Università degli Studi di Sassari

XRMC

The definitive manual

Version 6.4.4

Bruno Golosio
Tom Schoonjans
Antonio Brunetti
Giovanni Luca Masala
Piernicola Oliva

March 19, 2014

Contents

1 Introduction

XRMC is a Monte Carlo program for accurate simulation of X-ray imaging and spectroscopy experiments in heterogeneous samples. The use of the Monte Carlo method makes the code suitable for the detailed simulation of complex experiments on generic samples. Variance reduction techniques are used for reducing considerably the computation time compared to general purpose Monte Carlo programs. The program is written in C++ and has been tested on Linux, Mac OS X and MS Windows platforms.
XRMC is released under the terms of the GPLv3.
Unfortunately we cannot provide any other support apart from the information contained within this manual, mostly because we do not have the resources to do so. Please do not write us emails asking for help with your simulations, or for explaining unexpected results.
As is true for all non-trivial software, XRMC is not free of bugs. We try and fix bugs in every released version. In order to do this, we would like to ask all users to send us their bugreports, which should consist of the input-files that provoke the bug. Send reports to golosio@uniss.it.
The software package is being developed at the University of Sassari by Bruno Golosio, Tom Schoonjans, Antonio Brunetti, Giovanni Luca Masala and Piernicola OlivaA.
We have recently published a manuscript covering most of the features implemented in XRMC: B. Golosio, T. Schoonjans, A. Brunetti, P. Oliva, and G. L. Masala. Monte Carlo simulation of X-ray imaging and spectroscopy experiments using quadric geometry and variance reduction techniques. Computer Physics Communications, 2013.
You are kindly requested to include this paper in the reference list of your published work when you would decide to use XRMC for scientific purposes.

2 Installation instructions

2.1 Compiling from source

The source code can be downloaded from the **XRMC downloads** repository.
In order to compile the software from source, please ensure that you have xraylib installed, as well as a C++ compiler with OpenMP support (v3.0 or higher).
We are currently extending the functionality of the program through the use of plug-ins. So far one plug-in has been written that allows for the simulation of ED-XRF detector response functions. Support for this plug-in requires that XMI-MSIM is installed at compile time.
After downloading the tarball, unpack and compile the source code using the following commands:

```
gunzip xrmc-version.tar.gz
```

```
tar xvf xrmc-version.tar

cd xrmc-version

./configure

make
```

Optional, but recommended is to check if the compilation went well:

```
make check
```

Finally installation is performed with:

```
make install
```

The configure command takes several options (execute `./configure --help` to see a full list), the most important one being `--prefix=`, which is used to set the installation directory. The default value is `/usr/local`, which in most cases will require administrative privileges to install into.
On some Linux systems (RedHat based), you may have to run `ldconfig` after installation in order to have the xrmc executable link to the required shared libraries at runtime, provided that the xrmc shared library is installed in a location that is mentioned in `/etc/ld.so.conf`.
Otherwise you may have to adjust the LD_LIBRARY_PATH environment variable to correspond to the location where the xrmc shared library is installed (`${prefix}/lib64` or `${prefix}/lib` in most cases).
After installation, if the shell responds to the invocation of `xrmc` with a `command not found` error, you will have to modify the PATH environment variable to include the installation location of the xrmc executable, which should be `${prefix}/bin`

2.2 Windows installer

An installer has been written that will facilitate deployment on the Windows operating system. Currently the included binaries are 32-bit only. The installation wizard will download and install its dependencies if necessary and/or required (xraylib and XMI-MSIM).
The Windows installer can be downloaded from **XRMC downloads** repository.
After installation, the user can launch simulations from the command-line through executing.

```
xrmc.exe inputfile
```

The software can be easily removed using the Add/Remove Software utility in Control Panel

2.3 Linux

2.3.1 Fedora, Centos and Scientific Linux

To facilitate the installation on RPM based Linux distributions, the package includes a spec file which can be used to produce RPM packages for linux distributions that support them (Fedora, Red Hat etc). The developers have built 64-bit RPM packages of XRMC for the Fedora 16-20 and Redhat EL/CentOS/Scientific Linux 6 distributions. These can be downloaded from the RPM repository that is hosted by the X-ray Microspectroscopy and Imaging research group of Ghent University. Access to this repository can be obtained as follows for Fedora distros:

```
su -c 'rpm -Uvh http://lvserver.ugent.be/yum/xmi-repo-key-fedora.noarch.rpm'
```

and for Red Hat EL based distributions:

```
su -c 'rpm -Uvh http://lvserver.ugent.be/yum/xmi-repo-key-redhat.noarch.rpm'
```

The XRMC packages themselves can then be downloaded using yum:

```
su -c 'yum install xrmc'
```

Updates can be installed in a similar way:

```
su -c 'yum update xrmc'
```

The XMI-MSIM plug-in can be installed with:

```
su -c 'yum install xrmc-xmimsim'
```

2.3.2 Debian and Ubuntu

Packages were created for Debian and Ubuntu. Currently the following flavors are supported: Debian Squeeze and several Ubuntu versions.
In order to access these packages using your favorite package manager, execute the following command to import our public key:

```
curl http://lvserver.ugent.be/apt/xmi.packages.key |
sudo apt-key add -
```

Next, add the package download location corresponding to your distribution to the /etc/apt/sources.list file (as root):
Debian Squeeze:

```
deb http://lvserver.ugent.be/apt/debian squeeze stable
deb-src http://lvserver.ugent.be/apt/debian squeeze stable
```

Ubuntu Precise 12.04:

```
deb [arch=amd64] http://lvserver.ugent.be/apt/ubuntu precise stable
deb-src http://lvserver.ugent.be/apt/ubuntu precise stable
```

Ubuntu Quantal 12.10:

```
deb [arch=amd64] http://lvserver.ugent.be/apt/ubuntu quantal stable
deb-src http://lvserver.ugent.be/apt/ubuntu quantal stable
```

Ubuntu Raring 13.04:

```
deb [arch=amd64] http://lvserver.ugent.be/apt/ubuntu raring stable
deb-src http://lvserver.ugent.be/apt/ubuntu raring stable
```

Ubuntu Saucy 13.10:

```
deb [arch=amd64] http://lvserver.ugent.be/apt/ubuntu saucy stable
deb-src http://lvserver.ugent.be/apt/ubuntu saucy stable
```

When the sources.list file contains the correct download locations, update the apt cache by running:

```
sudo apt-get update
```

After this, one can install XRMC by executing the following command:

```
sudo apt-get install xrmc
```

The XMI-MSIM plug-in can be installed by executing

```
sudo apt-get install xrmc-xmimsim
```

2.4 Mac OS X

A request has been submitted to Macports for inclusion of XRMC into their repository. Installation instructions will be given when the package has been accepted. Until then, please compile XRMC from source.

3 User guide

3.1 Getting started

This section contains some basic recommendations for new XRMC users.

- It can be useful to start working with existing examples, which are described in section 6. You should select the example that is the closest to the experimental conditions that you want to simulate. First, run the example as is and analyze the obtained results. Next, modify the example in order to replicate the desired simulation conditions. It is better that you modify one device file at a time, so that in case something does not work as expected, it should not be too hard to track down the origin of the problem.

- The current version of XRMC does not include tools to visualize the experimental setup. However, you can use the program itself to visualize radiographic projections of the sample. Even if you want to run a simulation with a single element detector (for instance if you are simulating a spectroscopy measurement) it can be useful to define also a second source and an array detector, such as those used in the imaging examples, loaded by a second input file, used only for visualization purpose. This way you can verify the sample geometry.

- If the sample geometry does not look as expected, try to visualize only one three-dimensional object at a time. You can do this easily by commenting all the lines corresponding to the other objects in the *geom3d* file, as described in the section covering the three-dimensional object geometry description file.

- When defining the geometric description of the sample, pay particular attention to XRMCs limitations in defining three-dimensional objects, as outlined in a later section.

3.2 Using the program

Before running XRMC, you must prepare the input files that describe the experimental setup that you want to simulate. Those files include a main input file, a parameter file and some device files. The term device refers to a C++ object that is created to be used by the simulation and whose parameters are loaded from the corresponding file. It is not necessarily a physical device. For instance, the phase (material) array and the sample are considered to be devices. Each device file consists of a header that specifies the type and name of the device, a list of commands or variable names, and finally the ``End'' command. The order of commands/variable names is generally not important, unless they are part of the same group (for instance, the atomic number and weight percent of the elements in a material), as described in the following paragraphs. Each command/variable name can be followed by one or more arguments (real values, integer values or strings, depending on the command). Command/variable names need not to be in the same line. Comments can be inserted in all input files by preceding them by any of the following characters: `;', `!' or `#'. XRMC is case sensitive. Most commands and variable names in the input files follow the CamelCase convention, i.e. the first let-

ter of each concatenated word is capitalized. The following paragraphs describe the commands that can be used in all device files used by the program. Although all variables have a default value, and therefore most commands are not mandatory, it is strongly recommended to explicitly assign a value to all variables in the input files.

3.2.1 The main input file

The main input file specifies the commands for loading all device files, the command for running the simulation and the command for saving the output in a file.
A device is created, and its parameters are loaded from a file, through the command:

- Load *filename*

where *filename* is the name of the device file.
The typical setup for x-ray imaging/spectroscopy simulations include the following devices:

1. a source (*source* device);
2. a spectrum (*spectrum* device);
3. a detector (*detectorarray* device);
4. a sample (*sample* device);
5. a phase (material) composition array (*composition* device);
6. a quadric array (*quadricarray* device);
7. a three-dimensional object geometry description (*geom3d* device).

The associated input files are described in the following sections.
The order of the ``Load'' commands is not relevant.
After such commands, the simulation can be started by the command

- Run *devicename*

where *devicename* corresponds to the name of the device that controls the acquisition of the simulation results, i.e. in x-ray imaging/spectroscopy simulations the name of the *detectorarray* device, as specified in the header of the corresponding file.
Finally, the result of the simulation is saved by the command

- Save *devicename dataname filename*

where *devicename* has the same meaning as described for the previous command, *dataname* is a name associated to the data that should be saved, and *filename* is the name of the file used to save the results. The allowed entries for *dataname* depend on the device; for the *detectorarray* device the only allowed entry is *Image*.

3.2.2 The spectrum file

The source spectrum is modeled as the sum of two components: a set of discrete lines and a continuous component. The radiation can be unpolarized, partially polarized or totally polarized. The lines can have a Gaussian or a Dirac δ distribution (the latter one being simply a particular case of Gaussian distribution with $\sigma = 0$). Each line is specified by its mean energy E_l, by its intensity I_l and by σ_l.
In case of (partially or totally) polarized radiation, the intensity of the two components polarized along the local x and y directions are specified separately for each line.
The continuous component is defined by N samples at arbitrary energies E_1,... , E_N, by specifying for each sample the corresponding height of the spectral distribution I_1,... ,I_N. The height of the distribution in the interval between two consecutive energies of the sample E_i, E_{i+1} is approximated by a linear function of E that goes from I_i to I_{i+1}, therefore the spectrum in each interval between two consecutive samples has a trapezoidal shape. The area of the trapezium $(E_{i+1} - E_i)\times(I_i + I_{i+1})/2$ represents the intensity of the interval. In case of (partially or totally) polarized radiation, the height of the x and y components are specified separately for each sample of the continuous component.
There are two possible ways of extracting the initial energy of x-ray photons produced by the source:

1. extract random energies on the whole spectrum: the photon initial energy is extracted using the whole spectrum itself as a probability distribution;

2. loop on all lines and on all intervals of the spectrum: a loop is made on all lines and on all intervals of the spectrum; the initial energy of the photon is extracted according to the probability distribution limited to the single line or to the single interval; the event is assigned a weight proportional to the line/interval intensity.

The first manner corresponds to the traditional Monte Carlo approach. Lines or regions of the spectrum with lower intensity are less represented in the generated statistic, no matter how important is the contribution that they give to the detected signal. There can be some drawback in this approach. For instance, if the spectrum has a relative low intensity at higher energies and if the sample is a strongly absorbing object, the statistic of events with higher energies will be low even though they give the most important contribution to the detected signal.
The second approach is closer to deterministic integration methods and should usually be preferred to the first one. All lines and all interval are equally represented

in the generated statistic, and their relative probability is corrected by using the method of weighting the event.
If the second method is chosen, than for each interval of the continuous component there are two possible ways of extracting the photon energy:

1. extract the energy randomly according to the probability distribution inside the interval itself (which is modeled by a linear function, as discussed previously);

2. force the photon energy to be equal to the central energy of the interval.

