter.h*

XIII.5.1.3. Class

```
template <class T> class Filter2D
{
protected:
      int height;                            // height of the filter (along Y axis)
      int width;                             // width of the filter (along X-axis)
      Tensor1D <float> filter;               // vector containing the value of the 2D filter
public:
      Filter2D(int height_v, int width_v) :   // defaulltconstructor for Filter2D
            height(height_v), width(width_v),
            filter(1,width_v*height_v){}

      void apply(Tensor1D <T> &ibuf);        // Member function for applying the 2D filter to
                                       1D data array
};
```

XIII.5.1.4. Constructors

I. Default constructor

```
Filter2D(int height_v, int width_v)
```

The default constructor of the 2D filter should have two parameters telling you the size of the filter (i.e the *height* and the *width*).

XIII.5.1.5. Protected members

I. int height

The height of the 2D filter

II. int width;

The width of the 2D filter

III. Tensor1D <T> filter;

The vector containing the values of the 1D array elements of the filter to be applied on data where as the 2D filter has been organized into a 1D vector having a dimension of width*height.

XIII.5.1.6. 5. Data access method

```
void apply(Tensor1D <T> &data);
```

This function allows you to apply the 2D filter on 1D data array. Again, the 2D data array has been organized into a 1D vector of a dimension witdh*height.

XIV. CLASSES FOR PROJECTION OPERATORS

XIV.1. BACKPROJECTION UTILITIES

XIV.1.1. Description:

The section describes the backprojection which is using linear interpolation and a beamwise, incremental method proposed for volume reconstruction of data acquired by a cylindrical, multi-ring positron tomograph. These utilities have been split into two major parts:

- 2D incremental backprojection for backprojecting direct planes (method proposed by Cho et al. (1990));
- 3D incremental, beamwise backprojection (modification of the algorithm proposed by Egger et al. (1998b)).

Compared to the original papers cited above, these methods have been modified in two respects:

- a Jacobian has been added, as detailed in Defrise et al. (1990).
- a new interpolation scheme has been used in the axial *z*-direction.

See below for more details on these modifications.

XIV.1.2. Location

- *recon_buildingblock/backproj2D.cxx*
- *recon_buildingblock/backproj3D.xx*
- *recon_buildingblock/backproj3D_Cho.cxx*
- *include/recon_buildingblock/bckproj.h*

XIV.1.3.4. Principle of incremental backprojection

XIV.1.3.4. Linear interpolation

For every available direction of projection, the centre of every voxel is projected onto the projection plane, and the update value V_A of that voxel A is obtained from bilinear interpolation between the nearest four measured samples, V_1, V_2, V_3 and V_4 :

$V_A = (1 - ds_A)\,(1 - dz_A)\,V_1 + ds_A\,(1 - dz_A)\,V_2 + (1 - ds_A)\,dz_A\,V_3 + ds_A\,dz_A\,V_4,$ (Eq. XIV.4)

where ds_A and dz_A are the transaxial and axial interpolation distances respectively, as shown on figure X.2, and the voxel size is normalised to unity. The voxel value a_i is obtained by summing the update values V_A over the projections at all available angles.

Backprojection into a particular beam may now be performed easily and is thoroughly described in [Egger et al., (1998b)]. Let us first introduce the beam constants K_1, K_2 and K_3 as follows:

$V_A = V_1 + ds_A\,K_1 + dz_A\,K_2 + dz_A\,ds_A\,K_3,$ (Eq. XIV.5)

with $K_1 = V_2 - V_1$, $K_2 = V_3 - V_1$ and $K_3 = V_4 - V_2 - V_3 + V_1$.

A similar relation holds for a voxel B adjacent to A and the centre of which lies inside the same beam:

$V_B = V_{B,incr} + dz_B\,ds_B\,K_3,$ (Eq. XIV.6)

By substituting these into equation X.1, we have

$V_{B,incr} = V_{A,incr} + (ds_B - ds_A)\,K_1 + (dz_B - dz_A)\,K_2.$ (Eq. XIV.7)

Since using the above symmetry considerations it is sufficient to consider the cases with $0 \le \phi < \pi/4$ and $\theta \ge 0$, the values of $ds_B - ds_A$ and $dz_B - dz_A$ for a particular beam may only take four values each, depending on the position of voxel B with respect to voxel A. Let

$\delta s_x = ds_B - ds_A$ and $\delta z_x = dz_B - dz_A$ if B is reached from A by a displacement of –1 along the x-axis;

$\delta s_y = ds_B - ds_A$ and $\delta z_y = dz_B - dz_A$ if B is reached after a displacement of +1 along the y-axis;

$\delta s_d = ds_B - ds_A$ and $\delta z_d = dz_B - dz_A$ if B is reached diagonally in the transaxial plane by a –1 displacement along x and a +1 displacement along y;

$\delta s_z = ds_B - ds_A$ and $\delta z_z = dz_B - dz_A$, for a displacement of +1 along the z-axis.

Let $\delta s_{x,y,d,z}$ and $\delta z_{x,y,d,z}$ represent the groups $\{\delta s_x, \delta s_y, \delta s_d, \delta s_z\}$ and $\{\delta z_x, \delta z_y, \delta z_d, \delta z_z\}$ respectively. These beam invariants $\delta s_{x,y,d,z}$ and $\delta z_{x,y,d,z}$ are functions of the angles ϕ and θ and.

First, the beam invariants K_1, K_2 and K_3 are computed; they depend on the values of the four projection elements used to define the beam. Then the values of $\delta s_{x,y,d,z}$ and $\delta z_{x,y,d,z}$ are determined from the values of ϕ and θ. Moreover, all possible alternatives for the second and third terms on the right hand-side of Eq. XIV.5 may be calculated once for all voxels in the beam.

The update value of the first voxel in the beam must be calculated from Eq. XIV.4 and serves as the starting point for that particular beam. Once a voxel A is reached, the next voxel in the beam – voxel B – is determined by a searching flow algorithm (Eq. X.4) is used to obtain its update value V_B: $V_{B,incr}$ *by additions only*, by choosing the appropriate pre-calculated values depending on the direction of the displacement from A to B. The new values of dz_B and ds_B are obtained by adding the appropriate $\delta z_{x,y,d,z}$ and $\delta s_{x,y,d,z}$ to the current dz_A and ds_A respectively. The new update value V_B is then simply obtained from Eq. XIV.6. This process is repeated until the last voxel in the beam is reached.

This algorithm is only partly incremental, in the sense that only the first-order term $V_{B,incr}$ can be found by additions only. Two multiplications remain necessary to incorporate the second-order term depending on $ds_B dz_B$. A method to avoid this has been proposed by Cho [Cho90] in a spherical geometry. A fully incremental 3D algorithm presents an unavoidable computational overhead however, which more than compensates the gain due to the elimination of the multiplications. This algorithm still needs many fewer multiplications than standard voxel-driven bi-linear interpolation.

XIV.1.3.4. Piecewise linear interpolation

When the z-spacing between corners of the beam is twice as large as thez-voxel size, we should use the piece-wise interpolation as given in Fig.XIV.3. Instead of Eq. X.1, theupdate value V'_A of the voxel A is now obtained as follows:

$V'_A = (1 - ds_A)(3/2 - dz_A) V_1 + ds_A (3/2 - dz_A) V_2 + (1 - ds_A)(dz_A - 1/2) V_3 + ds_A (dz_A - 1/2) V_4$, if $1/2 < dz_A < 3/2$ (Eq. X.5a)

$V'_A = (1 - ds_A) V_1 + ds_A V_2$, if $dz_A <= 1/2$ (Eq. X.5b)

$V'_A = (1 - ds_A) V_3 + ds_A V_4$, if $dz_A >= 3/2$ (Eq. X.5c)

The two last equations can easily be incrementalised. We concentratehere on Eq. X.5a, and in the following we assume that $1/2 < dz_A < 3/2$. We can write Eq. X.5a in terms of the samebeam-constants as defined below Eq. X.2:

$V'_A = V_1 - K_2/2 + ds_A (K_1 - K_3/2) + dz_A K_2 + ds_A dz_A K_3$ (Eq. X.6)

For a voxel B adjacent to A and whose centre liesinside the same beam, we find:

$$V'_B = V'_{B,incr} + dz_B\, ds_B\, K_3, \qquad \text{(Eq. X.7)}$$

with

$$V'_{B,incr} = V'_{A,incr} + (ds_B - ds_A)(K_1 - K_3/2) + (dz_B - dz_A) K_2. \qquad \text{(Eq. X.8)}$$

We can conclude that the implementation follows the same lines as forlinear interpolation. The main change is that incremental values for Eq.X.5b,c have to be stored, and that at each update of a voxel, a test has tobe made on the value of dz_A. It turns out that the increase in CPU time for theincremental versions of piece-wise linear over ordinary linearinterpolation is only a few percent. This can be explained by the fact thatin half of the cases for dz_A , Eq. X.7 is not used, but a fully incremental version of Eq. X.5b,c.

XIV.1.3.5. Searching flow algorithm

The searching flow algorithm is used to locate the voxels inside a beam, starting from the first voxel. Due to the beam's known axial extension of two voxels, it may be restricted to finding only one half of the actual number of voxels inside the beam (Fig. XIV.6). Since displacements along z do not affect ds, the algorithm may easily be separated into a transaxial part, searching in the (x,y)-plane, and an axial part, searching along z. However, this not true for axially compressed data as the beam’s size is equal to the voxel size (see figure XIV-6).

The in-plane searching method proposed by He [He93] is applied; under the condition *pixel_size= bin_size* the direction of the displacement towards the next voxel in the (x,y)-plane is uniquely determined by the current value of ds:

if $ds \geq \cos\phi$,	move –1 along x
if $ds < 1 - \sin\phi$,	move +1 along y
if $1 - \sin\phi \leq ds < \cos\phi$,	move –1 along x and +1 along y.

For a voxel to belong to the beam, dz must be comprised between 0 and 1/2. For positive θ the following simple test thus applied to control displacement along z:

if $dz < 0$,	move one voxel along z,
else	stay in same transaxial slice.

The values of ds and dz are updated at every new position, and searching is continued until the last voxel in the beam is reached.

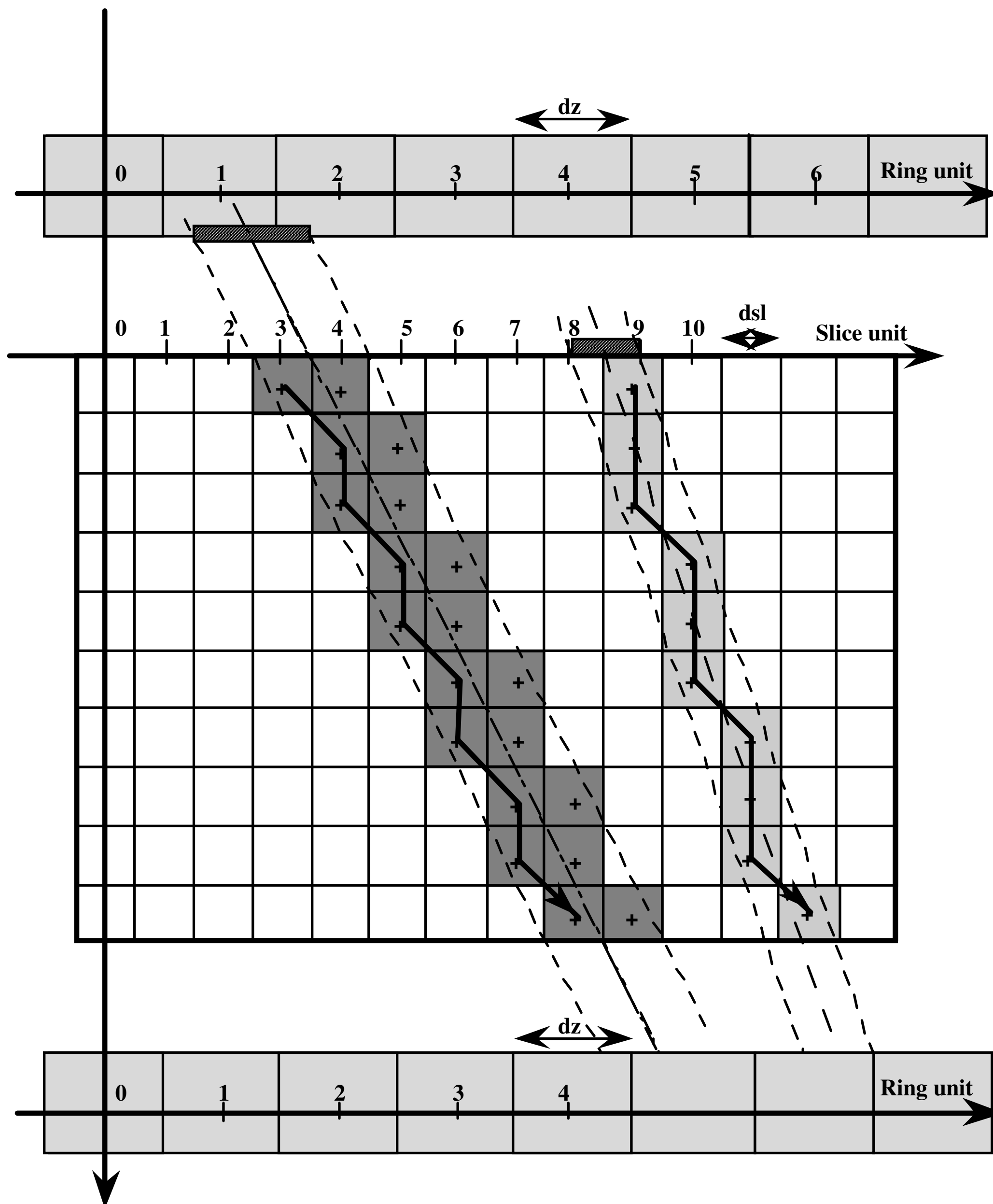


Figure XIV.6 : *The searching flow algorithm is used to locate the voxels inside a beam, starting from the first voxel. Two cases are shown depending on the axial compression : (a) dark gray (left beam), case for data with span =1, (b) light gray (right beem), case for dat with span > 1.*

