STIR
Overview for developers

Kris Thielemans

version 2.0

Contents

1 Introduction

The objective of this document is to describe the STIR common building blocks library.
The library has been designed so that it can be used for many different algorithms and scanner geometries, including both cylindrical PET scanners and dual-head rotating coincidence gamma cameras (although the latter needs some work). The library contains classes and functions to run parts of the reconstruction in parallel on distributed memory architectures, but the parallel features are not discussed here.
The building block classes discussed in this document are:

- Classes for images (2D and 3D)
- Classes for projection data (i.e. the measured data)
- Forward projection, back projection and projection matrix classes.
- Classes for iterative reconstruction algorithms
- IO

The documentation for the STIR library is generated automatically from the source using the *doxygen* program (`http://www.doxygen.org/`). This can produce an HTML version but also various other output formats (you can get e.g. RTF or LaTeX output by simply running doxygen on the source files that come in the STIR distribution).

2 General conventions

2.1 Units

Distances are in millimetre.
Angles are in radians.
Relative times are in seconds.

2.2 Coordinate system for image data

Although STIR is prepared for general images such as blobs on bcc grids (see the online documentation for class **DiscretisedDensity**), currently only voxels on a Cartesian grid are implemented (class **VoxelsOnCartesianGrid**). The coordinate axes for Cartesian grids are chosen as follows.

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

y-axis : vertical axis, pointing downwards

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 (z=0) is located in the middle of the first ring (i.e at the opposite side of the bed). Note that for images with an even size in x and y, the axis of the scanner *does* coincide with the centre of a pixel. In particular, for range of *2n*, the (internal) image coordinates would run from *-n* to *(n-1)*.

2.3 Info on projection data

Conventional axes and units used for projection data are shown in Figures 2 and 1..

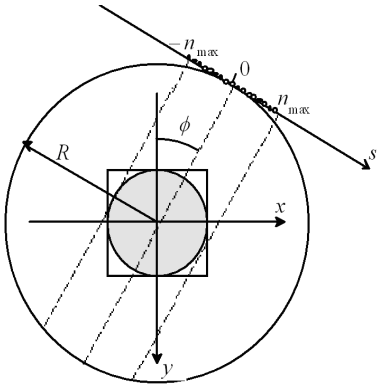

Figure 1: Axes and units within one transaxial slice of the target image. The transaxial section of the field of view is shown as a grey circle. The number of measured projection elements along the s-axis is odd, so that s =0 is positioned at the centre of the central projection element. The angles θ and ϕ define the direction of the line-of-response

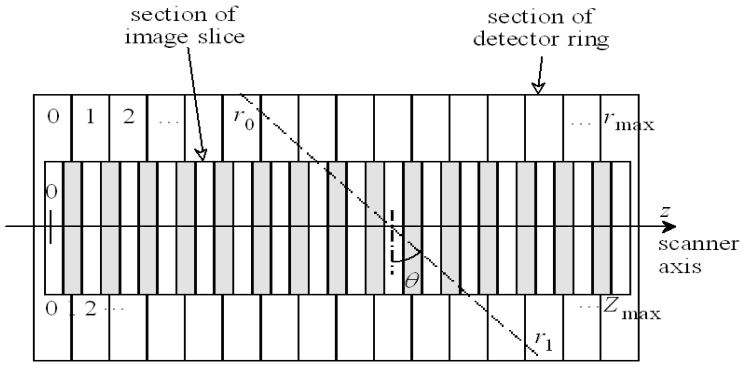

Figure 2: Sketch of main axes, units and angles used in the cylindrical scanner geometry (axial section, not to scale)

See the STIR glossary for some info on naming conventions for projection data.
In 3D PET, two data storage modes are generally used for a **segment**[1]

- where the 3D data is ordered by **sinogram** (i.e **axial position**, **view angle**, **tangential position**) (CTI terminology: *Volume* mode) (see Figure 3)
- where the 3D dataset is ordered by **view** (i.e **view angle**, **axial position**, **tangential position**) (CTI terminology: *View* mode) (see Figure 4)

This notation means that for a sinogram, the tangential position runs fastest.
In both modes, the 3D dataset has been stored in several segments where the number of axial

[1]The GE Advance file format does *not* store the data per segment, but per view. In addition, the segments are then mixed. However, once read into STIR, this organisation is no longer available.
The ECAT6 sinograms also have no easy way to understand which ring-pair corresponds to which sinogram number for 3D PET. This is the main reason why STIR does not attempt to read ECAT6 sinograms directly, but leaves this to a separate conversion utility.

positions in each **segment** depends on the **axial compression (span)**.
Note that we find the CTI terminology confusing, so you will see it nowhere in STIR except in this document.

From both modes, 2 different types of 2D datasets can be obtained :

- Sinogram i.e (View, tangential position) for a fixed axial position.
- Viewgram i.e (axial position, tangential position) for a fixed view.

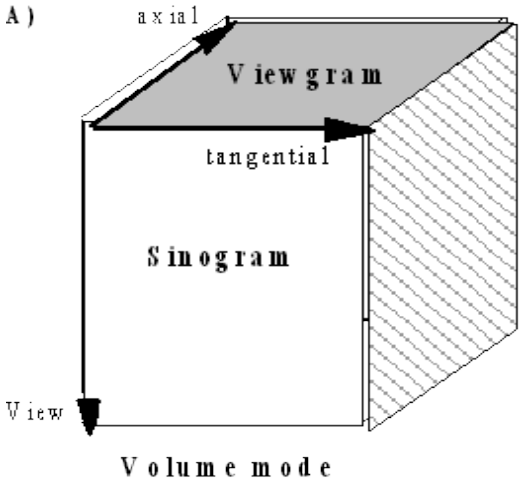

Figure 3: By sinogram storage.