The second method is the pure deterministic approach, and normally it should not be used.
The program offers the opportunity to resample the continuous component after its definition. In this case, the user must specify the starting energy, the energy step and the number of points used for the resampling. The intensities Ii are then recalculated for the new values of E_i. Normally this option will not be used, however it may be useful for variance reduction if the space between sampling energies in the continuous component definition is too large and a finer separation is desired or, in the opposite case, if the energy step in the initial definition of the spectrum is unnecessarily too small.

spectrum device file header

- Newdevice spectrum

- name (string)

Commands

- PolarizedFlag *val*
  - specifies is the beam is polarized;
  - *val* = 0: unpolarized beam;
  - *val* = 1: polarized beam;

- LoopFlag *val*
  - energy extraction method:
  - *val* = 0: extract extract random energies on the whole spectrum;

  - *val* = 1: loop on all lines and sampling points;

- ContinuousPhotonNum *val*
  - *val* (integer): multiplicity of events for each interval in spectrum;

- LinePhotonNum *val*
  - *val* (integer): multiplicity of events for each line in spectrum;

- RandomEneFlag *val*
  - enable/disable random energy in each interval of the continuous spectrum;
  - *val* = 0: random energy disabled;
  - *val* = 1: random energy enabled (recommended);

- Lines
  - specifies the discrete energy lines of the spectrum;
  - energies and σ are expressed in keV:

- N_l (integer): Number of lines in the spectrum

- E_1 σ_1 I_1 (real values): energy, width (rms) and intensity of the 1st line

- …

- E_{Nl} σ_{Nl} I_{Nl} (real values): energy, width (rms) and intensity of the N-th line
  - for polarized beams, or

- E_1 σ_1 I_{x1} I_{y1} (real values): energy, width (rms) and intensities of the two polarization components of the 1st line

- …

- E_{Nl} σ_{Nl} I_{xNl} I_{yNl} (real values): energy, width (rms) and intensities of the two polarization components of the N-th line

- ContinuousSpectrumFile

  – continuous component of the spectrum;

- N_l (integer): Number sampling points in the continuous spectrum

- *filename*: name of the file containing the continuous spectrum;

  – in the case of unpolarized beam, the file has the following format:

- E_1 I_1 (real values): energy and intensity of the 1st sampling point

- …

- E_{Nl} I_{Nl} (real values): energy and intensity of the N-th sampling point

  – while for polarized beam it has the following format:

- E_1 I_{x1} I_{y1} (real values): energy and intensities of the two polarization components of the 1st sampling point

- …

- E_{Nl} I_{xNl} I_{yNl} (real values): energy and intensities of the two polarization components of the N-th sampling point

- ContinuousSpectrum

  – same as ContinuousSpectrumFile, but the spectrum is loaded inline rather than from an external file. Clearly, the *filename* parameter is not used in this case

- Resample *val*
  - *val* = 0: do not resample continuous spectrum;
  - *val* = 1: resample continuous component of the spectrum;
  - If *val* = 1, the following arguments must also be provided:
- N_R (integer): number of resampling points;
- E_{min} (real): minimum resampling energy (keV);
- E_{max} (real): maximum resampling energy (keV);
- End
  - End of file.

The total number of generated events is given by the product of the multiplicities in the spectrum, in the interactions with the sample and in the detector pixels.

3.2.3 The source file

The current version of the code assumes that the x-ray beam is produced by a point source or by an extended source with a three-dimensional Gaussian distribution. A local coordinate system is associated to the source, specified by the vector position of the origin rs and by the orthonormal unit vectors $\mathbf{i}_s$, $\mathbf{j}_s$ and $\mathbf{k}_s$ (see Fig. 1), which are the directions of the local x, y, and z axis, respectively. The local z axis represents the main source direction, while the local x and y axes are used to define the beam polarization and angular aperture. Let θ_s and φ_s be the polar and azimuthal angle, respectively, relative to the source coordinate system. The user can specify the angular apertures θ_x and θ_y in the x and in the y direction, respectively. In general, the angular aperture of the source is elliptical and it is defined by the inequality:

$$\theta^2 \leq \theta_x^2 \cos^2 \phi_s + \theta_y^2 \sin^2 \phi_s \tag{1}$$

The source intensity distribution is assumed to be uniform within the solid angle limited by this angular aperture.

source device file header

- Newdevice source
- *name* (string)

Commands

- SpectrumName *name*
  - *name* (string): spectrum input device name;
- X *val1 val2 val3*
  - *val1 val2 val3* (real values) source x, y, z coordinates
- uk *val1 val2 val3*
  - source orientation: $\mathbf{k}_s$ components (local z axis direction, i.e. main source direction);
  - *val1 val2 val3* (real values): k_{sx}, k_{sy}, k_{sz}
- ui *val1 val2 val3*
  - source orientation: $\mathbf{i}_s$ components (local x axis direction);
  - *val1 val2 val3* (real values): i_{sx}, i_{sy}, i_{sz}
- Divergence *val1 val2*
  - beam divergence;
  - *val1 val2* (real values): θ_x, θ_y
- Size *val1 val2 val3*
  - Source size; the source is modeled as a three-dimensional Gaussian distribution:

- *val1 val2 val3* (real values): σ_x, σ_y, σ_z;

- Rotate *val1 val2 val3 val4 val5 val6 val7*
  - rotation of the source around the axis passing through $\mathbf{x}_0$ and with direction $\mathbf{u}$:
  - *val1 val2 val3* (real values): x_0, y_0, z_0;
  - *val4 val5 val6* (real values): u_0, u_0, u_0;
  - *val7* (real value) rotation angle θ (degrees);

- End
  - End of file.

If $\mathbf{i}_s$ is not perpendicular to $\mathbf{k}_s$, then xrmc will replace it with a vector parallel to the same plane, but perpendicular to $\mathbf{k}_s$. If $\mathbf{k}_s$ or $\mathbf{i}_s$ are not unit vectors then they will be normalized by the program. The vector $\mathbf{j}_s$ is computed by the program, so that $\mathbf{i}_s$, $\mathbf{j}_s$ and $\mathbf{k}_s$ constitute an orthonormal basis set.

3.2.4 The detector file

In general, the program can simulate two-dimensional array detectors with energy binning for each pixel. A single element detector can be simulated as a special case of array detector with only one pixel. The pixel shape can be defined as rectangular or elliptical. The latter possibility is particularly useful when a round, single element detector has to be simulated.
A local coordinate system is associated to the detector, specified by the vector position of its geometric center rd and by the orthonormal unit vectors $\mathbf{i}_d$, $\mathbf{j}_d$ and $\mathbf{k}_d$ (see Fig. 1), which are the directions of the local x, y, and z axis, respectively. The local z axis is perpendicular to the detector surface, while the local x and y axes are parallel to the detector rows and columns, respectively.
According to a variance reduction technique used in the code, each event is forced to end with a photon that reaches a pixel of the detector (and its weight is multiplied by a proper probability factor). The intersection between the last photon trajectory and the pixel can be a random position on the pixel surface, or it can be forced to be the midpoint of the pixel. The latter method is a *purely deterministic* approach, and normally it should not be used.
The user has the possibility to simulate a statistical uncertainty on pixel counts based on Poisson statistics. In the traditional Monte Carlo approach the number of

photons detected by each channel of each pixel is always an integer number. Using variance reduction techniques the events are weighted, therefore the estimated number of detected photons is a real number. However it is possible to round it to the closest integer number.
The weight associated to the probability that the last photon of an event reaches a detector pixel is proportional to a geometric efficiency factor ε, which is related to the solid angle from the interaction point to the pixel surface. If the last interaction occurs at a distance from the pixel that is comparable or smaller than the pixel size, ε can become very large. In order to avoid spikes in the signal on single pixels, it is useful to restrict ε. The default value is 2π.
Two acquisition modalities are available:

1. fluence: each channel simply counts the number of photons that it detects;

2. energy fluence: each channel sums up the energy of the photons that it detects.

The energy response of the detector can eventually be taken into account by using the first modality with a sufficient number of energy bins and by a proper postprocessing of the acquisition.

detectorarray device file header:

- Newdevice detectorarray

- *name* (string)

Commands

- SourceName *name*
  - *name* (string): source input device name;

- NPixels *val1 val2*
  - pixel number ($N_x \times N_y$);
  - *val1 val2* (integers): number of columns (N_x) and rows (N_y);

- PixelSize *val1 val2*
  - pixel size ($L_x \times L_y$, cm);

  - *val1 val2* (real values): rectangle sides (L_x, L_y);

- Shape *val*
  - pixel shape:
  - *val* = 0: rectangular;
  - *val* = 1: elliptical;

- dOmegaLim *val*
  - cut on ε (if this entry is 0, then ε is set to the default value 2π);
  - *val* (real value) ε;

- X *val1 val2 val3*
  - *val1 val2 val3* (real values) detector geometric center coordinates x, y, z;

- uk *val1 val2 val3*
  - detector orientation: $\mathbf{k}_d$ components (local z axis direction, i.e. normal with respect to the detector surface);
  - *val1 val2 val3* (real values): k_{dx}, k_{dy}, k_{dz}

- ui *val1 val2 val3*
  - detector source orientation: $\mathbf{i}_d$ components (local x axis direction);
  - *val1 val2 val3* (real values): i_{dx}, i_{dy}, i_{dz}

- ExpTime *val*
  - *val* (real value): exposure time (seconds)

- PhotonNum *val*
  - *val* (integer): multiplicity of simulated events per pixel;
- RandomPixelFlag *val*
  - *val* (integer): enable random point on pixels (0/1)
- PoissonFlag *val*
  - *val* (integer): enable Poisson noise on pixel counts (0/1)
- RoundFlag *val*
  - *val* (integer): round pixel counts to integer (0/1)
- HeaderFlag *val*
  - *val* (integer): use header in output file (0/1)
- AsciiFlag *val*
  - *val* (integer): use binary or ascii output file format (0/1)
- Rotate *val1 val2 val3 val4 val5 val6 val7*
  - rotation of the detector around the axis passing through $\mathbf{x}_0$ and with direction $\mathbf{u}$:
  - *val1 val2 val3*(real values): x_0, y_0, z_0;
  - *val4 val5 val6*(real values): u_x, u_y, u_z;
  - *val7* (real) rotation angle θ (degrees);
- PixelType *val*
  - Pixel content type:

  - *val* (integer): 0, 1, 2 or 3;
  - 0: fluence;
  - 1: energy fluence;
  - 2: fluence with energy binning;
  - 3: energy fluence with energy binning;

Only if energy binning is used, i.e. if pixel content type is 2 or 3:

- Emin *val*
  - *val* (real): minimum binning energy;
- Emax *val*
  - *val* (real): maximum binning energy;
- NBins *val*
  - *val* (integer): number of energy bins;
- SaturateEmin *val*
  - *val* (integer): saturate energies lower than Emin (0/1)
- SaturateEmax *val*
  - *val* (integer): saturate energies greater than Emax (0/1)
- End
  End of file.

If $\mathbf{i}_d$ is not perpendicular to $\mathbf{k}_d$, then xrmc will replace it with a vector parallel to the same plane, but perpendicular to $\mathbf{k}_d$. If $\mathbf{k}_d$ or $\mathbf{i}_d$ are not unit vectors then they will be normalized by the program. The vector $\mathbf{j}_d$ is computed by the program, so that $\mathbf{i}_d$, $\mathbf{j}_d$ and $\mathbf{k}_d$ constitute an orthonormal basis set.

3.2.5 The composition file

The materials that compose the sample are called *phases*. A composition file is used to list all the phases and to characterize them by their mass density and by their composition, i.e. the chemical formulas and weight fractions of the compounds that compose them. Each phase is assumed to be homogeneous. Phases are referred to by their user-defined names. There is a predefined phase named *Vacuum* with mass density equal to zero and no elements. This is the phase that fill the *universe*. If the user wants to simulate an experiment in a different medium (e.g. in air) he should first define it. However, only a finite region of space can be filled by a phase different from vacuum. The chemical formulas are parsed with the CompoundParser function of *xraylib*: examples of accepted formulas are `H2O` (water) and `Ca5(PO4)3F` (fluor apatite).

composition file header

- Newdevice composition
- *name* (string)

Commands

- Phase *name*
  - *name* (string): material name; define a new material;
- NCompounds *val*
  - *val* (integer): number of compounds N_e in the phase;
- Compound$_1$ w$_1$ (string, real): chemical formula and weight percent of the 1st compound;
- …
- Compound$_{Nc}$ w$_{Nc}$ (string, real): chemical formula and weight percent of the N-th compound;
- Rho *val*
  - *val* (real): mass density of the phase (g/cm³);

- End
  - End of file.