XIV.1.3.6. Use of geometrical symmetries

In the Cho algorithm, considerations of symmetry properties of the image volume may be exploited to reuse all geometrical quantities related to a particular voxel, such as the interpolation distances *ds* and *dz*, for all voxels in equivalent geometrical environments, without having to calculate them explicitly. Explicit calculation is only needed for voxels within beams with $n \geq 0$, $0 \leq \phi < \pi/4$, $\theta \geq 0$ and $r_0 = 0$. More details of the relationship between the values V_1, V_2, V_3 and V_4 of Eq. XIV.6, the distances *ds* and *dz*, and the values of the axial and radial co-ordinates are illustrated in Egger et al. (1998b):

• ***s-symmetry:*** Symmetry about the centre of the image volume is exploited to facilitate backprojection in beams with $s < 0$.;

• ***ϕ-symmetry :*** The information on *ds* and *dz* can also be shared with beams at azimuthal angles $\pi/2 - \phi$, $\pi/2 + \phi$ and $\pi - \phi$, (respectively, min90, plus90 and min180 in the source codes);

• ***θ-symmetry :*** The quantities *ds* and *dz* may also be used when backprojecting beams with negative *delta;*

• ***z–symmetry :*** The translational symmetry along the *z*–axis is the most obvious of all cases where the quantities *ds* and *dz* may be reused for voxels they were not initially calculated for.

Self-symmetric beams, *i.e.* beams that are their own image by one of the above symmetry relations, require a separate treatment to avoid double processing. This is the case with beams at $\phi = 0$, $\pi/4$, $\pi/2$ and $3\pi/4$. Beams located between $s = 0$ and the transaxial sampling distance *bin_size* contain both self-symmetric voxels, located in the immediate vicinity of $s=0$, and symmetric voxel pairs located closer to s = *bin_size* and –*bin_size* respectively. Care must be taken to correctly process the various cases.

XIV.1.3.1. Backprojection using linear interpolation

The contribution of one voxel to a particular projection element and vice versa must therefore be calculated on the fly, and this is the most compute intensive part of the entire reconstruction process. Various methods exist which make different types of approximations and yield slightly different results. Two main categories may be distinguished: methods based on bilinear interpolation on one hand, and ray-driven approaches on the other hand.

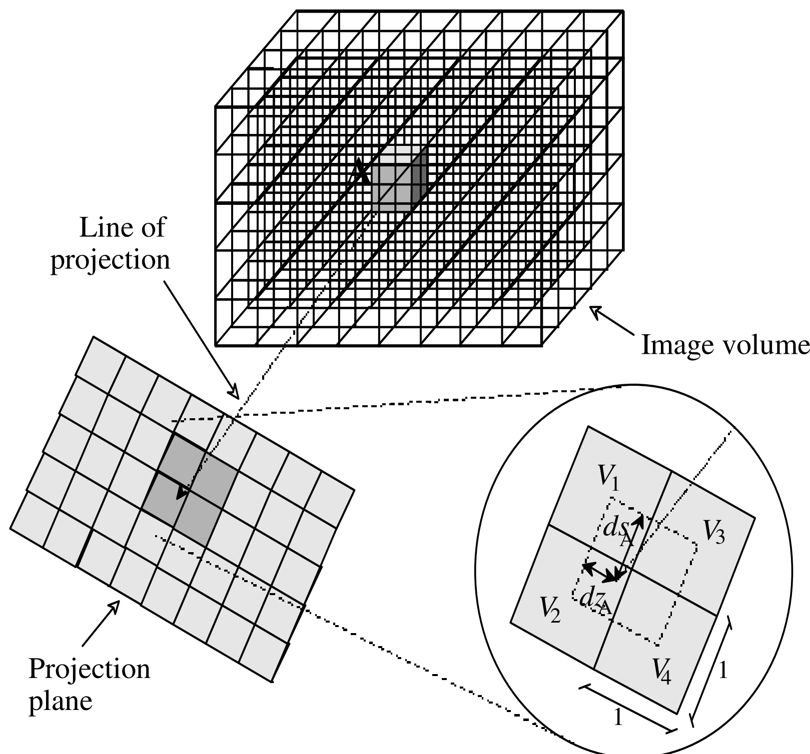


Figure XIV.1: *Forward- and backprojection based on bilinear interpolation. ds and dy are the transaxial and axial displacements. The centre of every voxel (here voxel A) is projected onto the projection plane, and is fitted to the discrete grid of existing projection elements by bilinear interpolation between the nearest four samples.*

In the former the centre of every voxel is projected onto the projection plane, and bilinear interpolation between the four nearest projection elements determines their respective weights. An illustration is given in Fig. IX.1. A variation of the interpolation method is the stretching method [Pet81], consisting of a preliminary multiplication of smaller, subdivided, projection elements by linearly interpolating between known values; fast nearest-neighbour approximations on these artificially oversampled projection arrays are then used to find the appropriate contribution to a given image element. The main disadvantage of this method, apart from being slightly less accurate than true interpolation algorithms, is the large amount of memory necessary to hold the fine projection grid.

In a ray-tracing approach the problem is taken from the other end. For a given detector pair, that is, sinogram element, the length of intersection of the corresponding LOR with a voxel in the image volume determines the weight this voxel carries in the projection element. See figure XIV.8 in the forward-projection section . More accurate strip integrals, covering the entire detector width, are usually used in 2D, but are too computationally intensive for use in 3D.

Both methods exhibit artefacts caused by the accumulation of inaccuracies at certain angles between the square (cubic) image grid and the direction of projection, in particular at 45 degrees. Fortunately this occurs in the forward-projection process for methods based on bilinear interpolation, whereas ray-driven approaches suffer from this type of error during backprojection, as is shown in two dimensions in Fig. XIV.2.

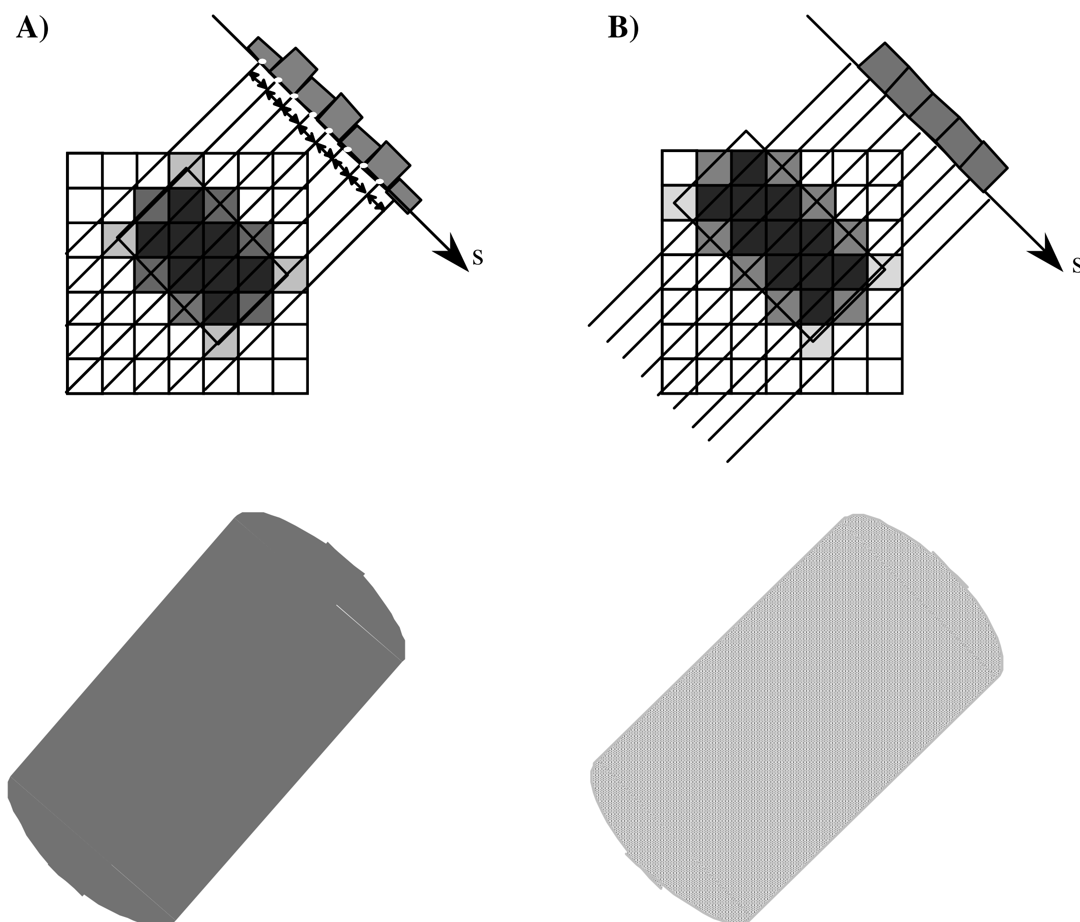


Fig. XIV.2: *Error accumulation during projection and backprojection of a rectangular object of constant intensity, at an angle of 45 degrees between the square image grid and the direction of projection: (top left) projection with method based on bilinear interpolation, (top right) projection with ray-driven method, (bottom left) images obtained from backprojection with the bilinear interpolation method and (bottom right) with the ray-driven method.*

The immediate conclusion is that a ray-driven method must be used for forward-projection, whereas a bilinear interpolation method is more suitable for backprojection. Their efficient implementation using geometric symmetries is given in detail in section XIV.1.3.6, for the forward-projection algorithm (similar to the technique proposed by Siddon [Sid85]) and now for the backprojection (partly incremental, developed from the original ideas of Cho [Cho90]).

XIV.1.3.2. Interpolation in the axial direction

For efficiency reasons, voxel sizes in the *x* and *y* direction are taken equal to the size of an (arc-corrected) bin. However, doing the same in the *z*-direction (*i.e.* taking *z* voxel size equal to the ring spacing of the scanner) results with most PET scanners in bad resolution in the *z*-direction. In the PARAPET projectors, the *z*-voxel size is assumed to be equal to half of the ring spacing (Figure XIII.3). This corresponds to the usual practice in 2D PET to have 'direct' and 'indirect' planes. When data are not compressed axially (see section VIII.1), this means that the sampling in *z* direction is twice as fast for the voxels than for the projection data. In this section, we briefly show that simple linear interpolation is not appropriate in this case.

We can view the action of an interpolating backprojector as the convolution of the data with an interpolating kernel. For example, interpolation in the 1-dimensional case can be done asfollows: for discrete data *fi* and a kernel *K(x)*:

fint(x) = \Sum_over_i fi K(i-x)

For linear interpolation in 1D, this kernel is given by a triangle function:

K(x) = 1-x, if $0>x>1$

K(x) = x-1, if $-1<x<0$

K(x) = 0, otherwise

In the present case, the interpolation kernel is two-dimensional (with coordinates *bin* and *ring*), we assume it is just aproduct of two one-dimensional kernels. Furthermore, we will take the same kernel for any *segment* and *view*. The contribution of 1 data element i,*j* is simply the shifted kernel *K(i-bin,j-ring)*.