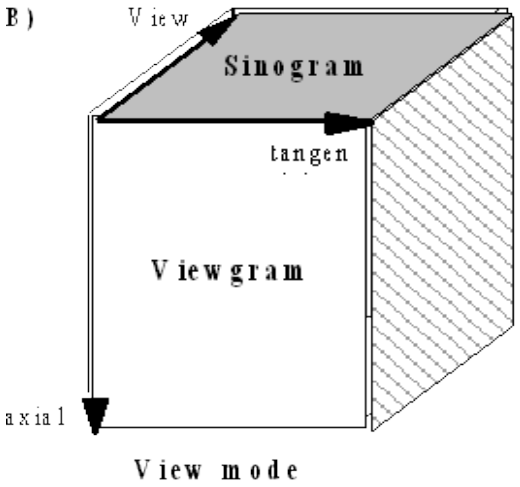

Figure 4: By viewgram storage.

For an N ring scanner, if there is *no* axial compression, there are *N-abs(segment_num)* sinograms (or axial positions) in each segment. The first sinogram always corresponds to a coincidence in the first ring of the scanner (ring_num=0), where the other ring would obviously be *ring_num=abs(ring_diff)=abs(segment_num)*.

Well ok, this is really true unless you messed around with the data and removed/added some sinograms.

The situation with axial compression is more complicated. Obviously there is more than 1 ring-difference in a segment now. This makes it kind of hard to count how many sinograms/axial_positions you will have and which ring-pairs sit in which segment. You can find some documentation on how we do this in ProjDataInfoCylindrical.cxx. See also the Michelogram documentation in the STIR glossary document.

In any case, currently STIR always assumes for each segment that the middle of the scanner corresponds to the middle of the set of sinograms (taking into account the ring difference).

View 0 currently corresponds to a vertical projection (for some scanners, this is not true. STIR currently ignores this, so will reconstruct a rotated image). Finally, in STIR code, the first ring of the scanner is the ring furthest from the bed. This corresponds to how the data are stored in ECAT6,7 and Interfile files.

3 Code conventions

3.1 Configuration file

The file *stir/common.h* contains general configuration info and tries to iron out some incompatibilities for certain compilers. If you include any STIR *.h* file, you are guaranteed to have included *stir/common.h* as well, hence it should never be included explicitly by a 'user'.

3.2 Namespace

The STIR library has its own C++ namespace: *stir*. All symbols are within this namespace. The effect of this is that conflicts with other symbols are impossible (except when somebody else is using the same namespace). STIR also uses sub-namespaces for certain things. This usage will probably be expanded in the future.
For older compilers, the namespace can be disabled (see *stir/common.h*). For this reason, we have introduced some macros such as *START_NAMESPACE_STIR*. You could use the macros in your own code as well, although people with a compiler that does not support namespace really should upgrade.

3.3 Naming conventions

Types start with capitals, every word is capitalised, no underscores, e.g. *DiscretisedDensity*.
Variables, methods and members are lower case, underscores between different words, e.g. *voxel_size*.
Variables, methods and members indicating

- a variable or member used to indicate the number of things starts 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*.
- Pointers to an object called *something_ptr*. Currently there is no naming distinction between shared pointers etc. This is probably a bad idea. For new code, we recommend to use *something_sptr* for shared pointers, and *something_aptr* for auto_ptrs (see section 3.8.2).

In most cases, access to data of a class is via a *get_something(), set_something()* pair.
In general, names are descriptive and hence long. We often take longer to decide about the name than to write the actual code. If you write new code, do the same. You will be grateful when you look back at the code a few months later (and not vilified by your successor).

3.4 File conventions

Most classes have their own *.h, .inl*, *.txx* and/or *.cxx* files. File names of such classes are simply *ClassName.h* etc. (preserving capitals). In general, looking at the *.h* file should give enough information on what a class/function does.
The *.inl* files contain inline code for functions. The main purpose for this is to keep the *.h* files as short and clean as possible.
The *.txx* files are similar to the *.inl* files. They contain non-inline definitions of template classes (or functions). They should only be included by the corresponding *.cxx* file where the template is then instantiated for any desired types. This makes it easier or a user to instantiate a template for a new type without modifying the original STIR code.[2]

All STIR include files should be included as *#include "stir/maybe_a_subdir/name.h"*
The *doxygen* comments generally occur in the *.h* file, unless they are very extensive. In this case, the *.h* file contains a brief comment, while the *.inl* or *.cxx* file contains the longer description.
To avoid unnecessary interdependencies between *.h* files (and hence unnecessary rebuilds when modifying one *.h* file), we avoid including *.h* files for 'supporting' classes as much as possible. Instead, we only declare the classes needed, e.g. *class Bin;* instead of *#include stir/Bin.h.*

3.5 Class definitions

Class definitions follow generally the following format (ignoring *doxygen* comments).

```
class A
{
public:
  enumerated types and other typedefs
  static member functions
  data members
  constructors
  destructors
  member functions
protected:
  enumerated types and other typedefs
  static member functions
  data members
  constructors
  destructors
  member functions
private:
  enumerated types and other typedefs
  friends if any
  static member functions
  data members
  constructors
  destructors
  member functions
};
```

Inline member functions are defined in the *.inl* file, and are ordered (where possible) such that if inline member function a() calls inline member function b(), then b should be defined first. (Otherwise very few compilers are able to inline the call to b from a.)