3.2.6 The quadric array file

The sample geometry is described through a set of quadrics, which are used to define the surfaces of solid objects. A quadric is a surface in the three-dimensional space defined as the locus of zeros of a quadratic polynomial. The general quadric is defined by the algebraic equation:

$$\sum_{i,j=1}^{3} x_i Q_{ij} x_j + \sum_{i=1}^{3} P_i x_i + R = 0 \tag{2}$$

If we define $x_4 = 1$, then the general quadric may be compactly written in vector and matrix notation as:

$$x A x^T = 0 \tag{3}$$

where $x = (x_1, x_2, x_3, x_4)$, x^T the transpose of x (a column vector) and A is a 4 x 4 matrix with A_{ij} for $i, j = 1,\ldots,3$, $A_{4i} = A_{i4} = P_i$ and $A_{44} = R$. The matrix A must be symmetric, thus $A_{ij} = A_{ji}$, □ $i, j = 1,\ldots,4$.
A quadric divides the space in two regions, one with $xAx^T > 0$, the other with $xAx^T < 0$.
We will call those two regions space *outside* the quadric (or *external* space) and space *inside* the quadric (or *internal* space), respectively, no matter whether the quadric is closed or not. Whenever a unit vector normal to the quadric surface has to be defined, by default we will assume that is is oriented toward the external space. Examples of quadrics are planes, ellipsoids, cylinders.

quadricarray file header

- Newdevice quadricarray
- *name* (string)

Commands

- Quadric A_{11} A_{12} A_{13} A_{14} A_{22} A_{23} A_{24} A_{33} A_{34} A_{44}
  - Generic quadric defined by its elements contents. Since the matrix has to be symmetric, only 10 elements are used.
- Plane x_0 y_0 z_0 u_x u_y u_z

  - Plane containing the point (x_0 , y_0 , z_0) with normal vector (u_x , u_y , u_z).

- Ellipsoid x_0 y_0 z_0 R_x R_y R_z
  - Ellipsoid with principal axis parallel to the main axis, centered in (x_0 , y_0 , z_0) with semi-axes R_x , R_y , R_z .

- CylinderX y z R_y R_z
  - Cylinder having the main axis parallel to the x axis with coordinates y, z on the yz plane, and having elliptical section with semi-axes R_y , R_z .

- CylinderY x z R_x R_z
  - Cylinder having the main axis parallel to the y axis with coordinates x, z on the xz plane, and having elliptical section with semi-axes R_x , R_z .

- CylinderZ x y R_x R_y
  - Cylinder having the main axis parallel to the z axis with coordinates x, y on the xy plane, and having elliptical section with semi-axes R_x , R_y .

- Translate Δx Δy Δz
  - Translate the position of the last defined quadric by (Δx Δy Δz).

- Rotate x_0 y_0 z_0 u_x u_y u_z θ
  - Rotate the last defined quadric around the axis passing through the point (x_0 , y_0 , z_0) and directed as (u_x , u_y , u_z) by an angle θ (expressed in degrees).

- TranslateAll Δx Δy Δz
  - Translate the position of all previously defined quadrics by (Δx Δy Δz).

- RotateAll x_0 y_0 z_0 u_x u_y u_z θ

  - Rotate all previously defined quadrics around the axis passing through the point (x_0 , y_0 , z_0) and directed as (u_x , u_y , u_z) by an angle θ (expressed in degrees).

- End
  - End of file.

3.2.7 The three-dimensional object geometry description file

A solid *object* is defined as a solid shape delimited by a set of quadric surfaces that separates the space inside from the space outside it. The quadrics delimiting an object must be properly oriented, in such a way that their normal vectors are directed outward with respect to the object itself.

- **Only convex objects are allowed in XRMC**. Non-convex shapes can be built by combining convex objects. The demo files show some examples of how to build non- convex shapes.

- **The surfaces delimiting different objects can not be in contact with each other.** The users have to pay particular attention to this point, because the code does not make any check on it. If the user wants two objects to be in contact with each other (for instance when building non-convex shapes by joining convex objects) a workaround is to separate them using two different quadrics, very close to each other but separated by a small gap, in such a way that the effect of the intermediate space on the radiation is negligible. The demo files show some examples of this trick.

- An object may contain other objects, or it may be contained in another object, as far as their delimiting surfaces are not in contact.

All objects must be contained in a finite region of space, called sample region.
The user is required to assign the phases (see the composition file) that are found inside and outside the object. For this purpose one can use the phases defined in the composition file, or from the NIST materials database. In the latter case, make sure that all spaces are preceded by a backslash. This functionality is offered through xraylibs `GetCompoundDataNISTByName` function.

geom3d device file header

- Newdevice geom3d

- *name* (string)

Commands

- QArrName *name*
  - *name* (string): quadricarray input device name;
- CompName *name*
  - *name* (string): composition input device name;
- Object *name*
  - *name* (string): 3d object name; defines a new object;
- *phase-in-name*
  - name of the phase (material) inside the object;
- *phase-out-name*
  - name of the phase (material) surrounding the object;
- N_q
  - number of quadrics that define the object surface;
- *quadric-name-1 quadric-name-2 ... quadric-name-*N_q
  - names of the quadrics that define the object surface;
- End
  - End of file.

3.2.8 The sample file

When a photon exits from the source or when it is produced by a scattering or fluorescence emission process, its trajectory is characterized by its position vector $\mathbf{r}_{ph}$ and by its direction vector $\mathbf{u}_{ph}$. The program evaluates the intersection of this trajectory with the quadric surfaces delimiting the objects, and divide it in N_s steps with uniform phases. Each step is a segment of the trajectory delimited by its intersection with different objects. There are two possible modalities of extracting the next interaction position:

1. extract the interaction position according to the interaction probability distribution along the trajectory, which is evaluated from the linear absorption coefficient of the phases and from the steplengths of the paths;

2. extract a step at random using a random integer number $0 \leq m < N_s$, Extract a random position on step m using a uniform probability distribution, and multiply the weight associated to the event by a proper factor;

The first option reflects the traditional Monte Carlo approach. The second is sometimes used for variance reduction. It may be useful, for instance, when an object that emits a relevant fluorescence signal, is surrounded by a strongly absorbing material, in such a way that the probability of the incident radiation reaching such an object is relatively low.

sample device file header

- Newdevice sample

- *name* (string)

Commands

- SourceName *name*

  - *name* (string): source input device name;

- Geom3DName *name*

  - *name* (string): geom3d input device name;

- CompName *name*

  - *name* (string): composition input device name;

- WeightedStepLength *val*
  - setting that determines the algorithm used for the extraction of the next interaction position:
  - *val* = 0 : method 1 described above;
  - *val* = 1: method 2) described above;
- FluorFlag *val*
  - activate Fluorescence (0/1); it can be useful to deactivate it in imaging experiments where fluorescent emission is not relevant, and thereby to save computational time;
- ScattOrderNum N_I
  - N_I (integer): maximum scattering order (0: transmission, 1: first order scattering or fluorescence emission, 2: second order scattering or fluorescence emission, …)
- M_1
  - multiplicity of simulated events for order 0;
- …
- M_{NI}
  - multiplicity of simulated events for order N_I;
- End
  - End of file.

As discussed previously, the total number of generated events is given by the product of the multiplicities in the spectrum, in the interactions with the sample and in the detector pixels.

3.2.9 The output file

By default, the output file is saved in raw binary format. The detector bin contents are written in C *double* format (64 bit real). If you prefer to save the data in ascii format, you can use the *AsciiFlag* command, which is described in the *detectorarray* file section.
The total number of entries is:

> N. of scattering orders × N. of energy bins × N. of columns × N. of rows

with the rows *running faster*.
If the *HeaderFlag* is set to 1 in the detector file, then the bin contents are preceded by a 60- bytes-long header, also in binary format, containing the following information:

- N. of scattering orders (C int format, 32 bit integer)
- N. of columns (C int format, 32 bit integer)
- N. of rows (C int format, 32 bit integer)
- Pixel size Sx (C double format, 64 bit real)
- Pixel size Sy (C double format, 64 bit real)
- Exposure time in sec. (C double format, 64 bit real)
- Pixel content type (C int format, 32 bit integer)
- N. of energy bins (C int format, 32 bit integer)
- Minimum bin energy (C double format, 64 bit real)
- Maximum bin energy (C double format, 64 bit real)

4 Advanced usage

This section covers some advanced topics that may be of interest to expert users. Some of the devices that will be discussed here inherit (remember: all devices are C++ objects) properties from other devices that have been explained in the User guide.

4.1 Direction-dependent beam intensity and energy spectrum

These paragraphs describes special devices used to define a beam with direction-dependent intensity and/or spectrum.

4.1.1 The anisotropicsource and the intensityscreen devices

The *anisotropicsource* and the *intensityscreen* devices are used to describe a source with intensity that depends on the emission direction. Note that for these devices the energy spectrum does not depend on the direction: if the user desires that not only the intensity, but also the spectrum depends on the direction, he should rather use the *beamsource-beamscreen* devices, described in the following section.
The anisotropicsource input file has exactly the same format as the source file, the only differences being the device type, specified after the Newdevice command, and an additional input device, of the intensityscreen type.

anisotropicsource device file header

- Newdevice anisotropicsource
- *name* (string)

Additional command

- IntensityScreenName *name*
  - *name* (string): intensityscreen input device name;

The intensityscreen device represents the beam intensity distribution on an ideal screen, located in an arbitrary position and with arbitrary orientation. The ideal screen is divided in pixels. The intensity distribution is defined by specifying the number of x-ray photons per seconds impinging on each pixel.

intensityscreen device file header

- Newdevice intensityscreen
- *name* (string)

Commands

- NPixels *val1 val2*
  - pixel number ($N_x \times N_y$);

  - *val1 val2* (integer values): number of columns (N_x) and rows (N_y);

- PixelSize *val1 val2*
  - pixel size ($L_x \times L_y$, cm);
  - *val1 val2* (real values): rectangle sides (L_x , L_y);

- X *val1 val2 val3*
  - *val1 val2 val3* (real values) screen geometric center coordinates x, y, z;

- uk *val1 val2 val3*
  - orientation: $\mathbf{k}_d$ components (local z axis direction, i.e. normal to the screen surface);
  - *val1 val2 val3* (real values): k_{dx}, k_{dy}, k_{dz}

- ui *val1 val2 val3*
  - orientation: $\mathbf{i}_d$ components (local x axis direction);
  - *val1 val2 val3* (real values): i_{dx}, i_{dy}, i_{dz}

- InterpolFlag *val*
  - *val* (integer): use interpolation inside pixels (0/1)

- ImageFile *filename*
  - *filename* (string) name of the file containing the intensity distribution on the screen pixels. The file should contain $N_x \times N_y$ entries (one for each pixel) stored in C double format (64 bit real) with the x-running-faster ordering scheme.

- Rotate *val1 val2 val3 val4 val5 val6 val7*

  - rotation of the screen around the axis passing through $\mathbf{x}_0$ and with direction **u**:

  - *val1 val2 val3*(real values): x_0, y_0, z_0;

  - *val4 val5 val6*(real values): u_x, u_y, u_z;

  - *val7* (real) rotation angle θ (degrees);

4.1.2 The beamsource and the beamscreen devices

The *beamsource* and the *beamscreen* devices are used to describe a source with both the intensity and the energy spectrum that depends on the emission direction. The beamsource input file has exactly the same format as the source file, the only differences being the device type, specified after the Newdevice command, and the input device, which is of type beamscreen instead of spectrum.

beamsource device file header

- Newdevice beamsource

- *name* (string)

Input device command The following command replaces the SpectrumName command used in the source device.