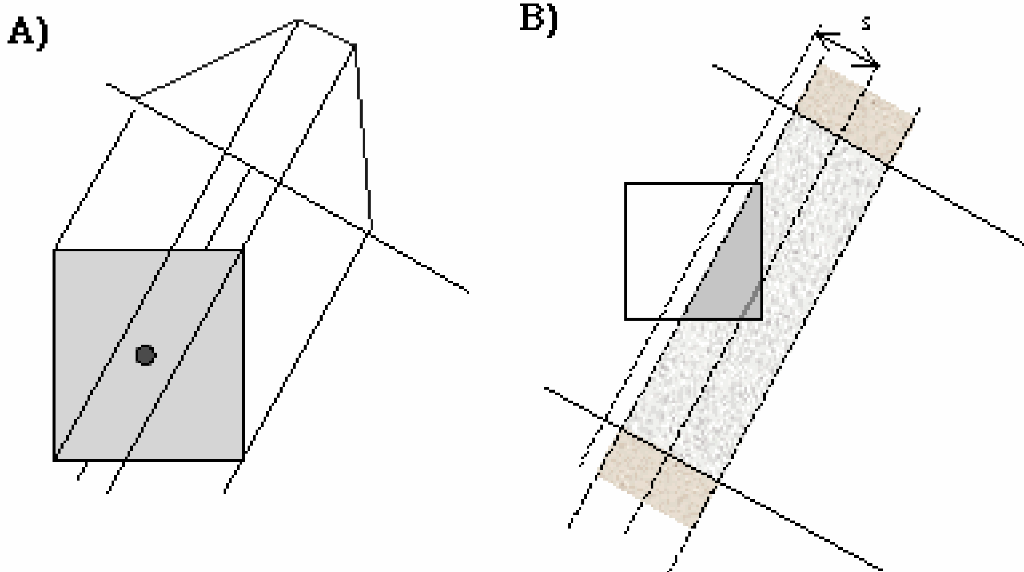


Figure XIV.3: *Illustration of two ways to approximate the probability of detection of an event in a voxel for the 2D case. The intersection length of the central LOR between the two detectors with the voxel, and the intersection area of the TOR with the voxel.*

We recall the connection between backprojection and the probability of detection mentioned in the previous section. The contribution of the backprojection of one bin to a particular voxel should approximate the probability that an event in that voxel is detected in that bin. As already mentioned, using the intersection length of the LOR with the voxel for this probability gives bad results when backprojecting. A much better approximation is the volume of intersection between the Tube-Of-Response (TOR) and the voxel (see Fig. XIV.3).

Fig.XIV.4 shows the area of intersection in the two-dimensional case, for different relative locations of the detectors and the voxel. The graph on the left shows that a triangle shape (i.e. linear interpolation) is a reasonable approximation for the area ofintersection for *any* angle, but *only* in the case where voxel sizes are equal to the detector sizes. Similarly, linear interpolation is only slightly worse than 'volume of intersection', but *only* in the case where voxel sizes are equal to the detector sizes.

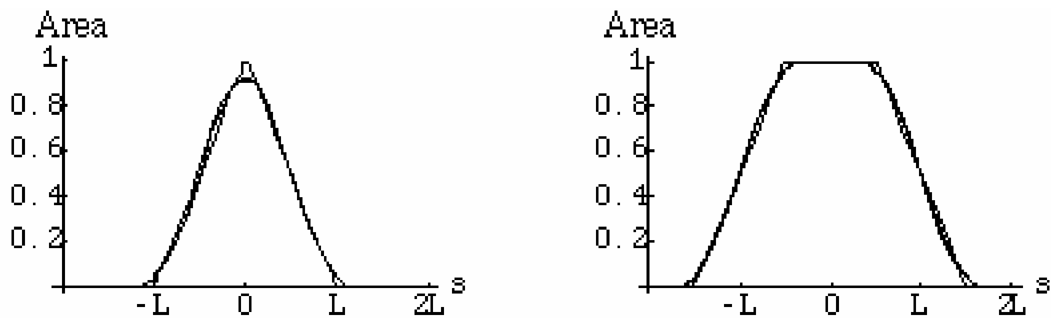


Figure XIV.4: *Plots of the area of intersection A of the TOR and a pixel of size LxL. See Fig. XIV.3 for the definitions. Each plot contains curves for 2 different angles of the TOR: 0 and π/4. Left:detector size = L. Right: detector size = 2*L.*

Fig. XIV.5 shows the interpolating kernels that we use in the axial direction. Obviously, in the case where the detector size is twice the voxel size, the piece-wise linear interpolation corresponds much better to the probability. Moreover, it will improve resolution in the *z*-direction.

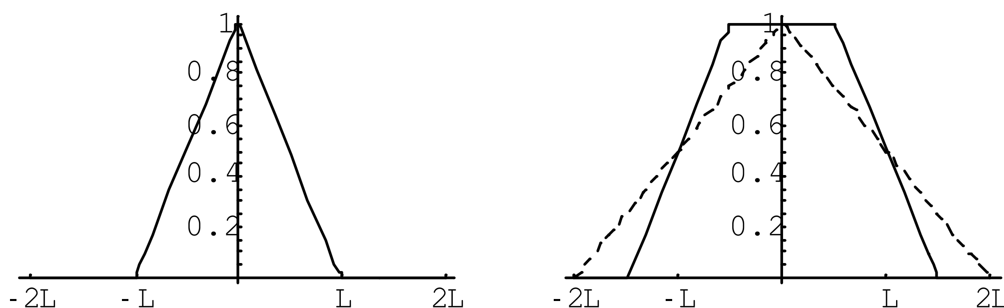


Figure XIV.5: *Interpolating kernels: linearinterpolation (dashed), building block interpolation. Left: detector size =L (the two curves are identical). Right: detector size = 2*L.*

An alternative way to understand the piece-wise interpolation is to estimate the detection probability as the sum of the probabilities of 2 pairs of detectors of half the size.

XIV.1.4 Algorithm

Sweeping the image volume beam by beam, where abeam is the volume delimited by the central rays of four adjacent projection elements, allows to make use of a number of beam constants. In addition, use is made of geometrical symmetries of the image volume. Combining this with the exploitation of the geometrical symmetries of the image volume results in a very fast implementation of the backprojection operator, about twelve times faster than a straightforward implementation of the same equations.

XIV.1.3.3. Inclusion of a Jacobian factor

Both projectors work in a coordinate system as detailed in section IV.4, but assuming that arc-correction on the bin-coordinate has been performed. In this coordinate system a geometrical factor has to be included corresponding to the size of a detector-tube (this geometrical factor was ignored in Egger et al. (1998b)). It is easiest to derive this factor by looking at backprojection of continuous data.

Projection space can be described by two vectors: a unit vector $\hat{n}$ giving the direction of the projection, and a vector $\vec{d}$ orthogonal to $\hat{n}$, giving the location of the intersection of the LOR with a projection plane (orthogonal to $\hat{n}$).

The projection of a point $\vec{r}$ on the projection plane is given by $r\vec{r} - (\vec{r} \cdot \hat{n})\hat{n}$. So, the backprojection of projection data $p(\hat{n}, \vec{d})$ contributing to $\vec{r}$ is given symbolically by

$$\int p(\hat{n}, \vec{d})\, \delta(\vec{r} - (\vec{r} \cdot \hat{n})\hat{n} - \vec{d})\, d\hat{n}\, d\vec{d} \qquad \text{(Eq. XIV.1)}$$

(using the 2-dimensional delta function on the projection plane), which can be simplified to

$$\int p(\hat{n}, \vec{r} - (\vec{r} \cdot \hat{n})\hat{n})\, d\hat{n} \qquad \text{(Eq. XIV.2)}$$

If the unit vector $\hat{n}$ is given in terms of a polar angle and an axial angle. Integration over $d\hat{n}$ is equivalent to integration over $d\phi d\theta \cos\theta$. However, the data on which the backprojection works are organised in a different coordinate system where there is no immediate correspondence with projection planes. The above formulas have to be written in terms of this new coordinate system. This involves a change of coordinates in the integral, and hence a Jacobian. It turns out that in the end, the cosinus factor drops out, and only the Jacobian remains. The value of the Jacobian is

$$\frac{4(R^2 - s^2)}{(4R^2 - 4s^2 + \delta^2)^{3/2}} \qquad \text{(Eq. XIV.3)}$$

where R is the scanner radius, s is the bin-coordinate and δ is the ring-difference of the LOR.

Interpolating backprojection can be seen as a discrete implementation of the backprojection operation (replacing the integrals by summation). This means that after doing the linear interpolation for a beam, the result has to be multiplied by the value average value of this Jacobian for that beam (see also Defrise et al. (1990)). Note that the Jacobian suppresses contributions from higher ring differences.

In implementations of analytic algorithms the s dependence of the Jacobian is usually ignored. This can only safely be carried out if s is much smaller than R for all bins in the FOV of the scanner. In our current implementation the full Jacobian is used when the preprocessor variable JAC is non-zero, otherwise the approximation is used.

For probabilistic algorithms, the backprojection operation is a matrix multiplication of the (transpose of) the probability matrix with the projection data, where the elements of the probability matrix are defined as the probability of detection of an event in a voxel by a detector-pair. The backprojection discussed here corresponds to the geometric part of the probability matrix. If the detectors would surround the whole object, the sum of these geometric probabilities over all

detector-pairs has to be 1. This corresponds to performing the backprojection integral with all projection data equal to 1, which gives the result 2. If the Jacobian is ignored, this integral does not converge, and a proper normalisation is impossible.
Summarising: the backprojector computes a discretised approximation to the continuous backprojection integral, and divides the result by 2.

XIV.1.4. 2D backprojection

XIV.1.4.1. Description

This contains the utilities for the backprojection of 2D direct sinograms.

XIV.1.4.2. Location

- *recon_buildingblock/backproj2D.cxx*
- *include/recon_buildingblock/bckproj.h*

XIV.1.4.3. Functions

Several data access methods have been written in order to backprojection into either a PET plane or image volume. The backprojection is operated for each view where the view parameter goes from 0 till view45 - 1. Also this supposes that bin_size = pixel_size. We now list the methods in "down-to-top" order.

```
void Backprojection_2D(const PETSinogram &Sino, PETPlane & Image2D, int view)
```

This method contains the kernel of the 2D backprojection algorithm and is the "lowest" order of the whole 2D backprojection methods.

```
void Backprojection_2D(const PETSegment &sinos, PETImageOfVolume &image, int view)
```

This method which is a subset of the previous method, implements a 2D backprojection on the stack of 2D data from a segment into the image volume for a specific view where

```
void Backprojection_2D(const PETSinogram &Sino, PETPlane & Image2D)
```

This method is similar to those above except it works with a specific sinogram (PETSinogram) instead of PETSegment.

```
void Backprojection_2D(const PETSegment &sinos, PETImageOfVolume &image)
```

This method implements a 2D backprojection on the stack of 2D sinogram data from a segment into the image volume.

```
void Backprojection_2D_flick(const PETSinogram &Sino, PETPlane &image, int view);        void
Backprojection_2D_flick(const PETSinogram &Sino, PETPlane &image)
```

The two previous functions are used for FORE algorithm as 2D backprojection is somewhat slightly different than those described here (see more details in D4.3).

XIV.1.4.4. Algorithm

Backprojection_2D function is used to backproject direct sinograms and used the incremental, beamwise method as described in Cho *et al.* [Cho, 1990] and it is a particular case of the incremental, beamwise, 3D backprojection procedure as only direct sinograms (delta = 0 or ±1) are processed. Details of the algorithm is described earlier.

XIV.1.4.5. Software implementation

Some prerequisted conditions in our implementation are :

```
• assert(sinos.min_ring() == image.get_min_z())
```

The first index of sinogram along ring axis must be equal to the first index of the image along x-axis.

```
• assert(sinos.max_ring() == image.get_max_z())
```

The last index of sinogram along ring axis must be equal to the last index of the image along x-axis.

```
• assert(sinos.get_average_ring_difference() ==0);
```

2D backprojection can be run on 2D direct projection data only i.e sinograms having an average of ring index difference equal to 0.

```
• assert(sinos.scan_info.get_bin_size() == image.get_voxel_size().x);
```

The bin size of the direct sinograms must equal to the image pixel size.

XIV.1.5. 3D backprojection

XIV.1.5.1. Description

There is a whole sequence of 3D backprojectors based on work by [Egger et al., 1997], working on segments or viewgrams. The interface is slightly complicated because projection data related by symmetry have to be passed in one function call.