[2]The *.txx* extension as used because the Insight Toolkit (ITK) uses it as well. It stands for template C++ presumably.

3.6 Argument order conventions

When passing output arguments (by reference or by pointer), those arguments occur FIRST in the list. For instance

```
void some_function(int& output, const SomeType& some_input_argument, ...);
```

This order of arguments allows the use of default arguments. Note that the standard C++ library does not use this convention.

3.7 Error handling

Problem reporting is via two functions *error()* and *warning()*. Currently, *error()* simply aborts the programme (after writing a diagnostic message).
In many places, validity of input arguments or of the state of an object is checked by *assert* macros. This code is only compiled when the *NDEBUG* preprocessor macro is not defined, such that a production version of the programs is not slowed down. If an assertion is false, the program aborts with info on the file and line number where the assertion failed.

3.8 Advanced (?) C++ features used

We attempted to follow the ANSI C++ standard as close as possible. We expect to have some marginal problems with a 'strict' ANSI C++ compiler, although there should now be very few cases(as gcc warns about a lot of stuff). We have used some preprocessor macros (see *stir/common.h*) to isolate work-arounds for older compilers. Ideally, all these #ifdefs should disappear at a later development stage of the library. We gradually give up on supporting older compilers anyway.

The library uses templates very often. This allows us to write 'generic' code, independent of specific types. For instance, multi-dimensional arrays correspond to

```
template <int num_dimensions, typename elemT>
class Array;
```

To avoid linking problems, the templates which we use are explicitly instantiated in the relevant *.cxx* files. If a user needs other types, (s)he will have to add the instantiations (or use a more recent compiler).
We do use partial class template specialisation in some places. However, (very ugly) work-arounds are provided for compilers that do not support this feature.
Very occasionally we use member templates (in such a form that it can be compiled by Visual C++, i.e. inline in the class definition). It would be possible, but slightly tedious, to write explicit code for older compilers, but we did not do this.
Another fairly recent C++ feature that we use is Run Time Type Information (RTTI). We almost exclusively use this to check validity of pointer (or reference) casts down a hierarchy ('down-casting'). See section 8 for an example. If a compiler does not support RTTI, it would be possible to declare a (essentially empty) dynamic_cast template function such that the above code would compile. The resulting programme would become inherently type-unsafe though, and we recommend that you upgrade your compiler. Some code does rely on explicit type checking, you would have to check this in detail if you don't have RTTI.

3.8.1 Iterators

An important C++ concept, used fairly often in STIR, is an *iterator*. The easiest way to think about iterators is as a sort of generalised pointer, used to iterate through a collection of objects of the same type. For instance, the following code would add 2 to all the elements of a vector:

```
void f(vector<int>& v)
{
  for (vector<int>::iterator iter = v.begin(); iter != v.end(); ++iter)
     *iter += 2;
}
```

So, just as pointers, you increment an iterator to go to the next element of the vector, and you use **iter* to access the object that the iterator refers to. However, the advantage is that the above code would just as well work for any other type of collection, as long as it provides an iterator interface. So, typical code in C++ would look as follows:

```
template <class Container>
void f(Container& v)
{
  for (Container::iterator iter = v.begin(); iter != v.end(); ++iter)
     *iter += 2;
}
```

Iterators are used with great success in the C++ Standard Template Library (STL), and STIR provides iterators for its container classes. In particular, for multi-dimensional arrays, it is possible to iterate through the array in 2 ways: using *Array<n,T>::iterator* whose iterators point to arrays of dimensions *n-1*, or using *Array<n,T>::full_iterator* which essentially provides a one-dimensional look at the whole array. So, adding 2 to all the elements of a multi-dimensional array can be done as follows

```
template <int n, class elemT>
void f(Array<n,elemT>& a)
{
  for (Array<n,elemT>::full_iterator iter = a.begin_all();
       iter != a.end_all(); ++iter)
     *iter += 2;
}
```

Note that use of the *begin_all()* and *end_all()* members which return *full_iterator* objects.
Of course, the above function is just an illustration, as in STIR this can be done by using:

```
Array<n,elemT> a = ...;
a += 2;
```

3.8.2 Shared pointers and other smart pointers

STIR uses the *shared_ptr* class considerably. Shared pointers are a specific type of 'smart pointers'. These have the advantage that they essentially clean up after you. That is, you do not have to call *delete* on them. They are even exception proof (something which you cannot achieve with an ordinary pointer). Generally, the destructor of the smart pointer will make sure that the object it points to is deleted (when appropriate).
std::auto_ptr is a standard smart pointer class which is suitable for pointers to objects where there's only one smart pointer for each object. Unfortunately, its syntax has been decided rather late in ANSI C++ and its implementation requires some advanced C++ features, leading to a non-standard implementation of auto_ptr in older compilers (including VC 6.0). For this reason, we do not use this smart pointer class too much, and generally use shared_ptr instead. This is somewhat unfortunate as these two smart pointers are generally quite different concepts.
shared_ptr is a STIR (or boost) smart pointer class which is suitable when there are (potentially) more than one pointer pointing to the same object. It keeps a reference count such that the object is (only) deleted when the last shared pointer that references it is destructed.