- BeamScreenName *name*

  - *name* (string): beamscreen input device name;

The beamscreen device represents the direction-dependent beam intensity and energy spectrum on an ideal screen, located in an arbitrary position and with arbitrary orientation. The ideal screen is divided in pixels and energy bins. The intensity distribution and the spectrum are defined by specifying, for each pixel and energy bin, the number of x-ray photons per seconds impinging on that pixel with energy inside that bin.

beamscreen device file header

- Newdevice beamscreen

- *name* (string)

Commands

- NPixels *val1 val2*
  - pixel number ($N_x \times N_y$);
  - *val1 val2* (integer values): number of columns (N_x) and rows (N_y);
- PixelSize *val1 val2*
  - pixel size ($L_x \times L_y$, cm);
  - *val1 val2* (real values): rectangle sides (L_x , L_y);
- X *val1 val2 val3*
  - *val1 val2 val3* (real values) screen geometric center coordinates x, y, z;
- uk *val1 val2 val3*
  - orientation: $\mathbf{k}_d$ components (local z axis direction, i.e. normal to the screen surface);
  - *val1 val2 val3* (real values): k_{dx}, k_{dy}, k_{dz}
- ui *val1 val2 val3*
  - orientation: $\mathbf{i}_d$ components (local x axis direction);
  - *val1 val2 val3* (real values): i_{dx}, i_{dy}, i_{dz}
- InterpolFlag *val*
  - *val* (integer): use interpolation inside pixels (0/1)
- PolarizedFlag *val*
  - *val* (integer): unpolarized/polarized beam flag (0/1)

- LoopFlag *val*
  - energy extraction method:
  - *val* = 0: extract random energies on the whole spectrum;
  - *val* = 1: loop over all energy bins;
- PhotonNum *val*
  - *val* (integer): multiplicity of simulated events for each energy bin;
- Emin *val*
  - *val* (real): minimum binning energy;
- Emax *val*
  - *val* (real): maximum binning energy;
- NBins *val*
  - *val* (integer): number of energy bins;
- ImageFile *filename*
  - filename (string): name of the file containing the intensity/energy spectrum distribution on the screen pixels/bins. For unpolarized beam, the file should contain $N_{bins} \times N_y \times N_x$ entries (one for each pixel) stored in C double format (64 bit real) with the x-running- faster ordering scheme. For polarized beam, the file should contain $2 \times N_{bins} \times N_y \times N_x$ entries, with the x-polarized component in the first half and the y-polarized component in the second half of the file.
- Rotate *val1 val2 val3 val4 val5 val6 val7*
  - rotation of the screen around the axis passing through $\mathbf{x}_0$ and with direction $\mathbf{u}$:

- *val1 val2 val3*(real values): x_0, y_0, z_0;
- *val4 val5 val6*(real values): u_x, u_y, u_z;
- *val7* (real) rotation angle θ (degrees);

4.2 Simulation of a realistic imaging detector response

A realistic imaging detector response can be simulated by issuing additional commands in the detectorarray input file. Such commands are used to define the detector *point spread function* (PSF), the source size and the detector efficiency versus energy.
The x/y projections of the detector PSF and of the source size are modeled by the superposition of one or more Gaussian functions. Those function can be fixed (not dependent on energy) or energy dependent. In the latter case, the parameters of all Gaussians (height, mean, sigma) for each energy bin are loaded from a separate file.

Additional commands

- ConvolveFlag *val*
  - *val* (integer): Generates convoluted image (0/1);
- Z12 *val*
  - *val* (real value): Distance between object plane and detector, used to take into account source size in detector image convolution;
- EfficiencyFlag *val*
  - *val* (integer): flag for using efficiency versus energy before convolution (0/1); to use it, ConvolveFlag MUST also be activated.
- EfficiencyFile *filename*
  - *filename*: name of the input file holding the efficiency;
  - file format (ascii):
  - ε_1: efficiency for the first energy bin

– …

– ε_{Nbins}: efficiency for the last energy bin

- GaussPSFx *N*

  – Gaussian model of detector PSF (x component): superposition of *N* Gaussian functions (*N* = 0 : disabled);

  – h_1 x_1 σ_1: height, mean and standard deviation of the first Gaussian function;

  – …

  – h_N x_N σ_N: height, mean and standard deviation of the *N*-th Gaussian function;

- GaussPSFy *N*

  – Gaussian model of detector PSF (y component): superposition of *N* Gaussian functions (*N* = 0 : disabled);

  – h_1 x_1 σ_1: height, mean and standard deviation of the first Gaussian function;

  – …

  – h_N x_N σ_N: height, mean and standard deviation of the *N*-th Gaussian function;

- GaussPSFxBinFile *N filename*

  – Energy dependent Gaussian model of detector PSF (x component): superposition of *N* Gaussian functions (*N* = 0 : disabled);

  – *filename* (string): name of the input file holding Gaussian function parameters; input file format (ascii): 3 *N* columns with height, mean and sigma of the *N* Gaussian functions for each energy bin:

  – h_1 x_1 σ_1 h_2 x_2 σ_2 … h_N x_N σ_N: first energy bin

– …

– h_1 x_1 σ_1 h_2 x_2 σ_2 … h_N x_N σ_N: last energy bin

- GaussPSFyBinFile *N filename*

  – Energy dependent Gaussian model of detector PSF (y component): superposition of *N* Gaussian functions (*N* = 0 : disabled);

  – *filename* (string): name of the input file holding Gaussian function parameters; input file format (ascii): 3 *N* columns with height, mean and sigma of the *N* Gaussian functions for each energy bin:

  – h_1 x_1 σ_1 h_2 x_2 σ_2 … h_N x_N σ_N: first energy bin

  – …

  – h_1 x_1 σ_1 h_2 x_2 σ_2 … h_N x_N σ_N: last energy bin

- GaussSourceX *N*

  – Gaussian model or source x size: superposition of *N* Gaussian functions (*N* = 0 : disabled);

  – h_1 x_1 σ_1: height, mean and standard deviation of the first Gaussian function;

  – …

  – h_N x_N σ_N: height, mean and standard deviation of the *N*-th Gaussian function;

- GaussSourceY *N*

  – Gaussian model or source y size: superposition of *N* Gaussian functions (*N* = 0 : disabled);

  – h_1 x_1 σ_1: height, mean and standard deviation of the first Gaussian function;

  - …

  - h_N x_N σ_N: height, mean and standard deviation of the N-th Gaussian function;

- GaussSourceXBinFile *N filename*

  - Energy dependent Gaussian model of source x size: superposition of N Gaussian functions ($N = 0$: disabled);

  - *filename* (string): name of the input file holding Gaussian function parameters; input file format (ascii): 3 N columns with height, mean and sigma of the N Gaussian functions for each energy bin:

  - h_1 x_1 σ_1 h_2 x_2 σ_2 … h_N x_N σ_N: first energy bin

  - …

  - h_1 x_1 σ_1 h_2 x_2 σ_2 … h_N x_N σ_N: last energy bin

- GaussSourceYBinFile *N filename*

  - Energy dependent Gaussian model of source y size: superposition of N Gaussian functions ($N = 0$: disabled);

  - *filename* (string): name of the input file holding Gaussian function parameters; input file format (ascii): 3 N columns with height, mean and sigma of the N Gaussian functions for each energy bin:

  - h_1 x_1 σ_1 h_2 x_2 σ_2 … h_N x_N σ_N: first energy bin

  - …

  - h_1 x_1 σ_1 h_2 x_2 σ_2 … h_N x_N σ_N: last energy bin

4.3 Free-space propagation phase contrast imaging

The x-ray phase contrast imaging techniques are based on the observation of interference patterns produced when an x-ray beam partially or totally coherent crosses an object characterized by variation in the real part of the refractive index with position. In a typical setup for this type of experiment, the radiation produced by a relatively small x-ray source acquires partial coherence during propagation in free

space, crosses an object placed at relatively large distance from the source, and produces an image on an observation plane (screen).
The method used by XRMC for phase contrast imaging simulation is described in ref. [4]. Such method can be used for monochromatic as well as for polychromatic sources. Furthermore, while other methods are valid only in the approximation of relatively large source-object distance, the method used by XRMC is suitable also in the case of small source-object distance. On the other hand, the thin-sample approximation is used: assuming that the object is thin along the radiation propagation direction, the deviation of the x-ray paths inside the object from straight lines can be neglected.
Phase contrast imaging experiments can be simulated in XRMC by using the special device *phcdetector*, which inherits all the variable names and commands of the detectorarray device, but has additional commands specific for phase contrast imaging. All the devices that can be used for imaging experiments can also be used for phase contrast imaging, except for the detectorarray device, which is replaced by the phcdetector device.
Note: the detector pixel size Lx × Ly must satisfy the inequality:

$$L_x \times L_y \ll Z_{12} \times \lambda \tag{4}$$

where Z_{12} is the object detector distance and λ is the radiation wavelength;

phcdetector device file header

- Newdevice phcdetector
- *name* (string)

Additional commands

- Z_{12} *val*
  - *val* (real value): Distance between object plane and detector;
- NScreenBorder *val1 val2*
  - additional rows and columns at the top, bottom, left and right sides of the detector ($NB_x \times NB_y$);
  - *val1 val2* (integer values): number of additional columns (NB_x) at the left and right sides, and additional rows (NB_y) at the top and bottom of the detector; the simulation is extended to those additional pixels in order to avoid discontinuities at the border in the simulated phase contrast image. If you are not sure of which values you should use, simply

do not use this command, so that the default values will be used.

- NInterpBorder *val1 val2*
  - additional rows and columns at the top, bottom, left and right sides of the detector, used for interpolation ($\text{NIB}_x \times \text{NIB}_y$);
  - val1, val2 (integer values): number of additional columns (NIB_x) at the left and right sides, and additional rows (NIB_y) at the top and bottom of the detector; those additional pixels are not included in the simulation, but they are used to interpolate the complex values of the transmission function from the border of the simulated image to the flat field values, to avoid discontinuities at the border in the simulated phase contrast image. If you are not sure of which values you should use, simply do not use this command, so that the default values will be used.

4.4 The XMI-MSIM plug-in

A plug-in was developed that allows xrmc to exploit some of the capabilities of the XMI-MSIM package. This requires however, that when compiling XRMC, a full installation of XMI-MSIM is already present on the system. In order to obtain XMI-MSIM, the user is referred to XMI-MSIMs Download and installation instructions. For users interested in writing plug-ins for XRMC, it is highly recommended to have a look at the source code of the XRMC-XMI-MSIM plug-in for inspiration.

4.4.1 Energy dispersive x-ray fluorescence detector response function

XMI-MSIM contains a number of routines that allow one to generate a detector response function for energy-dispersive x-ray fluorescence (ED-XRF) detectors. This is includes support for the Gaussian detector broadening, escape peaks (fluorescence and Compton) as well as peak pile-up (sum peaks).
For a full description, the reader is referred to Schoonjans et al (2012).
Invoking this detector response function is done in XRMC through defining a *detectorconvolute* device, which inherits from detectorarray.

detectorconvolute device file header

- Newdevice detectorconvolute
- *name* (string)

Additional commands

- CrystalPhase *name*
  - *name* (string): name of the phase (material) of the detector crystal;
- WindowPhase *name*
  - *name* (string): name of the phase (material) of the detector window;
- CrystalThickness *val*
  - *val* (real value): thickness (cm) of the phase (material) of the detector crystal;
- WindowThickness *val*
  - *val* (real value): thickness (cm) of the phase (material) of the detector window;
- PulseWidth *val*
  - *val* (real value): the time (s) necessary for the detector electronics to process one incoming pulse.
  - This parameter is optional: leave it out if the simulation of the pulse pile-up is not required;
- FanoFactor *val*
  - *val* (real value): measure of the dispersion of a probability dis- tribution of the fluctuation of an electric charge in the detector. Very much detector type dependent;
- Noise *val*
  - *val* (real value): the result of random fluctuations in thermally generated leakage currents within the detector itself and in the early stages of the amplifier components. Contributes to the Gaussian broadening (keV);

4.4.2 Generic x-ray tube emission spectrum: Ebel model

The *spectrum_ebel* device offered by the XMI-MSIM plug-in allows the user to generate x-ray tube emission spectra based on a number of parameters. The model that is implemented in XMI-MSIM is based on the work of Prof. Horst Ebel. More information can be found in his 1999 and 2003 papers.
This device inherits from the spectrum device.

spectrum_ebel device file header

- Newdevice spectrum_ebel
- *name* (string)

Additional commands

- TransmissionFlag (0/1)
  - Setting this parameter to 1 will assume that the tube is of the transmission type, i.e. with cathode and window on opposing sides of the anode.
- TubeCurrent *val*
  - *val* (real value): the current of the x-ray tube (mA);
- TubeVoltage *val*
  - *val* (real value): the operating voltage of the x-ray tube (V);
- ElectronAngle *val*
  - *val* (real value): angle of electron incidence with respect to the anode target surface (degrees);
- XrayAngle *val*
  - *val* (real value): angle of x-ray take-off with respect to the anode target surface (degrees);
- IntervalWidth *val*

  - *val* (real value): width in keV of the intervals that make up the continuous part of the generated spectrum;

- AnodeMaterial *element*
  - *element* (string): the material that the anode is made of;

- AnodeDensity *val*
  - *val* (real value): the density of the anode material (g/cm³);

- AnodeThickness *val*
  - *val* (real value): the thickness of the anode material (cm);
  - This parameter is only used when operating in transmission mode

- WindowMaterial *element*
  - *element* (string): the material that the tube window is made of;

- WindowDensity *val*
  - *val* (real value): the density of the tube window material (g/cm³);

- WindowThickness *val*
  - *val* (real value): the thickness of the tube window material (cm);

- FilterMaterial *element*
  - *element* (string): the material that the tube filter is made of;

- FilterDensity *val*
  - *val* (real value): the density of the tube filter material (g/cm³);

- FilterThickness *val*
  - *val* (real value): the thickness of the tube filter material (cm);

5 Description of the XRMC classes

In this section we will cover all of the C++ classes that are defined in XRMC, and that can be extended by the users in order to create custom devices.