XIV.1.5.2. Prerequisited conditions

3D backprojection prerequesites some conditions in order to be implemented and all these functions currently assume that (in pseudo-code)

• The number of bin elements must be equal to the size of the image along x-axis and must be an odd value i.e

```
get_num_bins() ==image.get_x_size()
get_num_bins() odd
```

• The origin of the image must be located at the middle of the image i.e

```
image.get_min_x() == -image.get_max_x()
```

• The image pixels has to have the same size as the physical sampling size in the middle of the scanner i.e

```
image.get_voxel_size().x == scan_info.bin_size
```

• The dimension of image along the scanner direction (i.e number of image pixels * physical sampling size) must not exceed the axial dimension of the scanner i.e

```
image.get_voxel_size().z * (image.get_z_size() + 1) == pos_view.scan_info.FOV_axial
```

with

```
max FOV diameter = physical sampling size * number of measured sinogram bins.
```

• The first plane of the image must be 0

```
image.get_min_z() == 0
```

Besides, 3D backprojection does a loop over all rings. However, because we use interpolation of a 'beam', each step takes elements from ring0 and ring0+1. So, data in a ring influences beam ring0–1 and ring0. All this means that we have to let ring0 run from rmin–1 to rmax. Although the maximum possible ring index diffrence in a scanner with n rings is obviously n–1, no beam may be defined with one sinogram only, and the maximum ring index difference allowed by this backprojection is n–2.

XIV.1.5.3. Location

- *recon_buildingblock/backproj3D.cxx*
- *include/recon_buildingblock/backproj.h*

XIV.1.5.4. Functions

Due to different form of projection data, several methods can be accessed through either by-segment routines or by-view routines.

I. By-segment routines

First some functions which work on a whole PETSegment. The routines use symmetry, so they require 2 PETSegments, where *segment_pos* has positive *average_ring_difference* = *delta*, while *segment_neg* has *average_ring_difference* = –*delta*. The function which takes a view argument where 0 <= *view* <= *num_views*/4 (= 45˚). They process 4 symmetry related views (2 for 0˚ and 45˚).

Functions are listed on 'top-down', that is, the higher ones call the lower ones.

```
void back_project(   const PETSegment& segment_pos,
                     const PETSegment& segment_neg,
                     PETImageOfVolume& image);

void back_project(   const PETSegment & segment_pos,
                     const PETSegment & segment_neg,
                     PETImageOfVolume & Image, const int view);
```

The following methods are similar to those above above except functions which take minimum and maximum ring index (*rmin*, *rmax*) arguments.

```
void back_project(   const PETSegment & segment_pos,
                     const PETSegment & segment_neg,
                     PETImageOfVolume & Image,
                     const int rmin, const int rmax);

void back_project(   const PETSegment & segment_pos,
                     const PETSegment & segment_neg,
                     PETImageOfVolume & Image,
                     const int view,
                     const int rmin, const int rmax);
```

The last method represents the "lowest method" for all 3D backprojection methods using by-segment routines. It is essentially a loop over rings, calling CHO routines discussed in section XIV.1.6.

The arguments are as follows

```
      const PETSegment & segment_pos,     // (in) segment with positive delta
      const PETSegment & segment_neg,     // (in) segment with negative delta
      PETImageOfVolume & Image,           // (out) returns the backprojected data into the image
                                          matrix
      const int view                      // (in) view number to be processed
      const int rmin                      // (in) lower ring number
      const int rmax                      // (in) upper ring number
```

II. By-view routines

1.5.4.II.1. On 8 PETViewgrams

The following functions backproject 8 viewgrams related by symmetry.

• Without minimum and maximum rings

```
void back_project(
            const PETViewgram & pos_view,
            const PETViewgram & neg_view,
            const PETViewgram & pos_plus90,
            const PETViewgram & neg_plus90,
            const PETViewgram & pos_min180,
            const PETViewgram & neg_min180,
            const PETViewgram & pos_min90,
            const PETViewgram & neg_min90,
            PETImageOfVolume & image);
```

• With minimum and maximum rings

```
void back_project(
            const PETViewgram & pos_view,
            const PETViewgram & neg_view,
            const PETViewgram & pos_plus90,
            const PETViewgram & neg_plus90,
            const PETViewgram & pos_min180,
            const PETViewgram & neg_min180,
            const PETViewgram & pos_min90,
            const PETViewgram & neg_min90,
            PETImageOfVolume & image,
            const int rmin, const int rmax);
```

Argument names encode this as follows:

pos : positive average ring difference delta

neg : average ring difference -delta

plus90 : view+90 degrees

min180 : 180 degrees –view

min90 : 90 degrees – view

And, the arguments are :

```
PETImageOfVolume &image                  // (in) the current image matrix obtained after
            processing sinograms with a special ring difference              (from 0 to max_delta)
PETViewgram &pos_view, &pos_plus90,
             &pos_min180, &pos_min90       // (out) viewgram with positive delta;
PETViewgram &neg_view, &neg_plus90,
             &neg_min180, &neg_min90       // (out) viewgram with negative delta;
const int rmin                           // (in) lower ring number
const int rmax                           // (in) upper ring number
```

with argument names which encode this as follows

pos : positive average ring difference delta

neg : average ring difference – delta

plus90 : view+90 degrees

min180 : 180 degrees – view

min90 : 90 degrees – view

1.5.4.II.2. On 4 PETViewgrams

The following functions backproject 4 viewgrams related by symmetry. They should be used for *view*=0 or 45 degrees. Calling the 8 viewgram version for these special views internally reroutes to the functions below (so you are wasting memory by using the 8 argument version). This function could also be used for other views, but are then less efficient than the 8 viewgrams versions (it could be useful for example in one of the iterative method such as OSEM, for larger number of subsets; In such a casse, view should be over the range $0 \le view < num_views/2$ (=90˚).

```
° Without minimum and maximum rings
void back_project(   const PETViewgram & pos_view,
                     const PETViewgram & neg_view,
                     const PETViewgram & pos_plus90,
                     const PETViewgram & neg_plus90,
                     PETImageOfVolume & image);

• With minimum and maximum rings
void back_project(   const PETViewgram & pos_view,
                     const PETViewgram & neg_view,
                     const PETViewgram & pos_plus90,
                     onst PETViewgram & neg_plus90,
                     PETImageOfVolume & image,
                     const int rmin, const int rmax);
```

The arguments are already described in the previous section (see on 8 viewgrams).

XIV.1.5.5. Software implementation

XIV.1.6. CHO backprojection

XIV.1.6.1. Description

This section describes the internal functions used by back_project. They implement the algorithm discussed in section XIV.1.3. They should not be used anywhere else.

They use the following symmetries:

- opposite ring difference
- views : view, view+90 degrees, 180 degrees – view, 90 degrees – view
- s,–s symmetry (while being careful when s==0 to avoid self-symmetric cases)

XIV.1.6.2. Location

The CHO functions are located in

- *recon_buildingblock/backproj3D_Cho.cxx*
- *include/recon_buildingblock/backproj.h*

XIV.1.6.3. Functions

The two internal CHO functions are as follows.

```
• void backproj3D_Cho_view_viewplus90_180minview_90minview(
        PETImageOfVolume & image,
```

```
        Tensor4D < float > &Projptr,
        const PETScanInfo &scan_info,
        float delta,
        float cphi, float sphi, int s, int ring0)
```

This method backprojects 8 beams related by symmetry views

```
• void backproj3D_Cho_view_viewplus90(
        PETImageOfVolume & image,
        Tensor4D < float > &Projptr,
        const PETScanInfo &scan_info,
        float delta,
        float cphi, float sphi, int s, int ring0);
```

This method backprojects 4 beams related by symmetry

The arguments are :

```
        PETImageOfVolume & image        // (out) returns the projection data into the image matrix
        Tensor4D < float > &Projptr     // 4D tensor which be detailed later on
        float delta                     // the delta value or (the average ring index
        difference)
        float cphi                      // Value of cos(phi)
        float sphi                      // Value of in(phi)
        int s                           // bin element number (transaxial displacement)
        int ring0                       // smallest ring index of the data fixed by the
        axial position of the beam
```

```
• static void find_start_values(
      const PETScanInfo &scan_info,          // Pointer to scanner information
      const float delta,                     // Ring index difference (or delta)
      const float cphi, const float sphi,    // Values repectively of cos(phi) and sin(phi)
      const int s,                           // Bin element number to be processed
      const int ring0,                       // Ring number to be processed
      const int fovrad,                      // Half the image size (in pixels)
      const double d_sl,                     // Sampling distance (in mm)
      int&X1, int&Y1, int& Z1,               // Initial starting voxel values
      double& ds, double& dz,                // Transaxial and axial interpolation distances
      respectively, as shown on Fig. 1
      double& dzhor, double& dzvert)         // Incremental quantities
```

The initialisation code (common to all symmetry cases) which finds the first voxel in the beam and the related values for ds and dz, incremental parameters described in X.3.

XIV.1.6.4. Software implementation

The function *back_project* works with an array of **Tensor4D** < float > Proj2424(0, 1, 0, 3, 0, 1, 1, 4) representing the four parameters [*sign_delta*], [*view_case*], [*sign_bin*], [*V*]

where

- *sign_delta* : has 2 values corresponding to the sign of the average ring index difference delta :

  - *sign_delta* = 0 => *delta* > 0, Positive ring difference
  - *sign_delta* = 1 => *delta* < 0, Negative ring difference

− *view_case* represents the azimuthal angle and has four values corresponding to

  - *view_case* = *1* => $0 < \phi < \pi/4$
  - *view_case* = *2* => $\pi/2 - \phi$

- $view_case = 3 \Rightarrow \pi/2 + \phi$
- $view_case = 4 \Rightarrow \pi - \phi$

\- s represents the bin element position regarding to the origin of sinograms (i.e the middle bin size) and has two values:

- $sign_bin = 0 \Rightarrow s > 0$
- $sign_bin = 1 \Rightarrow s < 0$

\- V represents the values of the 4 elements defining the edges of the beam

- V_1: (r_0, t),
- V_2: $(r_0, t + 1)$
- V_3: $(r_0 + 1, t,)$
- V_4: $(r_0 + 1, t + 1)$

with

r_0 or $r_0 + 1$, the smallest ring index of the data fixed by the axial position of the beam ($ring0 = r_O$ and $r0plus = r_O + 1$, in the source codes)

$\pm t$, $\pm(t+1)$, the transaxial displacement along bins direction ($s = t$, $splus = t + 1$, $ms = -t$; $msplus = -t - 1$, in the source codes)

Tables of the annexes 2 and 3 give an overview of all voxels directly related to one explicit calculation of *ds* and *dz* by symmetries in s, ϕ and θ. Voxels shifted along the z-axis by the z-symmetry are not shown. The first two columns indicate the sign of s and θ respectively. The last column gives for each parameter V_i the corresponding ring number (r_0 or $r_0 + 1$) and projection element number ($\pm s$ or $\pm(s+1)$).

XIV.1.7. Class for the Jacobian

XIV.1.7.1. Description

This class is used in the backprojection to take the Jacobian into account. It also includes a global normalisation factor of $1/2\pi$.

If the preprocessor symbol JAC is #defined to a non-zero number, theexact formula for the Jacobian is used, otherwise the approximation the bincoordinate is set to 0 is used.

XIV.1.7.2. Location

-include/recon_buildblock/bckproj.h

XIV.1.7.3. Definition

```
class Jacobian
{
public:
explicit Jacobian(const PETScanInfo& scan_info);
float operator()(const float delta, const float s) const;
};
```

XIV.1.7.4. Constructors

explicit Jacobian(const PETScanInfo& scan_info)

This stores various geometric factors related tothe scanner into some private members.

XIV.1.7.5. Data access members

float operator()(const float delta, const float s)const

This methodcomputes the Jacobian for a given ring difference delta and bin number *s*.

XIV.1.7.6. Usage

Typicalusage is as follows:

```
const Jacobian jacobian(segment.scan_info));
jacobian(segment.get_average_delta(), s+ 0.5);
```

XIV.2. FORWARD PROJECTION UTILITIES

XIV.2.1. 3D Forward projection

XIV.2.1.1. Description

The 3D forward projection function allows to estimate projection data given an image.

Every projection element p_j is related to the image voxel values a_i by the relation

$$p_j = \sum_i c_{ij} a_i \quad , \qquad \text{(Eq. XIV.8)}$$

where c_{ij} is the probability of detecting an annihilation inside voxel i within detector tube j. For a given detector tube j, that is, a given sinogram element, the length of intersection of the corresponding line of response (LOR) with a voxel in the image volume determines the weight c_{ij} this voxel carries in the projection element. This principle is illustrated in two dimensions (2D) in Fig. XIV.7. More accurate strip integrals, covering the entire detector width, are often used in 2D, but are as yet too computationally intensive for use in 3D.

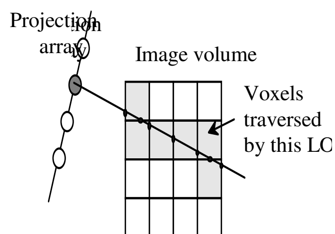


Figure XIV.7: *A LOR corresponding to one sinogram element is shown. The distance which intersects a particular voxel determines the weight this voxel carries in the projection process.*

XIV.2.1.2. Location

This function is located in

- *recon_buildingblock/fwdproj3D.cxx*
- *recon_buildingblock/fwdproj.3D_normalization.cxx*
- *include/recon_buildingblock/fwdproj3D.h*
- *recon_buildingblock/fwdproj.3D_normalization.h*

XIV.2.1.3. Functions

There is a whole sequence of 3D forward projectors. Several forward projection functions have been written depending on how we read projections data which can be organized either by view or by segment forms. The by-view routines will be used during parallelization of forward projection.