Caveat: if you modify the object of 1 shared_ptr, the change obviously applies to all shared_ptrs sharing that object.
Caveat: if you assign an ordinary pointer to a shared_ptr, you cannot delete the ordinary pointer anymore (it will be done by the shared_ptr). As a consequence, you cannot assign an ordinary pointer twice to a shared_ptr. It is thus better to go all the way, and not have any ordinary pointers anymore.
Caveat: do not initialise a shared_ptr with a pointer that cannot/should not be deleted. For instance, initialising it with the address of a local variable, or even a reference, will cause a delete on an object that is not allocated on the heap, and so probably crash your program.
It is clear that smart pointers are very useful, but also somewhat dangerous. For this reason, most classes do not return shared_ptrs to their members (even if they store one). Instead, they return a pointer to a const object. This prevents a user of a class to inadvertently change members of the class-object. This does involve a performance penalty when the user then needs to create a shared_ptr himself. For example, you'll see code like this

```
ProjData proj_data(...);
ProjDataInfo const * proj_data_info_ptr =
  proj_data.get_proj_data_info_ptr();
shared_ptr<ProjDataInfo> new_proj_data_info_sptr =
  proj_data_info_ptr->clone();
// now do something with new_proj_data_info_sptr
```

We provide the convenience function *stir::is_null_ptr()* to test if a pointer is 'null', i.e. does not point to anything. *is_null_ptr* works with ordinary pointers, shared_ptrs and auto_ptrs. Use it to avoid that your code depends on what type of (smart) pointer you are using.

3.9 Generic functionality of STIR classes

This is a (very incomplete) section describing some functionality that many STIR classes have in common.

3.9.1 Copying objects

When using pointers or references to objects of an abstract type, you can copy them using `clone()`, see section 3.8.2.

When you only want to create an object of the same characteristics, but without copying the data itself (e.g. images of the same size), use `get_empty_copy()`.

3.9.2 Comparing objects

Many STIR classes implement comparison using the usual `==` and `!=` operators. The implementation of these operators is somewhat tricky in class hierarchies. We try to follow the approach described in *Overriding the C++ Operator==, An approach that uses the Template Method design pattern* By Daniel E. Stevenson and Andrew T. Phillips, Dr. Dobb's Journal, June 2003, `http://www.ddj.com/184405409`.

For some classes, you might want to check only if two objects are of the same type. For instance, if images have the same sizes, origin etc, but not necessarily the same voxel values. This can be done using the member function `has_same_characterics`.

3.9.3 Parsing from text files

Many class-hierarchies allow to construct an object by parsing a text file, which uses an Interfile-like syntax. Examples are given in the User's guide. see section 5.

All these classes (and some others) have a member function `parameter_info()` which returns a string with all parameters of the object.

Note that many STIR classes are not completely constructed by parsing. Usually, it is necessary to call a `set_up` function to make the object usable.

3.9.4 typedefs

Many class hierarchies use the same typedefs. For example:

```
class ProjDataInfo
{
protected:
  typedef ProjDataInfo root_type; // root of hierarchy

};
class ProjDataInfoCylindrical: public ProjDataInfo
{
private:
  typedef ProjDataInfo base_type;
  typedef ProjDataInfoCylindrical self_type;
};
```

4 Overview of classes

This section provides an overview of the main ingredients of the library. Detailed description of these classes is in the documentation. Below we only mention some general features.

4.1 Images

Iterative algorithms generally assume that the activity density can be discretised in some way. That is, the continuous density can be approximated by having a linear combination of some basis-functions. The reconstruction problem will try to estimate the coefficients λ_{ijk} of the discretised density

$\sum_{ijk} \lambda_{ijk}\, b_{ijk}(\widehat{x})$

The base class corresponding to this kind of data is *DiscretisedDensity.*

We assume that the set of basisfunctions can be characterised by 3 indices[3] (ijk) such that i runs over a range of integers $i_1..i_2$,j runs over a similar range that can however depend on i, and k runs over a similar range that can depend on i and j. This concept of ranges is embodied in the *IndexRange* class. Multi-dimensional arrays which have such ranges are encoded by the *Array* class. This forms the data structure for the set of coefficients of the basisfunctions, hence *DiscretisedDensity* is derived from the *Array.*

In most useful cases, the basisfunctions will be translations of a single function *b(x)* (although scaling etc could occur depending on *ijk).* This means that the discretisation has a certain grid, corresponding to the centre of the basisfunctions. This structure is the next level in the image hierarchy. Currently we have the class *DiscretisedDensityOnCartesianGrid* to implement the case where the grid is formed by an orthogonal set of vectors. Another case would be e.g. *Discretised-DensityOnCylindricalGrid* (for a cylindrical coordinate system), but we have not implemented this yet.

The next level in the hierarchy is then finally the specification of the basis functions themselves. We currently have only voxels and pixels, but another useful case would be to use Kaiser-Bessel functions as the translated basisfunction *b(x)* (this is called *Blobs* in the literature). This leads us to the example image hierarchy in Figure 5.

[3]Actually most of the image hierarchy is templated in the number of dimensions. This means it can be used for 2D and 3D images, but also for higher dimensional cases (say temporal information).