5.1 Vector and matrix operations

Many of the calculations made by the program involve operations on three-dimensional vectors and on 3×3 or 4×4 real matrices. For such reason, three specialized classes, *vect3*, *matr3* and *matr4* have been implemented. Those classes exploit the operator overloading feature of the C++ programming language in order to represent vector and matrix operations in a natural and readable manner. For instance, a change of coordinates from the local coordinate system associated to a device to the absolute coordinate system of XRMC can be expressed by the vector operation:

$$\mathbf{r} = \mathbf{r_0} + \mathbf{u_i} \cdot x + \mathbf{u_j} \cdot y + \mathbf{u_k} \cdot z \tag{5}$$

with **r** the the position vector of a point relative to the absolute coordinate system, $\mathbf{r}_0$ the position vector of the local coordinate system origin, $\mathbf{u}_i$, $\mathbf{u}_j$ and $\mathbf{u}_k$ the direction vectors of the local coordinate system axis and *x*, *y* and *z* the coordinates of the point relative to the local coordinate system. This operation can be expressed using the *vect3* class as:

$$r = r_0 + u_i \cdot x + u_j \cdot y + u_k \cdot z \tag{6}$$

with r, u_i, u_j and u_k objects of the class *vect3*, x, y and z are of the C/C++ type *double*, and ``+" and ``*" are overloaded operators of the *vect3* class.
An alternative approach was based on more general matrix and vector base classes, and derived classes for specific dimensions. However, the computation time overhead of this approach was relevant, therefore the final choice was to use specialized classes for three and four dimensions.
The *vect3* class member variables are the three vector components, while its main member functions are:

- product/division of a vector by a scalar;
- sum and difference between vectors;
- scalar product of two vectors;
- vector product;
- modulus;
- vector normalization;

The first three are represented by the C/C++ algebraic operators, while the vector product is represented by the symbol ``^".
The matr3/matr4 member variables are the 9/16 matrix elements, while their main member functions are:

- matrix multiplication;
- matrix transposition operator;
- multiplication of a matrix by a vector;
- rotation matrix of an angle θ around an axis with direction **u**;

5.2 The photon class

XRMC uses a specialized photon class to describe x-ray photon transport and interaction with matter. A local coordinate system is associated with the photon, with the direction of the three axes x, y, and z defined by three unit vectors $\mathbf{i}_{ph}$, $\mathbf{j}_{ph}$, $\mathbf{k}_{ph}$, with $\mathbf{k}_{ph}$ the photon direction, $\mathbf{i}_{ph}$ the polarization vector, which is always perpendicular to the photon direction, and $\mathbf{j}_{ph}$, the unit vector perpendicular to both $\mathbf{i}_{ph}$ and $\mathbf{k}_{ph}$. The other variables that describe the photon state are its energy E and its weight w.
The main member variables of the photon class are:

- double w: event weight;
- double E: photon energy;
- vect3 x: photon position vector;
- vect3 ui, uj, uk: direction vectors of the photon local coordinate system;

The main member functions of the photon class are:

- int MoveForward(double step_length): moves the photon in the direction uk by a distance step_length;
- int ChangeDirection(double theta, double phi): updates the photon direction using the polar angle theta and the azimuthal angle phi in the local coordinate system;
- int MonteCarloStep(sample *Sample, int* iZ, int *iType): evaluates the photon next interaction atomic number, type and position;

- int CSInteractions(int Z, double *mu_interaction, double* cs_tot): evaluates the cross sections of the three interaction types with the extracted element;
- int InteractionType(double *cs_interaction, double cs_tot): extracts the interaction type (elastic/inelastic scattering or fluorescence);
- int SetFluorescenceEnergy(int Z): set the photon energy to that of the fluorescent emission line;
- int Scatter(int Z, int interaction_type); checks the interaction type and launches the corresponding method;
- int Scatter(int Z, int interaction_type, vect3 v_r): analogous to the previous function, but with the photon forced to have the direction v_r;
- int Fluorescence(): generates a fluorescent emission process;
- int Coherent(int Z): generates an elastic scattering process;
- int Incoherent(int Z): generates an inelastic scattering process.

5.3 The device and bodydevice classes

In XRMC the term device refers to a C++ object that is created in order to be used by the simulation, and whose parameters are loaded from the corresponding file. It does not necessarily correspond to a physical device. For instance, the phase (material) array and the sample are considered to be devices. The term device is also the name of the abstract base class from which all concrete device classes are derived. The member variables of the device class are:

- the device name;
- the device type;
- an index for the loop on the events;

The main virtual methods of the class are:

- the *ImportDevice* method, used to connect a device to one or more input devices through a device map;
- the *Load* method, used to load the device parameters from an input file;

- the *SetDefault* method, which sets default values for device parameters;
- the *Begin*, *Next* and *End* methods, used to control the loop on the events;
- the *Run* method;
- the *Save* method, which saves the device output to a file;

The last two methods are not used by all devices. Only the detectorarray device has implemented them, in the current version of the package.
Another important abstract class used in XRMC, derived from the device class, is the *bodydevice* class. Besides the device member variables and functions, a body-device is characterized by a position and a local coordinate system, through the following member variables:

- vect3 X: bodydevice position;
- vect3 ui, uj, uk: local coordinate system axis directions.

The classes derived from the bodydevice class are the *basesource* and *detectorarray* classes. The basesource class is an abstract class for x-ray sources. It is the base class for all classes that can send x-ray photons to other devices, which in the current version of the code are the *source* class (which can send photons to the sample or to the detector devices) and the *sample* class (which can send photons to another sample or to a detector devices). In future releases of the program, other classes derived from the basesource class could be used to represent optical elements, such as x-ray mirrors, multilayers and x-ray lenses.
The methods of the basesource class are:

- the *ModeNum* method, used to specify the number of modes the device can work as. With regard to the *sample* derived class, it refers to the number of scattering orders used in the simulation;
- the *Out_Photon* method, used to send a photon to the output device;
- the Out_Photon_x1_ method, used to generate an event with a photon forced to be directed toward the position x1, specified by the output device;

On the other hand, the concrete classes that are directly derived from the abstract device class are the *composition*, *spectrum*, and *geom3d* classes.
Each device can be connected to one or more input devices. The following figure shows schematically how the concrete devices used in a standard setup are inter-connected. The classes of those devices are described in the following sections.

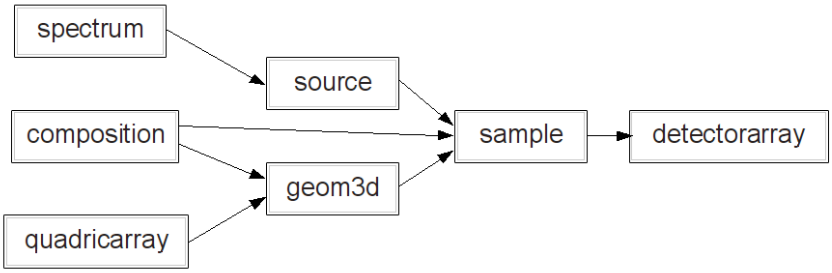

Figure 1: C++ classes associated to the main XRMC devices and connections used in a standard setup.

5.4 The source class

The current version of the code assumes that the x-ray beam is produced by a point source or by an extended source with a three-dimensional Gaussian distribution. A local coordinate system is attached to the source, which is determined by the vector position of the origin $\mathbf{r}_s$ and by the orthonormal unit vectors $\mathbf{i}_s$, $\mathbf{j}_s$ and $\mathbf{k}_s$, which correspond to the directions of the local *x* , *y* , and *z* axis, respectively. The local *z* axis represents the main source direction, while the local *x* and *y* axes are used to define the beam polarization and angular aperture. Let θ_s and φ_s be the polar and azimuthal angle, respectively, relative to the source coordinate system. The user can specify the angular apertures θ_x and θ_y in the *x* and in the *y* direction, respectively. In general, the angular aperture of the source is elliptical and it is defined by the inequality:

$$\theta^2 \leq \theta_x^2 \cos^2 \phi_s + \theta_y^2 \sin^2 \phi_s \tag{7}$$

The source intensity distribution is assumed to be uniform within the solid angle delimited by this angular aperture.
The main member variables specific of the source class (not belonging to its base classes) are the following:

- spectrum *Spectrum: pointer to the input spectrum device;
- string SpectrumName: name of input spectrum device;
- double Thx, Thy: beam divergence (θ_x and θ_y);
- double Omega: source aperture solid angle;
- double Sigmax, Sigmay, Sigmaz: source size in the local coordinate system.

The main member functions specific of the source class are:

- double CosThL(double phi); maximum value of the polar angle θ for a specified value of φ;
- int PhotonDirection(photon *Photon, int pol): extract the initial direction and polarization of a photon;
- int SetPhotonAxes(photon *Photon, int pol): build the photon local axis based on its direction and polarization;
- double POmega(vect3 vr): probability per unit solid angle that a photon has direction vr .

5.5 The spectrum class

The energy spectrum is modeled as the sum of two components: a set of discrete lines and a continuous component. The radiation can be unpolarized, partially polarized or completely polarized. The lines can have a Gaussian or a Dirac δ distribution (the latter one being simply a particular case of Gaussian distribution with $\sigma = 0$). Each line is specified by its mean energy, its intensity (two components if polarized), and its standard deviation.
The continuous component is defined by N samples at different energies and the corresponding height of the spectral distribution.
The main member variables of the spectrum class are the following:

- int PolarizedFlag: flag for polarized(1) / unpolarized(0) beam;
- int LoopFlag: flag for loop on all lines and all intervals of the spectrum;
- int RandomEneFlag: flag for extracting random energies in the intervals;
- int ResampleFlag: flag for resampling the continuous spectrum;
- int EneContinuousNum: number of sampling points in the continuous spectrum;
- double *ContinuousEne: energies of sampling points;
- double *ContSIntensity[2]: intensity at sampling points for the two polarization components;
- double ContinuousEnergyRange: energy range of the continuous spectrum;

- double ContinuousIntensity: total continuous intensity;
- double MaxIntensity: maximum continuous intensity;
- double *IntervalIntensity[2]: intensities of the intervals for the two polarization components;
- double *IntervalWeight[2]: weights of the intervals for the two polarization components;
- double *IntervalCumul: cumulative distribution of the intervals;
- int EneLineNum: number of discrete lines of the spectrum;
- double *LineEne: energies of the discrete lines;
- double *LineSigma: widths (sigma) of the discrete lines;
- double *LineIntensity[2]: intensities of the discrete lines for the two polarization components;
- double DiscreteIntensity: total intensity of the discrete spectrum;
- double *LineWeight[2]: weights of the discrete lines for the two polarization components;
- double *LineCumul: cumulative distribution of the lines;
- int ResampleNum: number of resampling points of the continuous spectrum;
- double Emin, Emax: minimum and maximum resampling energy;
- int ContinuousPhotonNum: number of events to be extracted for each interval;
- int ContinuousPhotonIdx: index of the event extracted for the interval;
- int LinePhotonNum: number of the events to be extracted for each discrete line;
- int LinePhotonIdx: index of the event extracted for the line;

- int PolIdx: index of polarization type (0: x, 1: y);
- int ModeIdx: mode index: continuous spectrum (0) or discrete lines(1);
- int IntervalIdx: index of the interval of the continuous spectrum;
- int LineIdx: index of the discrete line;
- double TotalIntensity: total intensity;

The main member functions specific for the spectrum class are the following:

- int ExtractEnergy(double *weight, double* Energy, int *polarization): generates a random energy value and polarization type;
- int ContinuousRandomEnergy(double *Energy, int* polarization): generates a random energy value from the continuous spectrum;
- int ExtractSpectrum(double trial_energy, double *x_intensity, double* y_intensity): returns the intensity of the continuous spectrum at the given energy value;
- int DiscreteRandomEnergy(double *Energy, int* polarization): generates a random energy value from the discrete part of the spectrum;
- int IntervalRandomEnergy(double *E, int interval_idx, int pol_idx): extract the energy value and polarization type from a specified interval of the continuous spectrum distribution;
- int Resample(): method for resampling the continuous spectrum.