I. By-segment routines

First the functions working on a whole PETSegment. The routines use symmetry, so they require 2 PETSegments, where *segment_pos* has positive *average_ring_difference* = +*delta*, while *segment_neg* has *average_ring_difference* = -*delta*. The function which takes a view argument uses symmetries in the views as well, here, 0 <= *view* <= *num_views*/4 (= 45 degrees) (see backprojection). They also process 4 symmetries related views. These functions works for the moment with an odd number of bins elements (*get_num_bins()* is odd).

```
void forward_project(  const PETImageOfVolume &image,
                       PETSegment &segment_pos,
                       PETSegment &segment_neg);

void forward_project(  const PETImageOfVolume &image,
                       PETSegment &segment_pos,
                       PETSegment &segment_neg,
                       const int view);

void forward_project(  const PETImageOfVolume &image,
                       PETSegment &segment_pos,
                       PETSegment &segment_neg,
                       const int rmin, const int rmax);

void forward_project(  const PETImageOfVolume &image,
                       PETSegment &segment_pos,
                       PETSegment &segment_neg,
                       const int view,
                       const int rmin, const int rmax);
```

where

```
PETImageOfVolume &image                 // (in) the estimated  image matrix obtained from 2D FBP ;
PETSegmentBySinogram &segment_pos       // (out) segment with positive delta;
PETSegmentBySinogram &segment_neg       // (out) segment with negative delta;
const int view                          // (in) view number to be processed (going from 0 to
        num_views/4)
const int rmin                          // (in) lower discrte ring number
const int rmax                          // (in) upper discrete ring number
```

II. By-view routines

2.1.3.II.1. On 8 PETViewgrams

The following forward projection functions work on 8 viewgrams. They should not be used for *view*=0 or 45 degrees .

```
void forward_project(  const PETImageOfVolume &Image2D,
                       PETViewgram & pos_view,
                       PETViewgram & neg_view,
                       PETViewgram & pos_plus90,
                       PETViewgram & neg_plus90,
```

```
                           PETViewgram & pos_min180,
                           PETViewgram & neg_min180,
                           PETViewgram & pos_min90,
                           PETViewgram & neg_min90);

   void forward_project(const PETImageOfVolume& Image2D,
                           PETViewgram & pos_view,
                           PETViewgram & neg_view,
                           PETViewgram & pos_plus90,
                           PETViewgram & neg_plus90,
                           PETViewgram & pos_min180,
                           PETViewgram & neg_min180,
                           PETViewgram & pos_min90,
                           PETViewgram & neg_min90,
                           const int rmin, const int rmax);
```

where

```
PETImageOfVolume &image                  // (in) the estimated  image matrix obtained from 2D FBP
PETViewgram    &pos_view, &pos_plus90,
               &pos_min180, &pos_min90    // (out) viewgram with positive delta;
PETViewgram    &neg_view, &neg_plus90,
               &neg_min180, &neg_min90    // (out) viewgram with negative delta;
const int rmin                            // (in) lower ring number
const int rmax                            // (in) upper ring number
```

with argument names which encode this as follows

pos : positive average ring difference delta

neg : average ring difference – delta

plus90 : view + 90 degrees

min180 : 180 degrees – view

min90 : 90 degrees – view

2.1.3.II.2. On 4 PETViewgrams

These functions forward project 4 viewgrams related by symmetry. They should be used for *view* = 0 or 45 degrees . Calling the 8 viewgrams version for these special views internally reroutes to the function below (so you are wasting memory by using the 8 argument version).

```
   void forward_project(  const PETImageOfVolume & image,
                           PETViewgram & pos_view,
                           PETViewgram & neg_view,
                           PETViewgram & pos_plus90,
                           PETViewgram & neg_plus90);

   void forward_project(  const PETImageOfVolume & image,
                           PETViewgram & pos_view,
                           PETViewgram & neg_view,
                           PETViewgram & pos_plus90,
                           PETViewgram & neg_plus90,
                           const int rmin, const int rmax);
```

where the argument are already described in the backprojection section (see on 8viewgrams)

XIV.2.1.4. Algorithm

The forward projection processing in the PROMIS algorithm is carried out with an implementation of Siddon's algorithm [Siddon et al., 1985], calculating the length of intersection of LORs with the image voxels and making use of geometrical symmetries of the image volume. For forward-projection, a ray-driven (as opposed to voxel driven) method is used in order to minimise artefacts caused by the accumulation of inaccuracies at certain angles between the square (cubic) image grid and the direction of projection, in particular at 45 degrees (Figure X.2). As it is ray-driven operation, the time needed to estimate unmeasured oblique projections data depends mainly on the number of elements in the unmeasured part of the oblique projection.

By considering pixel boundaries as the intersection of orthogonal sets of equally spaced planes, rather than independently, the computational complexity is greatly reduced. The points of intersection of a line with a set of parallel planes are particularly simple to determine. Once one such point has been found, all the others may be found by recursion. Figure XIV.8 illustrates how in the 2D case the intersection of a line with a grid consists of two sets of points: the intersection with the set of vertical planes (black circles), and the intersection with the set of horizontal planes (white circles). The generalisation to 3D is straightforward. As the distance between adjacent white or black circle is a constant that depends only on the voxel size and the angle of the LOR, Siddon's algorithm exploits this idea: first compute seperately the three set $\{\alpha_{x,i}\}$, $\{\alpha_{y,i}\}$, $\{\alpha_{z,i}\}$ of intersections of the line with the three sets of parallel planes; the parameter α corresponds to the distance along the line starting from one end, related to the total length of the line, and is thus always comprised between 0 and 1. Merge the three sets $\{\alpha_i\}$ in order to obtain one sorted set of parametric intersections of the line with the 3D grid.

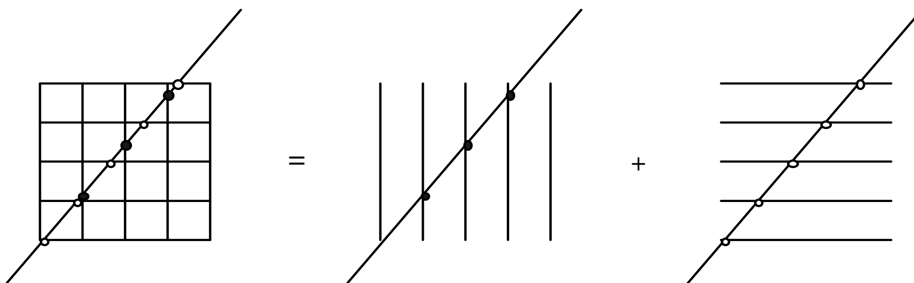


Figure XIV.8: *The lengths of intersection of a line with an array of pixels may be computed independently for every pixel (*left*), or, more efficiently, by considering pixel boundaries as the intersection of orthogonal sets of equally spaced planes (*middle *and* right*).*

The length of the intersection of the line with a particular pixel is then given by the difference between two adjacent parametric values in the merged set. Starting from the first voxel traversed by the line and moving along the LOR, the voxel values are weighted by the corresponding intersection length and summed in order to yield the projection of the image along that LOR. This value is then converted to the correct units by multiplying by the length of the LOR. A detailed description of this algorithm may be found in [Siddon et al., 1985].

Whereas voxel *values* do obviously not possess any known symmetry properties, all geometrical factors, such as lengths of intersection and voxel positions, follow the symmetries of the image grid. Symmetry operations act on all co-ordinates of a LOR, its azimuthal angle ϕ, its co-polar angle θ, and its radial and axial co-ordinates s and z respectively, and are used to reduce the amount of computation during the projection process. Details of these symmetries are

described in the next section. Using these various symmetries, the lengths of intersection computed for one voxel may be re-used straight away for as many as fifteen other voxels times the number of projected sinograms for a given ring index difference.

Since in our image representation the axial sampling distance of the image is half the distance between detector rings, the voxel size rather than the detector ring width must be used for the axial sampling distance of the forward projection, as shown in Fig. XIV.9.

For a 3D PET volume of N^3 voxels, the Siddon algorithm scales with 3*N*, with *N*, the number of planes, rather than N^3, the number of voxels.

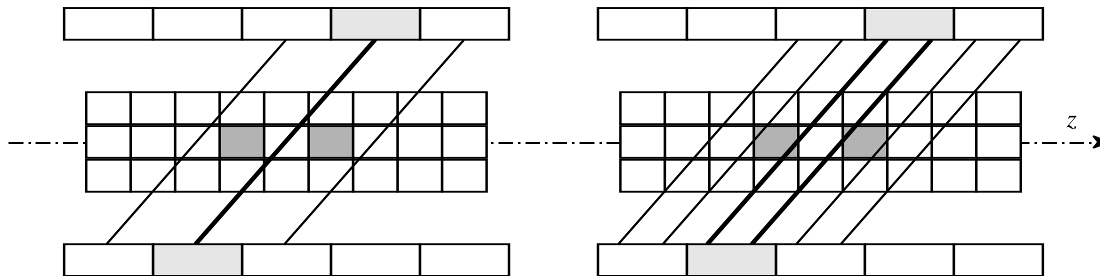


Figure XIV.9: *Axial section through the image volume and the detector rings. When forward-projecting with an axial sampling distance equal to the detector width, small details, such as the shaded voxels, remain unseen (*left*); the sampling distance of the projection must instead match that of the image (*right*).*

XIV.2.1.5. Software implementation

. Every image slice has a square cross-section, and the number of image elements per side is odd, i.e. one image element always lies at the centre of the slice

```
• image.get_min_z() == 0
```

The target image is divided into a number of slices perpendicular to the *z*-axis-transaxial slices shown in alternating shades of grey-numbered from z_o to z_{max}.(see figure IV.1).

```
• image.get_min_x() == -image.get_max_x()
```

The values of the units along different axes are chosen in the following way. Along the z-axis two systems are used concurrently: one unit of *v* is given by the ring spacing(inter-ring distance (*ring_spacing*), and it is thus twice of one unit of *z*, to the inter-slice distance (*bin_size*) (see figure IV.1.3).

Other prequisite parameters are .

```
. get_num_bins() ==image.get_x_size()
. image.get_voxel_size().x == scan_info.bin_size
. image.get_voxel_size().z * (image.get_z_size() + 1) == pos_view.scan_info.FOV_axial
. segment_pos.get_max_bin() == -segment_pos.min_bin()
. psegment_pos.get_num_bins() == image.get_x_size())
. segment_pos.get_max_bin() == segment_neg.get_max_bin()
. segment_pos.get_min_bin() == segment_neg.get_min_bin()
. segment_pos.get_average_ring_difference() == segment_neg.get_average_ring_difference()
```

The lowest forward_project(args...) calls Siddon program (see next section) to store symetry related into a 4D float tensor. This is done to keep frequently accessed data close to each other in memory (caching).

```
Projall[int Ring_num], [int ds], [int dz],, [float V] )
```

In the forward projection function, a loop which takes into account that axial ring size equals twice the voxel z_size. And this loop runs over 2 values offset of –0.25 and +0.25. The Siddon function changes when using CTI span>1 data. So, the origin in the the scanner space as well the origin of the image space starts from the middle of the first ring- However, for the origin of the image obtained from oblique sinograms are overlapping two rings. According to *z*-axis, the pixels of the image starts from –0.5 to 0.5 in the image space but starts from –0.25 to +0.25 in the scanner space (Figure XIV.10). This offset takes into account that axial ring size.

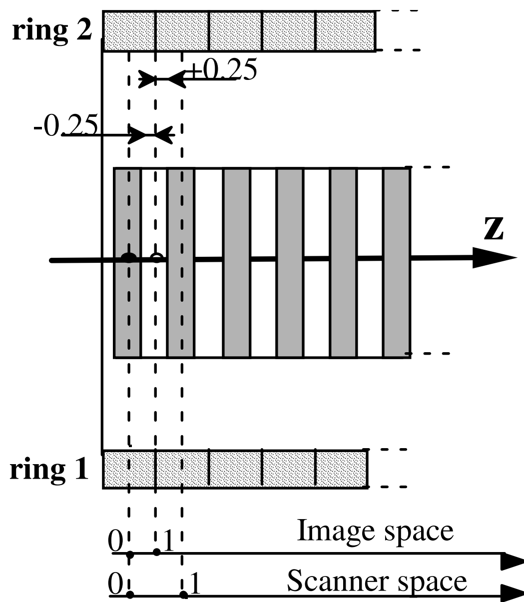


Figure XIV.10: *Explanation of the two offset values needed during the forward projection.*

XIV.2.2. Siddon Algorithm : Proj_Siddon()

XIV.2.2.1. Description

This Siddon projection function is used by forward_project as an internal function. It should not be used anywhere else [Siddon et al., 1985]. More details can be found elsewhere [Egger et al., 1998b]..