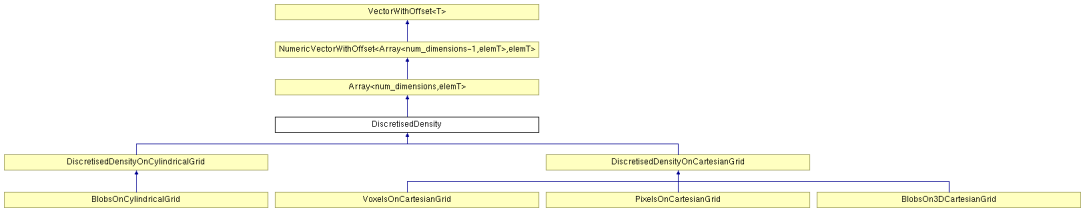

Figure 5: Hierarchy of classes for images. Only two of the bottom classes are currently implemented.

Although a Cartesian grid seems the logical choice for images, there are two reasons why one would like to use different types of grid. One important reason would be sampling efficiency. As an example, in 3D one needs less sampling points with a BCC grid while still being able to represent the same spatial frequencies. This becomes particularly important when using more complicated basis functions than voxels, like Kaiser-Bessel functions ('blobs'). In this case, calculations to perform the projections are much more expensive, so having less grid points becomes important. A second reason for changing the image grid is to increase the symmetry of the projection matrix. For example, for cylindrical scanners using a cylindrical grid would maximise the amount of repetition in the projection matrix. In particular, this would allow even for large PET scanners pre-calculation of the 'independent' part of the projection matrix, and loading it completely into memory when doing the reconstruction. This will not only speed up the reconstruction, but also enable more accurate models of the acquisition to be used, resulting in potentially better resolution and noise behaviour. We believe that this hierarchy can accommodate all known cases used for reconstructions in 3D PET.

4.2 Projection data classes

Different manufacturers use different sampling of projection space. As an example, the HiDAC uses 'polar' sampling. Similarly, Single Photon Emission Tomography (SPET) or Computed Tomography (CT) data are very similar to PET. By having projection data classes that allow for different geometries and file formats, our library should be useful for these image modalities as well.
The general class to access projection data is called *ProjData*. As in 3D-PET, the projection data are potentially huge, we do not generally store the whole data in memory. Instead, *ProjData* has methods for getting subsets of the data, i.e. *SegmentByView, SegmentBySinogram, Viewgram, Sinogram, RelatedViewgrams* (see section 4.4.1). In addition, the *ProjData* class provides access to a *ProjDataInfo* object (see Figure 6). This object completely describes the geometry of the data.

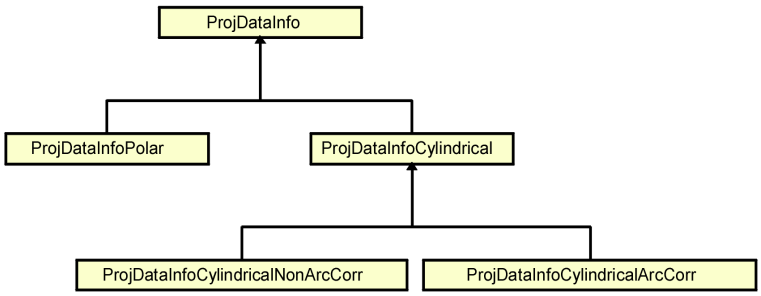

Figure 6: Hierarchy of classes for (geometric) information of the projection data.

Different file formats (or potentially other types of projection data) are handled by having derived classes of *ProjData*, which provide specific implementations for the data access methods. This hierarchy is at the moment very simple, but can easily be extended to accommodate different file formats.

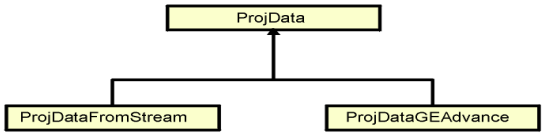

Figure 7: File formats for projection data

Currently missing is support for fan-beam data.
List-mode data is experimentally supported since version 1.2.

4.3 Data (or image) processor hierarchy

We provide a hierarchy for functions that modify an image, *e.g.* by filtering or thresholding it. The base class is *DataProcessor*, which is templated in `DataT`, to allow using other types, not only images. Aside from the virtual functions that specify the interface to an image processor, it also provides the basics for a registry of all data processors. See section 7 for some details. Note that the registry is specific to every `DataT`.

4.4 Projector classes

The next important type of ingredient for an iterative algorithm are the projection operations. The forward projection models the measurement. Generally, reconstruction algorithms are some kind of inversion procedure for the following problem

$$y_b = \sum_v P_{bv} \lambda_v$$

where y_b are the measured data. The exact interpretation of the projection matrix depends on the algorithm (it is most of the time probabilistic, in the sense that the above equation would only hold 'on average'). However, this need not concern us here.
Many algorithms (in particular EM-type algorithms) can be written solely in terms of multiplication with the projection matrix (*forward projection*) or with its transpose (*back projection*). This is why we have a *ForwardProjectorByBin* and *BackProjectorByBin* hierarchy. The basic objects handled by these projector classes are *RelatedViewgrams* and (a derived class from) *DiscretisedDensity*. Other algorithms (for instance ART or listmode algorithms) need more detailed access to the elements of the projection matrix, and hence will work with a *ProjMatrixByBin* object[4].