5.6 The phase class

The sample is composed of a number of materials called *phases*. Each phase is assumed to be homogeneous. Each phase is characterized by the number of atomic species that define it, through a list of the atomic numbers and weight fractions of these species, and by its mass density;
The main member variables of the phase class are:

- int Nelem: number of elements in the phase;
- double Rho: mass density of the phase in g/cm³;
- int *Z: atomic number array;

- double *W: weight fraction array;
- double *MuAtom: atomic interactions total cross section array;
- double LastMu: linear absorption coefficient calculated at the current energy.

The main member functions are:

- int Mu(double E): evaluates the absorption coefficient at energy E;
- int AtomType(int *Z, double* mu_atom): extract the atomic species with which the interaction will occur;

5.7 The composition class

The *composition* class contains an array of the phases used by the simulation.
Its main member variables are:

- int Nphases: number of phases;
- int MaxNPhases: maximum number of phases;
- phase *Ph: phase array;
- phase_map PhaseMap: map of phases with their names.

The main member function specific of this class is:

- int Mu(double E): evaluates the absorption coefficient of each phase.

5.8 The quadric class

The sample geometry is described through a set of quadric surfaces, which are used to define the surfaces of solid objects. A quadric is a surface in the three-dimensional space defined as the locus of zeros of a quadratic polynomial.
The main member functions of the quadric class are:

- matr4 Matr: 4×4 real matrix;
- int Ninters: number of intersection of a trajectory with the quadric;
- double tInters[2]: parametric coordinates of the intersections;
- int Enter[2]: crossing directions: from outside to inside or viceversa.

The main member functions of the quadric class are:

- double Prod3(double *x, double* y): evaluates the quadratic form $A_{ij}x_iy_j$ on 3d vectors;
- double Prod4(double *x, double* y); evaluates the quadratic form $A_{ij}x_iy_j$ on 4d vectors;
- int Inside(vect3 x): checks if x is inside (0) or outside (1) the quadric;
- int Ellipsoid(double *x0, double* a): builds an ellipsoid quadric;
- int Plane(double *x0, double* u): builds a plane quadric;
- int CilinderX(double *x0, double* a): builds a cylinder parallel to the x axis;
- int CilinderY(double *x0, double* a): builds a cylinder parallel to the y axis;
- int CilinderZ(double *x0, double* a): builds a cylinder parallel to the z axis;
- int Intersect(vect3 x0, vect3 u): intersections of a line with the quadric;
- int Transform(matr4 M): congruence transform of the quadric with matrix M;
- int SetElem(int i, int j, double elem): sets A_{ij} and A_{ji} to the value elem.

5.9 The quadricarray class

The *quadricarray* class contains an array of the quadrics used by the simulation. Its main member variables are:

- int Nquadr: number of quadrics in the array;
- quadric *Quadr: pointer to the quadric array;
- quadric_map QuadricMap: map of the quadrics with their names.

5.10 The qvolume class

A solid *object* is defined as a solid shape demarcated by a set of quadric surfaces that separates the space inside from the space outside it. The quadrics limiting an object must be properly oriented, in such a way that their normal vectors are directed outward with respect to the object itself.

In the current version of the implementation, objects must be convex: the user is expected to take care of splitting non-convex objects into convex ones when using them in the simulation.
An object may contain other objects, or it may be contained into another object, as far as their limiting surfaces are not in contact.
The class used in XRMC to represent three-dimensional objects is called *qvolume*.
The member variables of this class are:

- int Nquadr: number of quadrics delimiting the object;
- int iPhaseIn: index of the phase inside the object;
- int iPhaseOut: index of the phase surrounding the object;
- string PhaseInName: name of the phase inside the object;
- string PhaseOutName: name of the phase surrounding the object;
- quadric **Quadr: array of pointers to the quadrics delimiting the object.

Its main member function is:

- int Intersect(vect3 x0, vect3 u, double *t, int* iph0, int *iph1, int* n_inters): method for finding the intersections of a line with the object.

5.11 The geom3d class

The *geom3d* class contains an array of the three-dimensional objects used in the simulation.
Its main member variables are:

- quadricarray *Qarr: array of quadrics used in the geometric description;
- string QarrName : quadricarray input device name;
- composition *Comp: input composition device;
- string CompName: composition input device name;
- int NQVol: number of 3d objects used in the geometric description;
- int MaxNQVol: maximum number of 3d objects in the geometric description;

- qvolume *Qvol: array of 3d objects;

- string **QvolMap: map of the quadrics delimiting the objects.

The main method specific for the geom3d class is:

- int Intersect(vect3 x0, vect3 u, double *t, int* iph0, int *iph1, int* n_inters): method for finding the intersections of a straight line with all quadrics demarcating the three-dimensional objects.

5.12 The path class

When a photon leaves the source or when it was produced by a scattering or fluorescence emission process, it follows a straight trajectory defined by its position vector $\mathbf{r}_{ph}$ and by its direction vector $\mathbf{u}_{ph}$. The program evaluates the intersection of this trajectory with the quadric surfaces demarcating the objects, and divides it into N_s steps with uniform phases. Each step is a segment of the trajectory delimited by its intersection with different objects. The path class holds information about the intersection of a trajectory with the quadrics and about the segments between consecutive intersections.
The main member variables of the path class are:

- int Nsteps: number of intersections of a trajectory with all the quadrics in the sample;

- double *t: array of the intersections (distances from the starting coordinate);

- double *Step: step lengths between consecutive intersections S_i;

- int *iPh0: array of the entrance phase indexes;

- int *iPh1: array of the exit phase indexes;

- double *Mu: absorption coefficient at each step μ_i;

- double MuL: sum of $\mu_i \times S_i$;

- double *SumMuS: cumulative sum of $\mu_i \times S_i$.

The main member functions are:

- int StepMu(composition *comp): evaluates the absorption coefficient at each step of the intersection;

- double StepLength(int *step_idx, double* weight): extracts the next interaction position using the standard MC approach;

- double WeightedStepLength(int *step_idx, double* weight): extract the next interaction position using the weighted steplength approach.

5.13 The sample class

The *sample* class is a container used to join the information about the sample composition, the geometrical description and the type of interactions that can occur in the sample. The main member variables of this class are:

- basesource *Source: input source device;

- string SourceName: input source name;

- geom3d *Geom3D: input geom3d device;

- string Geom3DName: input geom3d name;

- composition *Comp: input composition device;

- string CompName: input composition device name.

- path *Path: object storing all info about intersections;

- int ScattOrderNum: number of scattering orders;

- int *PhotonNum: event multiplicity for each scattering order;

- int ScattOrderIdx: scattering order index;

- int PhotonIdx: event index.

The main member functions specific of the sample class are:

- int Intersect(vect3 x0, vect3 u); evaluates intersection of a trajectory with the sample objects;

- double LinearAbsorption(vect3 x0, vect3 u): evaluates the absorption coefficient at each step of the intersections;

- int Out_Photon_x1(photon *Photon, vect3 x1): generates an event with a photon forced to end on the point x1.

5.14 The detectorarray class

In general, the program can simulate two-dimensional array detectors with energy binning for each pixel. A single element detector can be simulated as a special case of array detector with only one pixel. The pixel shape can be defined as rectangular or elliptical. The latter possibility is particularly useful when a round, single element detector has to be simulated.
A local coordinate system is associated with the detector, specified by the vector position of its geometric center $\mathbf{r}_d$ and by the orthonormal unit vectors $\mathbf{i}_d$, $\mathbf{j}_d$ and $\mathbf{k}_d$ (see Fig. 2), which are the directions of the local x , y , and z axis, respectively. The local z axis is perpendicular to the detector surface, while the local x and y axes are parallel to the detector rows and columns, respectively.

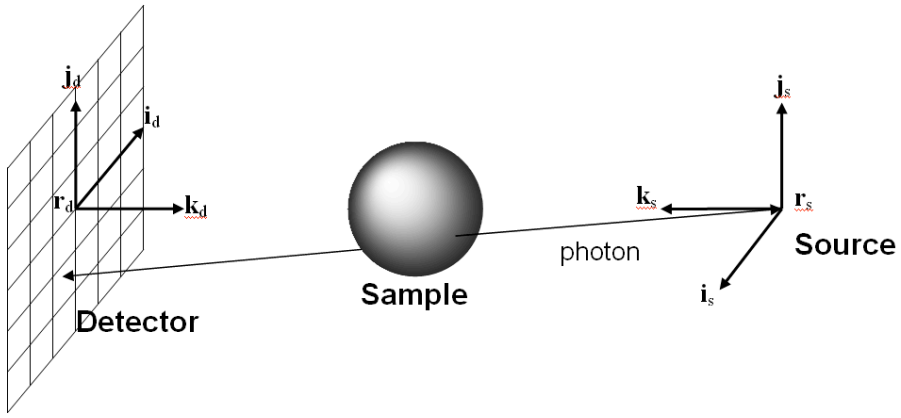

Figure 2: The standard experimental setup simulated by XRMC consists of an x-ray source, a sample and a detector (two-dimensional array or single element). A local coordinate system is attached to the source and to the detector. $\mathbf{r}_s$ and $\mathbf{r}_d$ are the position vector of the source and of the detector geometric center, respectively. $\mathbf{i}_s$, $\mathbf{j}_s$ and $\mathbf{k}_s$ are the directions of the source local x , y , and z axis, respectively. $\mathbf{i}_d$, $\mathbf{j}_d$ and $\mathbf{k}_d$ are the directions of the detector local x , y , and z axis, respectively. The detector local z axis is perpendicular to the detector surface, while the local x and y axes are parallel to the detector rows and columns, respectively.

Two acquisition methods are available:

1. fluence: each channel simply counts the number of photons that it detects;

2. energy fluence: each channel sums up the energy of the photons that it detects.

The energy response of the detector can eventually be taken into account through

using the first method with a sufficient number of energy bins and if a proper post-processing of the acquisition is performed.
The main member variables of the detectorarray class are:

- basesource *Source: input device (typically the sample device);
- string SourceName: name of the input device;
- int NX, NY, N: number of rows (NY), columns (NX) and pixels (NX × NY);
- double PixelSizeX, PixelSizeY, PixelSurf: pixel size and surface (cm²) ;
- int Shape: pixel shape (0: rectangular, 1: elliptical);
- vect3 *PixelX: pixel coordinates array;
- double ***Image: acquired image array;
- double ExpTime: exposure time (seconds);
- int PhotonNum: multiplicity of simulated events per detector pixel;
- int Nbins: number of energy bins;
- int PixelType: pixel content type: 0: fluence, 1: energy fluence, 2: fluence with energy binning, 3: energy fluence with energy binning;
- int SaturateEmin: flag to saturate energies lower than Emin;
- int SaturateEmax: flag to saturate energies greater than Emax;
- double Emin, Emax: minimum and maximum bin energy

The main member functions are:

- int Acquisition(): run the acquisition;
- int Clear(): clear the detector pixel bin contents;

6 Examples

The examples can be found after unpacking the tarball in the subdirectories of examples. The examples are described by comments in the corresponding input files. To run an example, go to the corresponding subdirectory and type the command:

```
xrmc input.dat
```

At the end of the simulation, the output will be stored in the file *image.dat* (or *output.dat* for the fluor_layers example). The output of the examples are either images in raw binary format or measured spectra in ascii format. The raw binary images can be opened with any image visualization program able to import such format. For instance, the figures in this document have been produced using ImageJ, which is a public domain image processing program, freely available for several platforms. In case you want to open the output images of the examples using this program, select from the menu:

file → import → raw…

Select the image and use the following settings in the import form:

- Image type: 64-bit real;
- Width: N. of columns;
- Height: N. of rows;
- Offset to first image: 60 if the image contains the header, 0 otherwise;
- Number of images: N. of scattering orders (1 if only transmission was simulated);
- Little endian byte order: depends on the architecture of your system;

6.1 Cylindrical shell

Directory: cylind_shell
To simulate a cylindrical shell using only convex objects, two objects have been used:

- An external full cylinder, delimited by two planes;
- An internal empty cylinder (the phase inside it is vacuum, phase index=0), delimited by other two planes very close to those of the external cylinder but separated from them by a small gap

The following figure shows the output image. Image size: 400 × 400

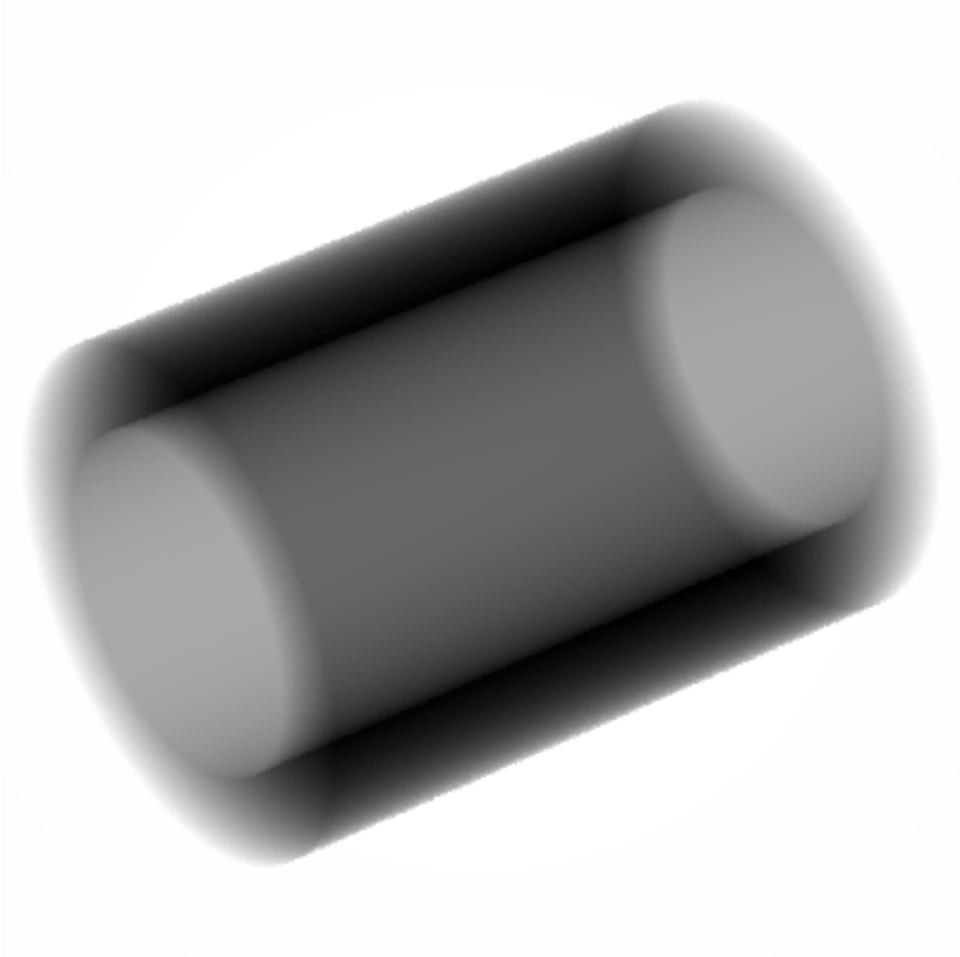

Figure 3: Example of simulation of radiographic images of a cylindrical shell. This example is described by comments in the corresponding input files.

6.2 Star shapes

Directory: star1 and star2
In this example we will demonstrate two ways to generate a five-pointed star shape. A first method consists of defining an outer pentagon (0 on the subfigure a) that receives the phase with the composition of interest. Next, five triangles are defined that are positioned within the pentagon, and are associated with the Vacuum phase. After ensuring that the gap between the pentagon and the adjacent sides of the triangles is sufficiently small, the simulation yields the expected five-pointed star. Subfigure b shows the different planes that were required to obtain this model. All objects are delimited by two planes parallel to the plane of the figure. The output image is shown in the next figure. Image size: 400 × 400

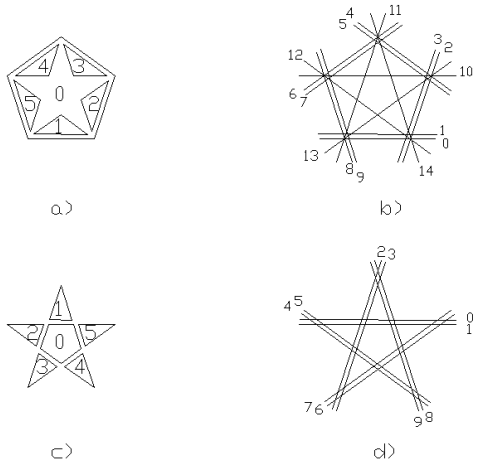

Figure 4: Schematic representations of two ways of building a star shape by combining convex objects. In the quadric definition files, the planes between adjacent objects are separated by a very small gap, however in this figure their separation is increased for clarity. (a) Geometrical objects used to compose the star shape described in example 2. (b) Planes used to define the geometrical object surfaces described in example 2. (c) Geometrical objects used to compose the star shape described in example 3. (d) Planes used to define the geometrical object surfaces described in example 3. All objects are delimited by two planes parallel to the figure plane. Those examples are described by comments in the corresponding input files.

The second method considers a different approach (subfigures c and d): the concave star is assembled from 6 convex objects (5 triangles and 1 pentagon). All 6 are associated with the phase with the composition of interest. As with the first method, all objects are delimited by two planes parallel to the plane of the figure.

6.3 Different types of quadrics

Directory: quadrics
Shown in the next figure. Image size: 400 × 400

6.4 A wheel shape

Directory: wheel
A wheel is built by combining different cylinders and using the rotation commands. The output image is shown in the next figure. Image size: 400 × 400

Figure 5: Example of simulation of a radiographic image: star shape

6.5 Objects with different compositions

Directory: materials
Four cylinders having different compositions (polymethyl methacrylate, adipose tissue, water, bone-equivalent plastic) and placed inside a polymethyl-methacrylate frame, are simulated.
The following figure shows the output image. Image size: 400 × 400

6.6 Anisotropicsource and intensityscreen

Directory: anisotropicsource
The sample simulated here is a cylinder, as was already discussed in a previous section. The intensity distribution is produced by a program in the sample directory that can obtained by compiling the *intensityscreen*image.c_ file. In this case

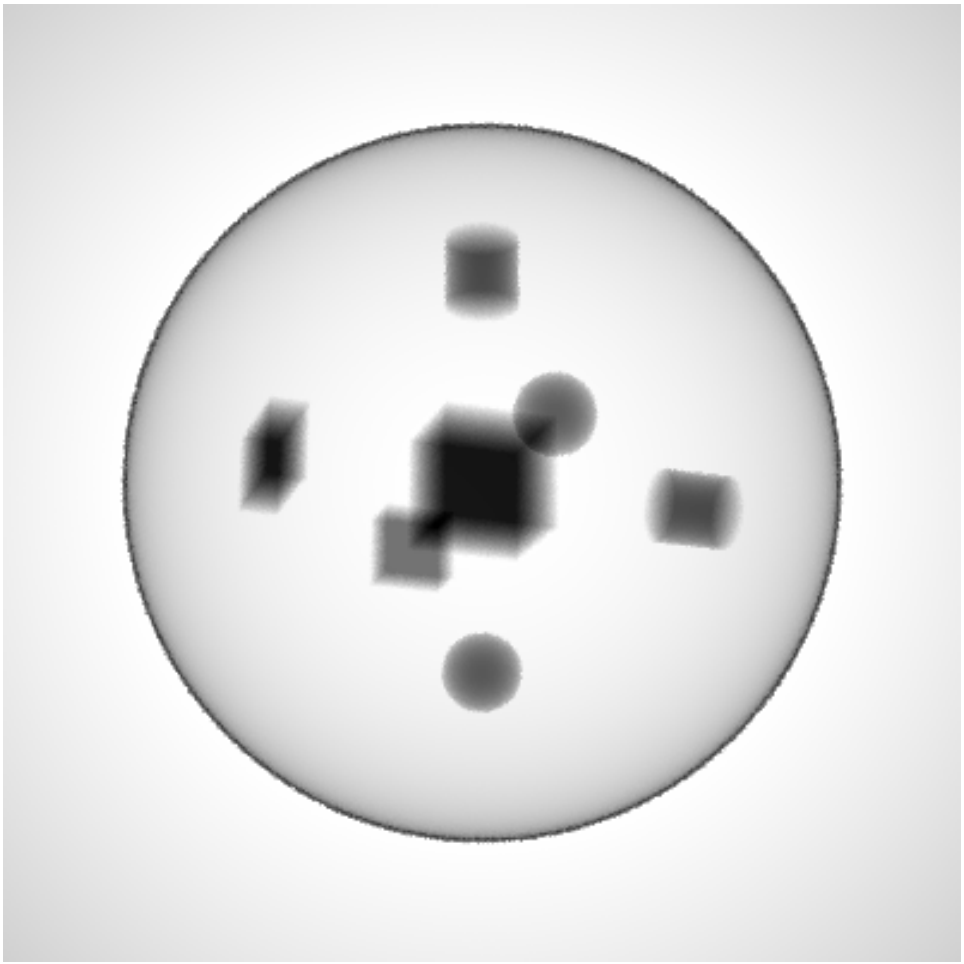

Figure 6: Example of simulation of a radiographic image: a thin spherical shell containing different types of quadrics

it is assumed that the (ideal) screen is placed at 115 cm removed from the source and divided in 100 × 100 pixels. The intensity on this ideal screen is distributed according to a two-dimensional Gaussian function with $\sigma_x = 4$ cm and $\sigma_y = 2$ cm. A README file can be found in the directory of this example containing instructions on how to compile and run the program.
The output image is shown in the following figure. Image size: 400 × 400.

6.7 Beamsource and beamscreen

Directory: beamsource
The sample simulated here is a cylinder, as was already discussed in a previous section. The program *beamscreen*image.c_ produces the file with the intensity and

Figure 7: Example of simulation of a radiographic image: a wheel

energy distribution on an ideal screen placed at 115 cm distance from the source and divided in 100 × 100 pixels. The intensity on this ideal screen is distributed according to a two-dimensional Gaussian function, with σ_x = 4 cm and σ_y = 2 cm. The energy spectrum depends on the radial distance r from the screen center, and it is a gaussian function with σ = 4 keV and centered in

$$E_c = E_{c0} + (E_{c1} - E_{c0}) \times \frac{r}{L} \tag{8}$$

where E_{c0} = 50 keV, E_{c1}=100 keV and L is the half-side of the screen. A README file in the same directory explains how to compile and run the program beam-screen_image.c.
The simulation can be run by typing the command:

xrmc input.dat

Figure 8: Example of simulation of a radiographic image: four cylinders having different compositions and placed inside a frame.