XIV.2.2.2. Location

The function is declared in

- *recon_buildingblock/siddonproj.cxx*

- *include/recon_buildingblock/fwdproj.h.*

XIV.2.2.3. Functions

```
void Proj_Siddon(       const PETImageOfVolume &image, Tensor4D<float> &Projptr,
                        const PETScanInfo &scan_info,
                        const float cphi, const float sphi, const float delta, const int s,
                        const float R, const int rmin, const int rmax, const float offset,
                        const int Siddon);
```

with

PETImageOfVolume &*image*, // (in) image matrix to forward project

Tensor4D<float> &*Projptr*, // (in) a 4D tensor of dimensions corresponding to *ring_num*, *ds dz*, *Voxel_case*, respectively

const PETScanInfo &scan_info, // (in) scanner information

float *cphi*, // (in) value of cos(*phi*);

float *sphi*, // (in) value of sin(*phi*);

int *s*, // (in) bin number;

float *R*, // (in) FOV radius;

int *rmin*, // (in) lower discrete ring index ;

int *rmax*, // (in) upper discrete ring index;

float *offset*, // (in) taking two values ± 0.25 (see more details in XI.2.5)

int *Siddon*; // (in) index of *Siddon* case for the symmetry properties with

1: $\phi = 0$ or 45, and $s > 0$;

2: $\phi = 0$ or 45, and $s = 0$;

3: ϕ □ 0 and 45, and $s > 0$;

4: ϕ □ 0 and 45, and $s = 0$.

The lowest forward_project(...) call Siddon program to store symetry related into a 4D float tensor. This is done to keep frequently accessed data close to each other in memory (caching).

Projall(int *Ring_num*, int *ds*,int *dz*,float *V*)) with:

ring num : the ring number to be processed (the rings of the scanner are numbered from r_{min} to r_{max};

ds : the bin element position regarding the the middle bin size (i.e origin) and has 2 values

- ds= 0 when s>0,
- ds = 1 when s<0;

dz : the displacement along the z-axis, 2 values corresponding to the position of *dz* with regards to the centre position

- dz =0 stay in the same slice,
- dz= 1, move one voxel along z;

V : 4 values corresponding to the four nearest voxel coordinates to the centre of voxel being projected onto the projection plane (ie:

- V_1: (r_0, s,
- V_2: (r_0, s+ 1),
- V_3: (r_0 + 1, s),
- V_4: (r_0 + 1, s+ 1))

with r_O is the corresponding ring number to be processed and *s*, the projection element number

XIV.2.2.4. Uses of geometrical symmetries

As already explained in the forward projection section, symmetry operations act on all co-ordinates of a LOR, ϕ, θ, s and z, and may be exploited to substantially reduce the amount of computation during the projection process. More details can be found in Egger et al. (1998b).

◀ **z-Symmetry**

The most obvious is the translational symmetry along the z–axis: since the axial image sampling is double the axial data sampling, parallel LORs shifted by one detector ring will traverse the image in geometrically equivalent positions shifted by double the voxel size along the z-axis; all geometrical quantities, such as lengths of intersection, need only be calculated for one LOR and are shared with all the voxels related by the translational symmetry group.

◀ **s-Symmetry**

Symmetry about the centre of the image volume

- voxel $A_0(X,Y,Z)$ on LOR (s, ϕ, θ, z)
- its counterpart $A_s(-X,-Y, Q)$ on LOR $(-s, \phi, \theta, z)$ with $Q = 4r_0 + 2 + 2\delta - Z$

◀ **ϕ–Symmetry**

The azimuthal angle of the LOR possesses a four fold symmetry. A voxel lying on a LOR given by the co-ordinates (s, ϕ, θ, z) between detector rings $(r_0 ; r_0 + 1)$ shares its geometrical properties with the following three additional voxels:

- voxel $A_1(Y,X,Q)$ on LOR $(s, \pi/2 - \phi, \theta, z)$, with $Q= 4r_0 +2 + 2\delta - Z$.
- voxel $A_2(-Y,X,Z)$ on LOR $(s, \pi/2 + \phi, \theta, z)$,
- voxel $A_3(-X,Y,Q)$ n LOR $(s, \pi - \phi, \theta, z)$.

◀ **θ–Symmetry**

Voxels along LORs at angle θ possess equivalent positions on LORs at angle $-\theta$;

- voxel $A_\theta(X,Y,Z')$ on LOR (s, ϕ, θ, z)
- voxel $A_\theta(X,Y,Q')$ on LOR $(s, \phi, -\theta, Q)$ with $Q = 4r_0 + 2 + 2\delta - Z$

Using all combinations described above the lengths of intersection computed for voxels $A_0(X,Y,Z)$ on LOR (s, ϕ, θ, z) may be used away for as many as others voxels times the number of projected sinograms $(r_0 ; r_0 + \delta)$ for the given angle θ.

XV. INTERFILE SUPPORT

XV.1. CLASSES FOR PARSING A FILE WITH INTERFILE-TYPE KEYS (CLASS KEYPARSER AND KEYARGUMENT)

XV.1.1. Description

Interfile headers consist of a sequence lines of the following form:

Keyword := argument

The KeyArgument class defines the possible 'types' of the argument. Supported types are:

- no argument (KeyArgument::None)
- a string (KeyArgument::ASCII)
- a list of strings (KeyArgument::LIST_OF_ASCII)
- an unsigned long (KeyArgument::ULONG)
- an integer number (KeyArgument::INT)
- a list of integers (KeyArgument::LIST_OF_INTS)

- a double (KeyArgument::DOUBLE)
- a list of doubles (KeyArgument::LIST_OF_DOUBLES)
- a string, but from a list of possible values (KeyArgument::ASCIIlist)

The KeyParser class provides the basic mechanisms to parse this type of header. It defines no keywords itself, this has to be done by a class derived from KeyParser.

XV.1.2. Location

- include/KeyParser.h
- buildblock/KeyParser.h

XV.1.3. Definition of the KeyArgument class

```
class KeyArgument
{
public:
enum type {NONE,ASCII,LIST_OF_ASCII,ASCIIlist, ULONG,INT,
    LIST_OF_INTS,DOUBLE, LIST_OF_DOUBLES};
};
```

XV.1.4. Definition of the KeyParser class

```
class KeyParser
{
public:
 KeyParser();
 bool parse(istream& f);
  bool parse(const char * const filename);
  ////// functions to add keys and their actions
  typedef void (KeyParser::*KeywordProcessor)();
  void add_key(const string& keyword,
      KeyArgument::type t, KeywordProcessor function,
      void* variable= 0, const ASCIIlist_type * const list = 0);
  void add_key(const string& keyword, KeyArgument::type t,
      void* variable, const ASCIIlist_type * const list = 0);
protected :
 // Override the next function if you need to check values of the keys
  // Returns 0 of OK, 1 of not.
  virtual int post_processing() ;
  ////// predefined actions for add_key()
  // to start parsing, has to be set by first keyword
  void start_parsing();
  // to stop parsing
  void stop_parsing();
  void do_nothing() {};
  // set the variable to the value given as the value of the keyword
  void set_variable();
};
```

XV.1.5. Usage

As this class does not contain any keywords, it is cannot be used by the 'end-user'. However, it does present the interface which the 'end-user' needs to know about. As an example, the InterfileImageHeader class is derived from KeyParser. A program would work as follows

```
Istream input("headerfile.hv");
InterfileImageHeader hdr;
if (!hdr.parse(input))
    {      PETerror("\nError parsing interfile header,\n"); Abort(); }
// do something with public members of hdr here
```

The derived class has to set the keywords as in the example below:

```
  add_key("INTERFILE",
      KeyArgument::NONE, &KeyParser::start_parsing);
  add_key("number of bytes per pixel",
      KeyArgument::INT, &bytes_per_pixel);
  // add some more here
  add_key("END OF INTERFILE",
      KeyArgument::NONE, &KeyParser::stop_parsing);
```

As the implementation of this class is still somewhat unstable, further details are postponed to the next version of this deliverable.

XV.2. A CLASS FOR THE COMMON KEYWORDS IN THE INTERFILE HEADER (CLASS INTERFILEHEADER)

XV.2.1. Description

This is a base class that will be further refined depending on the type of data the Interfile header contains. Its public variables correspond to the values of the common Interfile keywords (see D1.1 for details on the Interfile headers).

XV.2.2. Location

- include/InterfileHeader.h
- buildblock/InterfileHeader.cxx

XV.2.3. Definition

```
class InterfileHeader : public KeyParser
{
public:
InterfileHeader();

protected:
 virtual int post_processing();          // contains consistency checks
 ASCIIlist_type PET_data_type_values;
 int PET_data_type_index;

public :
  string                                       data_file_name;
  NumericType                                  type_of_numbers;
  ByteOrder                                    file_byte_order;
  int                                    num_dimensions;
  int                                    num_time_frames;
  vector<string>                               matrix_labels;
  vector<vector<int> >                   matrix_size;
  vector<double>                               pixel_sizes;
  vector<vector<double> >                image_scaling_factors;
  vector<unsigned long>                  data_offset;
};
```

XV.3. A CLASS FOR INTERFILE HEADERS CONTAINING IMAGES (CLASS INTERFILEIMAGEHEADER)

XV.3.1. Description

This class contains the information for images. At the moment, only one image per file is supported. This functionality will be integrated with class *PETImageData* in the next version.

XV.3.2. Location

- include/InterfileHeader.h

- buildblock/InterfileHeader.cxx

XV.3.3. Definition

```
class InterfileImageHeader : public InterfileHeader
{
public:
InterfileImageHeader();

protected:
virtual int post_processing();           // extra consistency checks
};
```

XV.4. A CLASS FOR INTERFILE HEADERS CONTAINING PROJECTION DATA (CLASS INTERFILEPSOVHEADER)

XV.4.1. Description

This class contains the information for projection data. At the moment, only one PETSinogramOfVolume per file is supported. It contains additional public members specific to projection data. This functionality will be integrated with class *PETAcquisitionData* in the next version.

XV.4.2. Location

- include/InterfileHeader.h
- buildblock/InterfileHeader.cxx

XV.4.3. Definition

```
class InterfilePSOVHeader : public InterfileHeader
{
public:
InterfilePSOVHeader();

protected:
 virtual int post_processing();          // extra consistency checks

public:
  vector<int> segment_sequence;
  vector<int> min_ring_difference;
  vector<int> max_ring_difference;
  vector<int> num_rings_per_segment;

  int num_segments;
  int num_views;
  int num_bins;
  PETSinogramOfVolume::StorageOrder storage_order;
};
```

XV.5. INTERFILE FUNCTIONS

XV.5.1. Description

This is a set of functions to read and write interfile data. This functionality will be integrated in the PETStudy class and its derived classes.

read_interfile_image() reads the first image from an Interfile header and returns a PETImageOfVolume object.

write_basic_interfile() writes an image (either Tensor3D or PETImageOfVolume). It returns 'true' when successful, 'false' otherwise. Extensions .hv and .v will be added to the parameter 'filename'.
read_interfile_PSOV () reads the first set of projection data in the Interfile header and returns a PETSinogramOfVolume object.

XV.5.2. Location

include/interfile.h

buildblock/interfile.cxx

XV.5.3. Definition

```
PETImageOfVolume read_interfile_image(istream& input);
PETImageOfVolume read_interfile_image(const char *const filename);
template <class NUMBER>
bool write_basic_interfile(const char * const filename,
                                            const Tensor3D<NUMBER>& image,
                                            const Point3D& voxel_size,
                                            const NumericType output_type = NumericType::FLOAT);

template <class NUMBER>
inline bool
write_basic_interfile(const char * const filename,
                  const Tensor3D<NUMBER>& image,
                  const NumericType output_type = NumericType::FLOAT);
bool
write_basic_interfile(const char * const filename,
                  const PETImageOfVolume& image,
                  const NumericType output_type = NumericType::FLOAT);
PETSinogramOfVolume read_interfile_PSOV(istream& input);
PETSinogramOfVolume read_interfile_PSOV(const char *const filename);
```

XVI. MISCELLANEOUS UTILITY FUNCTIONS

XVI.1. FAST FOURIER TRANSFORM UTILITIES

XVI.1.1. Description

This concerns all the fast Fourier transform utilities. All the functions decribed below (four1, fourn, realft, realft3, convlv) are extracted from NUMERICAL RECIPES IN C [Press et al., 1992].