4.4.1 Symmetries

An important feature of geometric projectors is that several parts of the projection matrix are the same. This is because different (generalised) voxels and bins are related by symmetry. It is important to make use of these symmetries for two reasons. It allows storing only a part of the

[4]Some algorithms would need column-wise access to the projection matrix (i.e. by voxels). We did not implement any of these algorithms, so we do not have appropriate projection classes for them. Modification of the row-wise classes is straightforward though.

projection matrix (useful for disk storage and caching), and in on-the-fly projection operations it can be used to speed up the computation, as there is no need to recompute the related elements. As the number of projection data elements related by symmetry depends on the specific geometries (e.g which class derived from *DiscretisedDensity* is used), we need classes (e.g. *RelatedViewgrams*) that store all related data, classes for describing the symmetries (e.g. *DataSymmetriesForViewSegmentNumbers*), and the necessary operations (*SymmetryOperation*).
At the moment, the only symmetries implemented are specific for Cartesian grids. A discussion of these symmetries is presented in [Par4.1], but also in the online documentation.

4.4.2 ForwardProjectorByBin hierarchy

We provide two derived classes of *ForwardProjectorByBin*:

- *ForwardProjectorByBinUsingRayTracing* computes the P_{bv} elements as the length of intersection of one Line Of Response with the voxel. The actual implementation uses a version of Siddon's algorithm, enhanced to use all possible symmetries. This implementation was discussed in [Par4.1].
- *ForwardProjectorByBinUsingProjMatrix* performs forward projection for any *ProjMatrix*. Indeed, all functionality is already provided by the *ProjMatrix* class. The current class only provides the implementation of the *ForwardProjectorByBin* interface, such that algorithms can use any *ProjMatrix* object available.

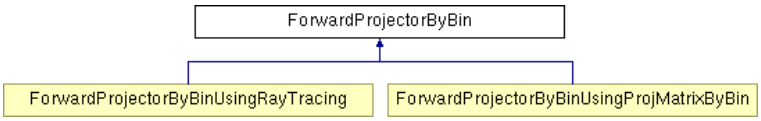

Figure 8: Forward projectors

This hierarchy is uses the registry mechanism discussed in section 7.

4.4.3 BackProjectorByBin hierarchy

This is very similar to the previous section. We provide two derived classes of *BackProjectorByBin*:

- *BackProjectorByBinUsingInterpolation* computes the P_{bv} elements via interpolation between the projection data for the LOR through the centre of the voxel. We have two different interpolation mechanisms (linear, and piece-wise linear) as discussed in [Par4.1].
- *BackProjectorByBinUsingProjMatrix* performs back projection for any *ProjMatrixByBin*.

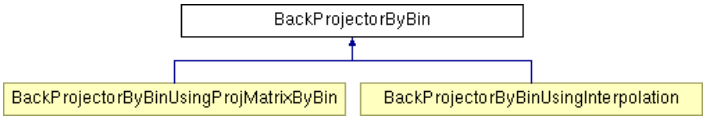

Figure 9: Back projectors

This hierarchy is uses the registry mechanism discussed in section 7.

4.4.4 ProjMatrixByBin hierarchy

This is a base class for row-wise access to the projection matrix P_{bv}. This class provides 2 essential mechanisms aside from the virtual functions that will be used to get the row of the matrix:

- It can cache the elements of the projection matrix. This caching is obviously useful if you can store the part of the matrix that you need in memory (for instance, a slave might not need the whole matrix). However, even if you cannot store the whole matrix, most algorithms need access to a subset of these elements more than once (for instance, OSEM would need them for forward projecting and for back projecting). Caching can be disabled.
- The application of symmetries is provided at base-class level: the derived classes do not have to bother about this, and can concentrate on computing the 'independent' part of P_{bv}.

At the moment, we have two derived classes:

- *ProjMatrixByBinUsingRayTracing* uses essentially the same implementation as *ForwardProjectorByBinUsingRayTracing*, but returns the row of the projection matrix. However, because symmetries are not handled 'in-line', and elements need to be stored, using a *ProjMatrixByBinUsingRayTracing* object for forward projection is not as efficient as using *ForwardProjectorByBinUsingRayTracing*.
- *ProjMatrixByBinFromFile* handles the case where the projection matrix is stored on disk [not distributed yet]. Because of the support provided by *ProjMatrixByBin*, only the 'independent' part of the projection matrix needs to be stored. Our current implementation does not yet provide a very compact format for storing the elements (although they are of course stored sparsely).

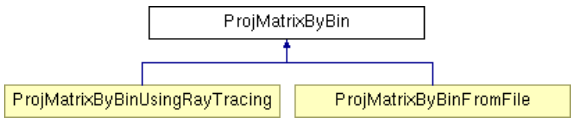

Figure 10: Projection matrix classes

This hierarchy uses the registry mechanism discussed in section 7.

4.4.5 ProjMatrixElementsForOneBin

This class provides sparse storage for a row of the projection matrix. It has methods to access the data using an STL-style iterator (which is essentially a generalised pointer). This means that the actual way to store the data is hidden from the user. In principle, this format could be very compact, or alternatively very efficient. At the moment, we provide a format that is somewhere in between. For instance, image indices are not stored incrementally, as although it would allow very compact storage, it is detrimental for speed.

4.5 Objective functions

Many iterative (reconstruction) algorithms can be formulated in terms of an objective function, which is the algorithms tries to maximise (or minimise). Since STIR 2.0, this is implemented as a class hierarchy based on `GeneralisedObjectiveFunction`, where we use the convention that the objective function is maximised.