The output image is stored in the file *image.dat*.
The file spectrum_detector.dat defines an energy sensitive detector with 25 energy bins, which can be used to visualize the intensity as a function of the energy. The simulation can be run by typing the command:

```
xrmc input_spectrum.dat
```

The results are stored in the file image_spectrum.dat, which contains a stack of 25 images, one for each energy bin. The following igure shows the output image for the central bin (energy between 74 and 76 keV). Image size: 400 × 400

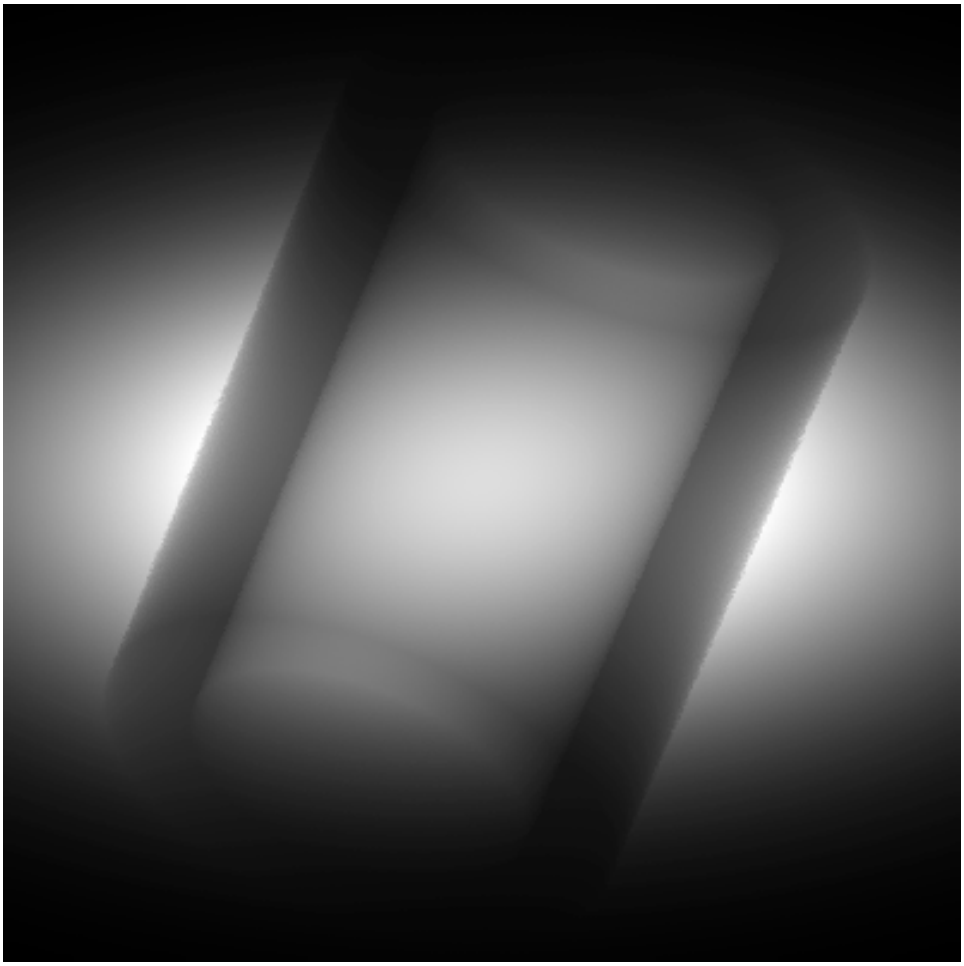

Figure 9: Example of image produced using the anisotropicsource/intensityscreen devices. This example is described by comments in the corresponding input files.

6.8 Free-propagation phase contrast imaging

Directory:
The sample is a 100 mm diameter PMMA (Polymethyl methacrylate) wire. The source-object distance z_1 and the object-detector distance z_{12} are 23 m and 3.5 m, respectively. The source size is 100 mm FWHM (42.5 mm standard deviation). The detector pixel size is 0.5 mm. The source is monochromatic, with E = 17 keV. The unconvoluted image is stored in the file *image.dat*. The image convoluted to take into account the source size is stored in the file convoluted_image.dat. The following figure shows the unconvoluted and the convoluted images. Image size: 800 × 800

7 References and addtional resources

7.1 Bibliography

- Voxel-based Monte Carlo simulation of X-ray imaging and spectroscopy experiments. Spectrochimica Acta Part B, 59, 2004. DOI

- A library for X-ray matter interaction cross sections for X-ray fluorescence applications. A. Brunetti, M. Sanchez del Rio, B. Golosio, A. Simionovici and A. Somogyi. Spectrochimica Acta B, 59(10-11), 1725-1731, 2004. DOI

- The xraylib library for X-ray--matter interactions. Recent developments. T. Schoonjans, A. Brunetti, B. Golosio, M. Sanchez del Rio, V. A. Solé, C. Ferrero and L. Vincze. Spectrochimica Acta Part B, 66(11-12), 776-784, 2011. DOI

- Phase contrast imaging simulation and measurements using polychromatic sources with small source-object distances. B. Golosio, P. Delogu, I. Zanette, M. Carpinelli, G. L. Masala, P. Oliva, A. Stefanini, and S. Stumbo. Journal of Applied Physics, 104(9), 2008. DOI

- A General Monte-Carlo Simulation of Energy-Dispersive X-ray-Fluorescence Spectrometers Part 1. Unpolarized Radiation, Homogeneous Samples. Laszlo Vincze, Koen Janssens and Freddy Adams. Spectrochimica Acta Part B, 48(4), 553-573, 1993. DOI

- A General Monte-Carlo Simulation of Energy-Dispersive X-ray-Fluorescence Spectrometers Part 2. Polarized monochromatic radiation, homegeneous samples. Laszlo Vincze, Koen Janssens, Fred Adams, M.L. Rivers and K.W. Jones. Spectrochimica Acta Part B, 50(2), 127-147, 1995. DOI

- A General Monte-Carlo Simulation of Energy-Dispersive X-ray-Fluorescence Spectrometers Part 3. Polarized polychromatic radiation, homogeneous samples. Laszlo Vincze, Koen Janssens, Fred Adams and K.W. Jones. Spectrochimica Acta Part B, 50(12), 1481-1500, 1995. DOI

- A General Monte-Carlo Simulation of Energy-Dispersive X-ray-Fluorescence Spectrometers Part 4. Photon scattering at high X-ray energies. Laszlo Vincze, Koen Janssens, Bart Vekemans and Fred Adams. Spectrochimica Acta Part B, 54(12), 1711-1722, 1999. DOI

- A General Monte-Carlo Simulation of Energy-Dispersive X-ray-Fluorescence Spectrometers Part 5. Polarized radiation, stratified samples, cascade effects, M-lines. Tom Schoonjans, Laszlo Vincze, Vicente Armando Solé, Manuel

Sanchez del Rio, Philip Brondeel, Geert Silversmit, Karen Appel, Claudio Ferrero. Spectrochimica Acta Part B, 70, 10-23, 2012. DOI

- A General Monte-Carlo Simulation of Energy-Dispersive X-ray-Fluorescence Spectrometers Part 6. Quantification through iterative simulations. Tom Schoonjans, Laszlo Vincze, Vicente Armando Solé, Manuel Sanchez del Rio, Karen Appel, Claudio Ferrero. Spectrochimica Acta Part B, 82, 36-41, 2013. DOI

7.2 Useful links

- xraylib
- XMI-MSIM

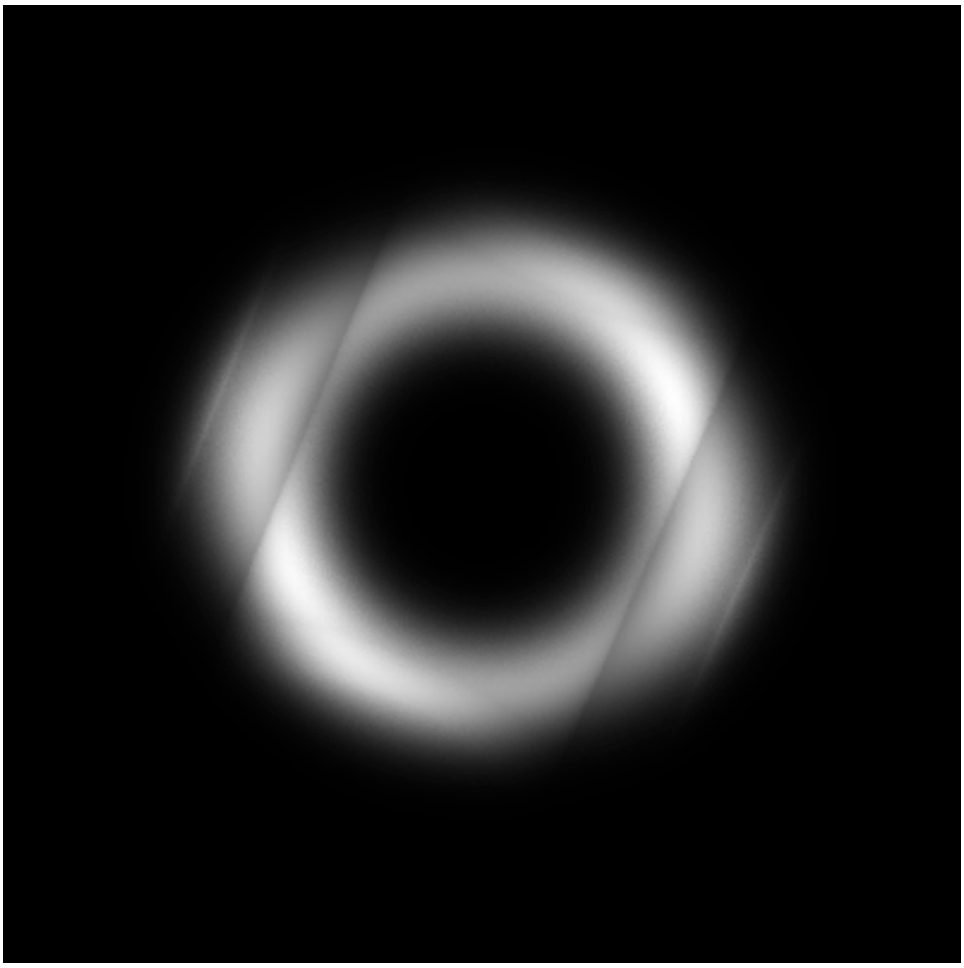

Figure 10: Example of image produced using the beamsource/beamscreen devices. This image represents the central energy bin (energy between 74 and 76 keV). This example is described by comments in the corresponding input files.

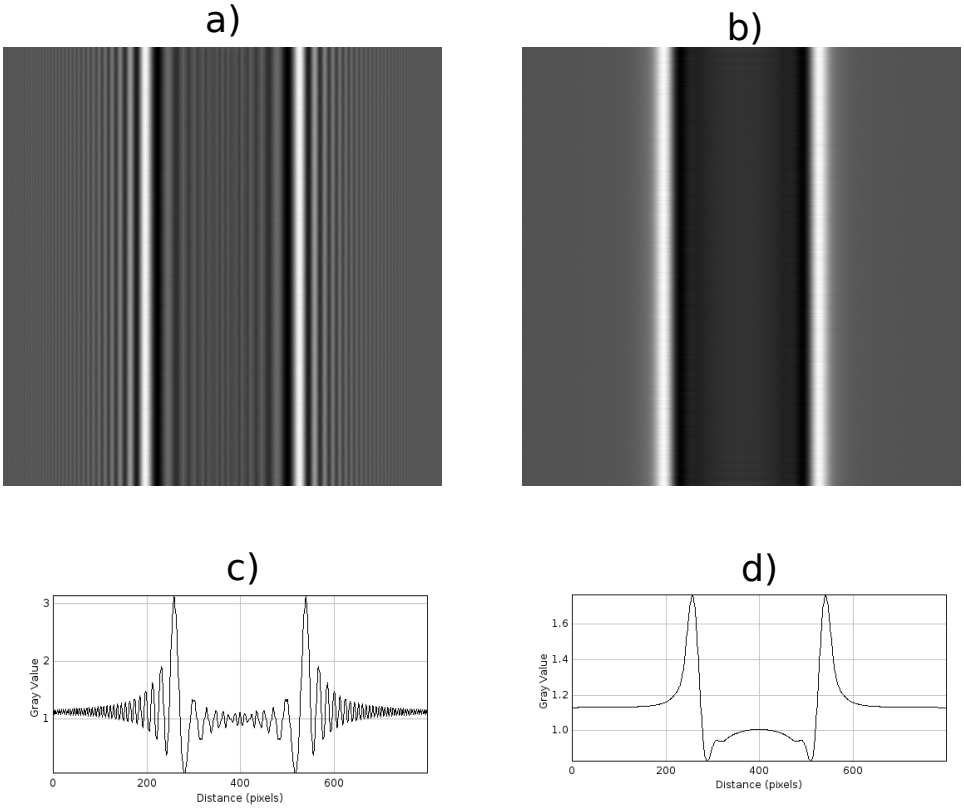

Figure 11: Example of phase-contrast imaging experiment simulation. The sample is a 100 mm diameter PMMA (Polymethyl methacrylate) wire. The source-object distance z1 and the object-detector distance z12 are 23 m and 3.5 m, respectively. The source size is 100 mm FWHM (42.5 mm standard deviation). The detector pixel size is 0.5 mm. The source is monochromatic, with E = 17 keV. Figure (a) represents the unconvoluted image, while figure (b) represents the image convoluted to take into account the source size. Image size: 800 × 800. Figures (c) and (d) represent the intensity profile for the unconvoluted and for the convoluted images, respectively.