XVI.1.2. Location:

- *recon_buildingblock/fft.cxx*
- *include/recon_buildblock/fft.h*

XVI.1.3. 1D discrete Fourier transform : four1()

XVI.1.3.1. Description

This function replaces data by its discrete Fourier transform, if isign is input as 1; or replaces data by nn times its inverse discrete Fourier transform, if isign is input as –1. data is a complex array of length nn, input as a real array data[1..2*nn]. nn MUST be an integer power of 2 (this is not checked for !).

XVI.1.3.2. Function

```
void four1(
      Tensor1D<float> &data,              // (in/out) 1D array data
      int nn,                             // (in) length of the array of data
      int isign)                          // (in)isign =1 => discrete Fourier transform ,
                                            isign = -1,. discrete inverse Fourier transform
```

XVI.1.4. Ndim-dimensional discrete Fourier transform: fourn()

XVI.1.4.1. Description

This function replaces data by its ndim-dimensional discrete Fourier transform, if isign is input as 1. nn[1..ndim] is an integer array containing the lengths of each dimension (number of complex values), which MUST all be powers of 2. data is a real array of length twice the product of these lengths, in which the data are stored as in a multidimensional complex array: real and imaginary parts of each element are in consecutive locations, and the rightmost index of the array increases most rapidly as one proceeds along data. For a two-dimensional array, this is equivalent to storing the array by rows . If isign is in input as –1 , data is replaced by its inverse transform time the product of the lengths of all dimensions.

XVI.1.4.2. Function

```
void fourn (
      Tensor1D<float> &data,              // (in/out) N-dimensional 1D array. On output, returns
             data by its ndim-dimensional discrete Fourier transform
      Tensor1D <unsigned long> &nn,       // (in) vector containing the length of each dimension,
             which must all be powers of 2
      int ndim,                           // (in) Number of dimensions in the data vector
      int isign);                         // (in) isign =1 => ndim-dimensional Fourier transform ,
                                            isign = -1,. ndim-dimensional inverse Fourier
                                            transform
```

XVI.1.5. Convolution: convlv()

XVI.1.5.1. Description

This routine convolves a real vector of length n with a given filter which is given at frequencies 0 to 1/2 in steps of 1/n (with N = n/2+1).

XVI.1.5.2. Function

```
void convlv (
      Tensor1D<float> &data,              // (in/out) 1D data array. On output, returns data by
             its the convolved (or filtered) data
      Tensor1D <float> &filter,           // (in) filter vector to be applied on data
      int n);                             // (in) length of the data vector
```

XVI.1.6. Convolution: convlvC()

XVI.1.6.1. Description

Same as above (convlv) except that the first component of the complex transform is returned in data[1], the last component in data[2].

XVI.1.6.2. Function

```
void convlvC (
```

```
      Tensor1D<float> &data,              // (in/out) real 1D data array. On output, returns data
              by its the convolved (or filtered) data
      Tensor1D <float> &filter,           // (in) filter vector to be applied on data
      int n);                             // (in) length of the data vector
```

XVI.1.7. Fourier transform: realft()

XVI.1.7.1. Description

This routine calculates the Fourier transform of a set of n real-valued data points, replaces this data (which is stored in array data[1..n] by the positive frequency half of its complex Fourier transform. The real-valued first and last components of the complex transform are returned as elements data[1] and data[2], respectively. n must be a power of 2. This routine also calulates the inverse transform of a complex data array if it is the transform of real data (result in this case must be multiplied by 2/n).

XVI.1.7.2. Function

```
void realft (
      Tensor1D <float> &data,             // (in/out) Input data to calculte its Fourier
        transform. On output,returns the positive frequency
                                            half of its complex Fourier transform
      int n,                              // (in) length of data array (must be a power of 2).
      int isign);                         // (in) isign =1 => Fourier transform ,
                                             isign = -1, inverse Fourier transform
```

XVI.1.8. Complex fast Fourier transform as two complex arrays: realft3()

XVI.1.8.1. Description

Given a three-dimensional real array data[1..nn1][1..nn2][1..nn3] (where nn1=1 for the case of a logically two-dimensional array), this routine returns (for isign=1) the complex fast Fourier transform as two complex arrays: On output, data contains the zero and positive frequency values of the third frequency component, while speq[1..nn1][1..2*nn2] contains the Nyquist critical frequency values of the third frequency component.

First (and second) frequency components are stored for zero, positive, and negative frequencies, in standard wrap-around order. See text for description of how complex values are arranged. For isign=–1, the inverse transform (times nn1*nn2*nn3/2 as a constant multiplicative factor) is performed, with output data (viewed as a real array) deriving from input data (viewed as complex) and speq. The dimensions nn1, nn2, nn3 must always be integer powers of 2.

XVI.1.8.2. Function

```
void rlft3(
      Tensor3D <float> &data,             // (in/out)3D array of data; On output, data contains
                                            the zero and positive frequency values of the third
                                            frequency component
      Tensor2D <float> &speq,             // (out) 2D array containing the Nyquist critical
                                            frequency values of the third frequency component
      unsigned long nn1,                  // (in) length of the first dimension (must be a power of 2).
      unsigned long nn2,                  // (in) length of the second dimension (must be a power of 2).
      unsigned long nn3,                  // (in) length of the third dimension (must be a power of 2).
      int isign);                         // (in) isign =1 => complex fast Fourier transform ,
                                            isign = -1,. inverse Fourier transform
```

XVI.2. VARIOUS UTILITY FUNCTIONS

XVI.2.1. display DATA

XVI.2.1.1. Description

A set of functions to display general data (2D or 3D tensors). The functions are templated for generality. In the text below, we use the following names for the templated types:

NUMBER is the type of elements in the **Tensor3D** or **Tensor2D** object.

SCALE is the type of the scale factors

CHARP is the type of the text strings (char * or String)

There is an effective threshold at 0 currently (i.e. negative numbers are cut out).

XVI.2.1.2. Location

- *buildingblock/display.cxx*,
- *buildingblock/screen.c*
- *buildingblock/gen.c*
- *buildingblock/screen.h*
- *buildingblock/gen.h*
- *buildingblock/screengen.c*
- *include/display.h*

XVI.2.1.3. Definitions

```
/* The function for Tensor2D objects */
template <class NUMBER, class SCALE, class CHARP>
void display(const Tensor3D<NUMBER>& plane_stack,
               const VectorWithOffset<SCALE>& scale_factors,
               const VectorWithOffset<CHARP>& text,
               double maxi = 0, int zoom = 0,
               int num_in_window = 0);

/* A version without scale factors and text */
template <class NUMBER>
void display(const Tensor3D<NUMBER>& plane_stack,
          double maxi = 0, int zoom = 0,
          int num_in_window = 0);

/* The function for Tensor2D objects */
template <class NUMBER, class SCALE, class CHARP>
inline void display(const Tensor2D<NUMBER>& plane,
               const SCALE scale_factor,
               const CHARP& text,
               double maxi = 0, int zoom = 0,
               int num_in_window = 0);

/* A version without scale factors and text */
template <class NUMBER>
void display(const Tensor2D<NUMBER>& plane,
          double maxi = 0, int zoom = 0,
          int num_in_window = 0);
```

```
/* A version with txt */
template <class NUMBER, class CHARP>
void my_display(const Tensor3D<NUMBER> &image, CHARP text);

template <class NUMBER>
void display2D(const Tensor2D<NUMBER> &plane);

/* A version for displaying 8 viewgrams */
void display_8_views(const PETViewgram& v1, const PETViewgram& v2,
                     const PETViewgram& v3, const PETViewgram& v4,
                     const PETViewgram& v5, const PETViewgram& v6,
                     const PETViewgram& v7, const PETViewgram& v8);
```

XVI.2.1.4. Usage

The display functions of **Tensor3D** objects take parameters as follows:

plane_stack:

the **Tensor** object

scale_factors :

a **VectorWithOffset** of factors which are multiplied with the numbers in the Tensor object to give the "real" values

text

a **VectorWithOffset** of strings that are displayed below the images

maxi :

a double which gives the ("real") value that will correspond to the maximum of the colour scale. All bigger values are displayed with the same colour. If maxi is 0, all planes are scaled independently.

zoom :

an int giving the number of times the image should be enlarged. Enlargement currently is with linear interpolation, giving reasonably smooth images. If *zoom* = 0, maximum enlargement is used.

num_in_window :

an int giving the desired maximum number of images displayed in one window (there can be less images displayed if they do not fit). If *num_in_window* = 0, the maximum number of images will be displayed.

Note that the scale_factors and text arrays are supposed to have the same range as the outer dimension of the **Tensor3D** object.

For **Tensor2D** objects, the parameters are similar, but scale_factors and text are not vectors anymore.

XVI.2.1.5. Implementation details

The implementation of these functions is rather primitive, and taken from an old library by KT. This library runs on any Xwindows system, but also on PCs in MSDOS mode (a number of (older) graphics cards for higher resolution are supported). This implementation is only intended for testing purposes.

Images are displayed a "screen" (or window) at a time. If all images do not fit, a second screen is displayed after the first is dismissed, which is done by pressing any key when the window is selected.

There is no interaction with the displayed images (aside from changing the colour scale by pressing the right mouse button).

XVI.2.2. ERROR MESSAGE : PETERROR()

XVI.2.2.1. Description

A function that writes warning messages to the screen. It uses the same syntax is the C function *printf*. Should be superseded by using 'debug' and 'warning' streams later on.

XVI.2.2.2. Location

- *include/pet_common.h*

XVI.2.2.3. Function

```
PETerror(char *);
```

XVI.2.2.4. Example

To display a failure in memory allocation

```
PETerror("Failure of memory allocation\n");
```

XVI.2.3. ABORDING PROGRAM : ABORT()

XVI.2.3.1. Description

A function to stop the execution of the program. Should be superseded by exceptions later on.

XVI.2.3.2. Location

- *include/pet_common.h*

XVI.2.3.3. Function

```
Abort();
```

XVI.2.4. CHECKING : ASSERT ()

XVI.2.4.1. Description

The same macro as in the standard include file <assert.h> with one difference: it is only non-empty when the pre-processor symbol _DEBUG is defined.

This function is used in order ot check if the boolean expression is true otherwise it will abort. This function is used quite often for instance to check in the case of tensors whether the index of the tensor is ranged between the start and the last index of each dimension of the tensor (1D, 2D, 3D or 4D) as follows:

assert((i>=start)&&(i<(length+start)))

XVI.2.4.2. Location

- *include/pet_common.h*

XVI.2.4.3. Function

```
assert(bool expr)
```

XVI.2.5. Predefined streams

XVI.2.5.1. Description

There are four different output streams *debug_stream, log_stream, warning_stream, error_stream.* They default to write their output to the *cerr* stream. If you want to duplicate the output to another stream use for instance:

```
ofstream log("logfile.log");
log_stream.add(log);
```

XVI.2.5.2. Location

- *include/pet_streams.h*

XVII. REFERENCES

Cho Z H, Chen C M and Lee S Y **(1990)** "Incremental algorithm – A new fast backprojection scheme for parallel beam geometries," IEEE Transactions on Medical Imaging, 9-2, p. 207-217.

Colsher J G **(1980)** "Fully three-dimensional positron emission tomography," Physics in Medicine and Biology, 25-1, p. 103-115.

Comtat C, Morel C, Defrise M and Townsend D W **(1993)** "The Favor algorithm for 3D PET data and its implementation using a network of transputers," Physics in Medicine and Biology, 38, p. 929-944.

Daube-Witherspoon M.E. and Muehllehner G. **(1987)** Treatment of axial data in three-dimensional PET. J Nucl Med. 82:1717-1724.

Defrise M., Townsend D.W. and Clack R. **(1989)** Three-dimensional image reconstruction from complete projections. Phys Med Biol 34: 5: 573-587.

Defrise, D. Townsend, A. Geissbuhler, **(1990)** "Implementation of three-dimensional image reconstruction for multi-ring positron tomographs". Phys. Med. Biol. (1990), Vol. 35, No 10, 1361-1372.

Defrise M, Townsend D W and Clack R **(1992)** "Favor: a fast reconstruction algorithm for volume imaging in PET," Conference Records 1991 IEEE Medical Imaging Conference, p. 1919-1923.

Defrise M., Kinahan P. and Townsend D. **(1995)** A new rebinning algorithm for 3D PET: Principle, implementation and performance. Conference Records 1995 International meeting on fully three-dimensional image reconstruction in radiology and nuclear medicine, Aix-Les-Bains, France. 235-239.

Edholm P R, Lewitt R M and Lindholm B, **(1986)** Novel properties of the Fourier decomposition of the sinogram,"in International Workshop on Physics and Engineering of Computerised Multidimensional Imaging and Processing, Proceedings of the SPIE, 671, p. 8-18,.