Some iterative algorithms use an 'objective function' only in a loose sense. They might for instance allow generalisations which no longer optimise a function. For example in the case of EMML with non-matching forward and back projectors, the 'gradient' that is computed is generally not the gradient of the log-likelihood that corresponds to the forward projector. However,

one hopes that it still points towards the optimum. The corresponding objective function is implemented in the class `PoissonLogLikelihoodWithLinearModelForMeanAndProjData`.

There can be different objective function that use common operations. For instance, the objective function could implement a least squares criterion, or a Poisson log-likelihood. Both of these have a model for the mean of the measured data (given an image). Generally speaking, STIR implements those common operations using a separate class. As an example, projection operations are implemented via a pointer to a `ProjectorByBinPair`, which in itself is a small class hierarchy using either a `ProjMatrixByBin` object, or a `ForwardProjectorByBin/BackProjectorByBin` pair.

Often, one includes a penalty (or prior) in the objective function (see doxygen documentation for the class `GeneralisedPrior`). The penalty is expected to be a function that increases with higher penalty, so it will be *subtracted* from the unregularised case.

See the doxygen documentation `GeneralisedObjectiveFunction` for more details.

4.6 Reconstruction classes

The final ingredient for performing reconstructions is the reconstruction algorithm itself. We also have a hierarchy for this, as many iterative algorithms are similar, and differ only in some minor steps. We do not discuss the actual 'leaves' of the Reconstruction tree here.
The tree starts at the abstract Reconstruction base class. Classes ending in "*Reconstruction*" are abstract base classes for a family of algorithms. Classes including the name of an algorithm are concrete ones whose principal purpose is to implement the reconstruction method of the algorithm.

Certain algorithms will only work with particular objective functions, while others (such as gradient ascent) might work with arbitrary objective functions.

This arrangement gives the users of the library a flexible range of implementation choices, allowing them to experiment with new algorithms and new variations of old ones with minimal coding effort.

The `Reconstruction` hierarchy is (since STIR 2.0) templated in `TargetT`. This is the type of the output, i.e. normally the image type. Templating this type gives flexibility to have different output-types, such as parametric images or even normalisation factors.

5 Parsing from text files (or strings)

Many STIR classes are based on the `ParsingObject` class, and hence are able to

- ask the parameters to the user interactively
- read the parameters from an Interfile-style file
- return parameter info such that an Interfile-style input file can be created from the current set of parameters.

In the simplest case, the only thing you need to do is to add the variables and keywords to the keymap. All functionality will then follow. See the following example.

```
class A : public ParsingObject
{
  private:
  typedef ParsingObject base_type;

  int number; float a;

  void initialise_keymap()
  {
    base_type::initialise_keymap();
    this->parser.add_start_key("My Class Parameters");
```

```
    this->parser.add_key("number", &number);
    this->parser.add_key("width", &a);
    this->parser.add_stop_key("END My Class Parameters");
  }

  void set_defaults()
  {
    base_type::set_defaults();
    // specify defaults for the parameters in case they are not set.
    number = 1;
    a = 2.4F;
  }
};

int main(int argc, char **argv)
{
  A a;
  // parse a file (first argument on the command line)
  a.parse(argv[1]);
  // list them back to cout
  std::cout << a.parameter_info();
  return EXIT_SUCCESS;
}
```

Check the doxygen documentation for `ParsingObject`.

6 IO

6.1 Images

For images, output file formats and (since version 2.0) input file format are represented by objects as well, as instantiations of classes derived from `OutputFileFormat` and `InputFileFormat` respectively. All of the file formats supported by STIR are stored in registries (see section 7).

Currently recommended way to read an image is as follows:

```
typedef DiscretisedDensity<3,float> DataType ;
std::auto_ptr<DataType> density_aptr = read_from_file<DataType>(filename);
```

You will have to include `stir/IO/read_from_file.h` for this to work. The function `read_from_file` will open the file, read an initial block of data, check if that corresponds to the signature of any of the file formats in the default registry, and use the first matching file format to read the data.

For output, you can use

```
typedef DiscretisedDensity<3,float> DataType ;
shared_ptr<OutputFileFormat<DataType > output_format_sptr =
   OutputFileFormat<DataType >::default_sptr();
output_format_sptr->write_to_file(filename, density);
```

You will have to include `stir/IO/OutputFileFormat.h` for this to work. Currently, the default output file format in STIR is Interfile (see the User's Guide).

At present, STIR does not provide a way to open an image file for reading and writing. Patient/study information etc is currently ignored. More dangerously, image orientation as well.

It is relatively straightforward to add a new file format by deriving a new class from `InputFileFormat` (for input), and adding that to the default registry (see section 7 for more details).

6.2 Numerical arrays

There are some functions to read/write numerical vectors/arrays from file. Check `stir/IO/read_data.h` (and `write_data.h`) for binary IO, and `stir/stream.h` for text-based IO.

6.3 Projection data

At present, there is no registry yet for file formats of projection data. For reading, use

```
shared_ptr<ProjData> proj_data_sptr = ProjData::read_from_file(filename);
```

This has an optional argument for read/write access to the file.

For writing, you could use

```
ProjDataInterfile proj_data(proj_data_info_sptr, filename);
```

We currently have no easy way to copy all data across from one `ProjData` object to another. You will have to write an explicit loop over segments.

6.4 Dynamic or gated data

STIR 2.0 does not provide any direct facilities for reading 4D data (aside from the ECAT conversion utilities). Basic support should come in version 2.1. In the mean time, there is some preliminary functionality via the `TimeFrameDefinitions` class and the `get_time_frame_info` utility.

7 Registries of classes

Several class hierarchies in STIR keep a registry of all *leaf* derived classes (i.e. classes at the end of the hierarchy). This registry allows the user to select *at run-time* which image processor, projector, matrix, output file format etc. (s)he wants to use. It is fairly easy to add your own leaf class to the hierarchy such that it will be added to the registry. How to do this needs more documentation, but see the doxygen documentation of classes *RegisteredObject* and *RegisteredParsingObject*.
The recommended way to add your own class (as always) is to take a look at an existing leaf in the hierarchy, copy the files and change what you need. Also make sure you new class gets linked in by adding it to one of the *registries.cxx files.
See also section 9.

8 Using class hierarchies

The main strength of object-oriented programming is that you can write code that is as generic as possible[5]. In STIR, we have tried to use this feature as much as possible (although there are still remaining generalisations to be implemented). This works by having a base class and a bunch of derived classes. Examples of class hierarchies are given above. The question is however how you use these hierarchies. Some attempt is given here to answer this, but of course, for serious work, you really should read a good C++ book.
We will use the DiscretisedDensity hierarchy as an example, but the same holds for nearly all other hierarchies in STIR (or indeed C++). In addition, you will see some templates being used.
Rule 1: You should try to write your code using the type of the base-class. If this is not possible, use a class as close to the base-class as you can.
Example: If you have a function that manipulates images, write it in terms of DiscretisedDensity, not of VoxelsOnCartesianGrid.
Rule 2: Pass references or smart pointers as arguments of functions. Use references if you

[5] In C++, another very powerful way to write generic code are templates. But you will have to read about those somewhere else.

can (such that your code would work for any type of pointer). Never (or almost never) return references, but return smart pointers (or a const pointer to an object that is guaranteed to stay alive long enough).
Example:

```
template <int num_dimensions, typename elemT>
shared_ptr< DiscretisedDensity<num_dimensions, elemT> >
do_something(const DiscretisedDensity<num_dimensions, elemT>& density1,
             const DiscretisedDensity<num_dimensions, elemT>& density2);
```

This is necessary, as you generally cannot construct an object of the type of the base-class. Similarly, it is impossible to copy an object of a base-class. This is solved by using the *virtual copy constructor* idiom:

```
shared_ptr< DiscretisedDensity<3,float> > new_density_sptr =
     density.clone();
```

This constructs an object of the same type as the given density.
Of course, many functions do need to know the actual type of the parameters, either because they can work with only one type (for instance, ForwardProjectorByBinUsingRayTracing handles only VoxelsOnCartesianGrid), or because the processing will be different for different types. This can be solved by using C++ Run Time Type Information (RTTI), preferably in the form of dynamic_cast.

```
template <typename elemT>
void f(DiscretisedDensity<3, elemT>& density)
{
  VoxelsOnCartesianGrid<elemT> * voxels_ptr =
     dynamic_cast< VoxelsOnCartesianGrid<elemT> * > (&density);
  if (voxels_ptr == NULL)
     error("f: can only handle images of type VoxelsOnCartesianGrid\n");
  ...
}
```

Note that dynamic_cast of pointers allows easy checking if the types matched. When using references, a non-matching type will cause an exception to be thrown. There currently is no code in STIR using/catching exceptions, so if you do not catch the exception yourself, your program will abort mysteriously.

9 Extending STIR with your own files

If you want to add your own files to the libraries, or have your own main programs, you can of course set-up your own directory and

- make sure that your compiler finds the STIR include files in **STIR/include**
- make sure that you link with **STIR/opt/libstir.a** (replace **opt** with whatever value you used for the *DEST* option of make)
- make sure that you link with the *_registries

All of this is made easier by using
make install-all INSTALL_PREFIX=/my/favorite/location
Nevertheless, wipping up your own makefile is slightly painful, so we provide the following alternative.
Put all your files in (subdirectories of) **STIR/local** (the distribution will never contain any files

in that directory)
Create a file **STIR/local/extra_dirs.mk** containing a list of your subdirectories (see **STIR/samples/local/** for an example). This file (if it exists) will be imported by **STIR/Makefile**. Put in each of your subdirectories one or more files called **lib.mk**, **exe.mk**, **test.mk** which list all files you want to compile. These files have to be in a specific (but simple) format. The easiest way to do this is to copy for instance **buildblock/lib.mk**, **utilities/exe.mk**, **test/test.mk** and change the list of filenames.
The result will be that, when you use the relevant *make* command in the **STIR** directory,

- all your files listed in one of the **lib.mk** will be included in **libstir.a**
- all your executables will be build and/or installed
- all your test programs will be run

10 Conclusion

In many cases, a reconstruction algorithm does not have to know about the type of the image it gets. In our set-up this means it can/should be implemented in terms of a *DiscretisedDensity* object, and *ForwardProjectorByBin* and *BackProjectorByBin* objects (or *ProjMatrixByBin* object). This means that once for instance EM is implemented, it can be used on voxels, blobs, different grids etc. without having to rewrite the algorithm itself (of course, the projectors do depend on the actual type of image, so you would have to write those).
The STIR library allows maximal code re-use when implementing new functionality. Its structure and its documentation features make it into a unique resource for the PET community.