Egger M L **(1996)** Fast Volume Reconstruction in Positron Emission Tomography: Implementation of Four Algorithms on a High-Performance Scalable Parallel Platform, PhD Thesis, University of Lausanne

Egger ML, Herrman Scheurer AK, Joseph C, and Morel C **(1996)** Fast volume reconstruction in positron emission tomography: implementation of four algorithms on a high-performance scalable parallel platform, Conf. Rec. IEEE Med. Imag. Conf., Anaheim, 1996, pp. 1574-1578 (New York IEEE, 1997)

Egger ML, and Morel C. **(1998a)** "Execution times of five reconstruction algorithms in 3D positron emission tomography". Phys. Med. Biol. 43 p 703-712.

Egger ML, Joseph C, and Morel C. **(1998b)** "Incremental beamwise backprojection using geometrical symmetries for 3D PET reconstruction in a cylindrical scanner geometry". Phys. Med. Biol.43 p. 3009-3024.

He Y J, Cai A and Sun J-A **(1993)** "Incremental backprojection algorithm: Modification of the searching flow scheme and utilization of the relationship among projection views," IEEE Transactions on Medical Imaging, 12-3, p. 555-559.

Herrmann Scheurer A K, Egger M L, Joseph C and Morel C **(1995)** "A Monte Carlo phantom simulator for positron emission tomography," in Proceedings of the 1994 Workshop on Supercomputers in Brain Research, Jülich, 205-209, Herrmann H J, Wolf D E and Pöppel E, editors, World Scientific Publishing.

Kinahan P E and Rogers J G **(1989)**, "Analytic 3D image reconstruction using all detected events," IEEE Transactions on Nuclear Science, 36-1, p. 964-968.

Lewitt R.M., Muehllehner G. and Karp J.S. **(1994)** Three-dimensional reconstruction for PET by multi-slice rebinning and axial image filtering. Phys Med Biol. 39:321-340.

Press W H, Teukolsky S A, Vetterling W T and Flannery B P **(1992)** Numerical Recipes in C: the Art of Scientific Computing, 2nd edition, Cambridge University Press

Siddon R L **(1985)** "Fast calculation of the exact radiological path for a three-dimensional CT array," Medical Physics, 12-2, p. 252-255.

XVIII. ANNEXES

ANNEX 1

LIST OF ALL PARAPET SOURCE CODES NEEDED

FOR PET IMAGE RECONSTRUCTION

(Data Classes, common building blocks,specific building blcoks, test files, data display...)

• **include/:**

pet_common.h	// Common utilities
utilities.h	// Common utilities (user interaction and stream initialisation)
VectorWithOffset.h	// Class for vectors with offsets
NumericVectorWithOffset.h	// Class for for vectors with numeric elements
Tensor1D.h	// Class for 1D tensors
Tensor2D.h	// Class for 2D tensors
Tensor3D.h	// Class for 3D tensors
Tensor4D.h	// Class for 4D tensors
Tensorbase.h	// Base class for tensors
TensorFunction.h	// Base class for some additional functionality for Tensor objects
convert.h	// Classes for conversion between tensors of different numeric types
NumericInfo.h	// Class for numeric type information
Point3D.h	// Class for 3D coordinates (x,y,z)
PETScannerInfo.h	// Class for PET scanner information
PETScanInfo.h	// Class for PET scan information
sinodata.h	// Classes for projection data
imagedata.h	// Classes for image data
study.h	// Classes for general study information
Filter.h	// Classes for filtering
Reconstruction.h	// Base classes for reconstruction algorithms (both analytic and iterative)
KeyParser.h	// Class for parsing Interfile keys

InterfileHeader.h // Classes for headers for images and projection data
interfile.h // Functions for reading/writing Interfile data.

Timer.h // Classes for timers
debug.h // Debug utilities
display.h // Classes for displaying data on the screen

• include/recon_buildblocks/:

This directory contains all the common building blocks for image reconstruction (FFT, backprojection, forward projection, 2D FBP, zooming and timers)

fft.h // General base class for FFT utilities
bckproj.h // General base classe for backprojection utilities
fwproj.h // General base class for forward projection utilities
zoom.h // General base classes for rescaling, resizing, shifting data
timers.h // General base classes for timers

• buildblock/:

This directory contains the implementation of a few classes and functions of tensors, conversion, PET scanner information.

T*ensor1D.cxx* // Member functions of include/Tensor1D.h
T*ensors.cxx* // Member functions of include/Tensor[234]D.h
convert.cxx // Implementations for include/convert.h')
sinodata.cxx // Member functions of include/sinodata.h
imagedata.cxx // Member functions of include/imagedata.h
PETScannerInfo.cxx // Member functions of include/PETScannerInfo.h
PETScanInfo.cxx // Member functions of include/PETScanInfo.h
KeyParser.cxx // Member functions of include/KeyParser.h
InterfileHeader.cxx // Member functions of include/InterfileHeader.h
Interfile.cxx // Implementations for include/interfile.h

viewdata.cxx // Member functions for view data operations

• display/:

This directory contains utilities for displaying data on the screen

display.cxx // implementation of the interface
gen.h // Standard include file where all incompatibilities between various systems are taken care of
gen.c *// See gen.h*
screen.h // macros (and a few functions) for displaying stuff on a screen

screen.c // see screen.h
screengen.h // macros (and a few functions) for displaying stuff on a screen

• recon_buildblock/:

This directory contains the source code for reconstruction building blocks of backprojection (2D and 3D), FBP, forward projection, zooming ...

filtRamp.cxx // Ramp filter utilities
FBP2D.cxx // 2D filtered backprojection utilities
bakproj2D cxx // Utilities for 2D backprojection
backproj3D cxx // Utilities for 3D backprojection
baakproj3D_Cho.cxx // Utilities for CHO backprojection (this file is a subset of backproj3D.cxx)
fwdproj3D.cxx // Utilities for 3D forward projection
siddonproj.cxx // 2D filtered backprojection utilities (this file is a subset of fwdproj3D.cxx)
zoom.cxx // Zooming utilities (rescaling, shifting, resizing data)
timers.cxx // CPU timers and macros for timer utilities used in the reconstruction building blocks

• test/:

This directory contains some general test programs

ecat6bysegmentc.xx // Conversion of ECAT 6 data into bitmap organized by the following order storage (segment, ring, view, bin)
// This file will be moved to buildingblocks directory
createsino.cxx // Utilities for creating sinograms
readsino.cxx // Display bitmap data (sinograms or images)
// This file will be moved to buildingblocks directory
tensor.cxx // Test file for checking tensors problems (conversion, assignement...)
Timer.cxx // Test file for printing out CPU timing

• recon_test/:

This directory contains some general test programs

bctkest.xx // Test file for debugging 2D and 3D backprojection
fwdkest.xx // Test file for debugging 2D and 3D forward projection
zoomtest.xx // Test file for debugging zooming data

• utilities/ :

ecat6bysegmentxx // Conversion of ECAT 6 CTI data into bitmap organized by the following order storage (segment, ring, view, bin). This is for data with span = 1
ecat6bysegmentspan.xx // Conversion of ECAT 6 CTI data into bitmap organized by the following order Storage (segment, ring, view, bin). This is for data with span=3 and 7
conv2cti.xx // Conversion of bitmap into ECAT6 CTI format

vox.cxx	// Display *image* bitmap data with a lot of useful utilities (images manipulation, min/max value, getting rim...)
readsino.cxx	// Display bitmap data (sinograms or images)
	// This file will be moved to buildingblocks directory
swap_byteorder.*cxx*	// Functions for changing the byteorder of a file
	// This file will be moved to buildingblocks directory

ANNEX 2:

SYMMETRIC VOXELS AND CORRESPONDING V PARAMETERS

FOR POSITIVE RING INDEX DIFFERENCES Δ.

s	θ	ϕ	Voxel co-ordinates	V parameters
+	+	$0 < \phi < \pi/4$	(X, Y, Z) $(X, Y, Z+1)$	V_1: (r_0, n) V_2: $(r_0, n+1)$ V_3: (r_0+1, n) V_4: $(r_0+1, n+1)$
–	+	$0 < \phi < \pi/4$	$(-X, -Y, Q)$ $(-X, -Y, Q-1)$	V_1: $(r_0+1, -n)$ V_2: $(r_0+1, -n-1)$ V_3: $(r_0, -n)$ V_4: $(r_0, -n-1)$
+	+	$\pi/2 - \phi$	(Y, X, Q') $(Y, X, Q-1)$	V_1: (r_0+1, n) V_2: $(r_0+1, n+1)$ V_3: (r_0, n) V_4: $(r_0, n+1)$
–	+	$\pi/2 - \phi$	$(-Y, -X, Z)$ $(-Y, -X, Z+1)$	V_1: $(r_0, -n)$ V_2: $(r_0, -n-1)$ V_3: $(r_0+1, -n)$ V_4: $(r_0+1, -n-1)$
+	+	$\pi/2 + \phi$	$(-Y, X, Z)$ $(-Y, X, Z+1)$	V_1: (r_0, n) V_2: $(r_0, n+1)$ V_3: (r_0+1, n) V_4: $(r_0+1, n+1)$
–	+	$\pi/2 + \phi$	$(Y, -X, Q')$ $(Y, -X, Q-1)$	V_1: $(r_0+1, -n)$ V_2: $(r_0+1, -n-1)$ V_3: $(r_0, -n)$ V_4: $(r_0, -n-1)$
+	+	$\pi - \phi$	$(-X, Y, Q)$ $(-X, Y, Q-1)$	V_1: (r_0+1, n) V_2: $(r_0+1, n+1)$ V_3: (r_0, n) V_4: $(r_0, n+1)$
–	+	$\pi - \phi$	$(X, -Y, Z)$ $(X, -Y, Z+1)$	V_1: $(r_0, -n)$ V_2: $(r_0, -n-1)$ V_3: $(r_0+1, -n)$ V_4: $(r_0+1, -n-1)$

The symbol Q is used for $4r_0 + 2 + 2\delta - Z$.

ANNEX 3:

SYMMETRIC VOXELS AND CORRESPONDING *V* PARAMETERS

FOR NEGATIVE RING INDEX DIFFERENCES Δ.

s	θ	ϕ	Voxel co-ordinates	V parameters
+	–	$0 < \phi < \pi/4$	$(X, \quad Y, \quad Q$ $(X, \quad Y, \quad Q-1)$	V_1: $(r_0 + 1, \quad n)$ V_2: $(r_0 + 1, \quad n + 1)$ V_3: $(r_0, \quad n)$ V_4: $(r_0, \quad n + 1)$
–	–	$0 < \phi < \pi/4$	$(-X, \quad -Y, \quad Z)$ $(-X, \quad -Y, \quad Z+1)$	V_1: $(r_0, \quad -n)$ V_2: $(r_0, \quad -n - 1)$ V_3: $(r_0 + 1, \quad -n)$ V_4: $(r_0 + 1, \quad -n - 1)$
+	–	$\pi/2 - \phi$	$(Y, \quad X, \quad Z)$ $(Y, \quad X, \quad Z+1)$	V_1: $(r_0, \quad n)$ V_2: $(r_0, \quad n + 1)$ V_3: $(r_0 + 1, \quad n)$ V_4: $(r_0 + 1, \quad n + 1)$
–	–	$\pi/2 - \phi$	$(-Y, \quad -X, \quad Q')$ $(-Y, \quad -X, \quad Q-1)$	V_1: $(r_0 + 1, \quad -n)$ V_2: $(r_0 + 1, \quad -n - 1)$ V_3: $(r_0, \quad -n)$ V_4: $(r_0, \quad -n - 1)$
+	–	$\pi/2 + \phi$	$(-Y, \quad X, \quad Q$ $(-Y, \quad X, \quad Q-1)$	V_1: $(r_0 + 1, \quad n)$ V_2: $(r_0 + 1, \quad n + 1)$ V_3: $(r_0, \quad n)$ V_4: $(r_0, \quad n + 1)$
–	–	$\pi/2 + \phi$	$(Y, \quad -X, \quad Z)$ $(Y, \quad -X, \quad Z+1)$	V_1: $(r_0, \quad -n)$ V_2: $(r_0, \quad -n - 1)$ V_3: $(r_0 + 1, \quad -n)$ V_4: $(r_0 + 1, \quad -n - 1)$
+	–	$\pi - \phi$	$(-X, \quad Y, \quad Z)$ $(-X, \quad Y, \quad Z+1)$	V_1: $(r_0, \quad n)$ V_2: $(r_0, \quad n + 1)$ V_3: $(r_0 + 1, \quad n)$ V_4: $(r_0 + 1, \quad n + 1)$
–	–	$\pi - \phi$	$(X, \quad -Y, \quad Q')$ $(X, \quad -Y, \quad Q-1)$	V_1: $(r_0 + 1, \quad -n)$ V_2: $(r_0 + 1, \quad -n - 1)$ V_3: $(r_0, \quad -n)$ V_4: $(r_0, \quad -n - 1)$

The symbol Q' is used for $4r_0 + 2 + 2\delta - Z$.