Image convolvers in CASToR

October 22, 2024

Foreword

CASToR is designed to be flexible, but also as generic as possible. Any new implementation should be thought to be usable in as many contexts as possible; among all modalities, all types of data, all types of algorithms, etc.

Before adding some new code to CASToR, it is highly recommended to read the general documentation *CASToR_general_documentation.pdf* to get a good overview of the project, as well as the programming guidelines *CASToR__programming_guidelines.pdf*. Also, the philosophy about adding new modules in CASToR (*e.g.* projectors, optimizers, deformations, image processing, etc) is fully explained in *CASToR__add_new_modules.pdf*. Finally, the doxygen documentation is a very good resource to help understanding the code architecture.

1 Summary

This HowTo guide describes how to add your own image convolver into CASToR. CASToR is mainly designed to be modular in the sense that adding a new feature should be as easy and flexible as possible. This guide begins with a rough description of the image convolver part of the CASToR architecture that explains the chosen philosophy. Then follows a step-by-step guide that explains how to add a new image convolver by simply adding a new class with few mandatory requirements.

2 The image convolver architecture

The image convolver part of the code is based on 2 main classes: *oImageConvolverManager* and *vImageConvolver*. The main program will instantiate and initialize the *oImageConvolverManager*. It is in charge of reading command line options and instantiating the different children of *vImageConvolver*. As many convolvers as desired can be used/combined, each one is declared by the option *-conv* when executing the reconstruction code. To get some help on how to use it and a list of the implemented convolvers, execute the program with the option *-help-conv*. The option format follows the philosophy described in *CASToR_HowTo__add_new_modules.pdf*. Any image convolver can be used at different places during the program execution. For iterative algorithms, it can be used on the image to be forward-projected, on the back-projected correction terms, on the current estimated image either as a post-processing step (applied to the image to be saved), or as an intra-iterations processing (the convolved image is put back into the next update as the current estimate).

During the initialization the *oImageConvolverManager* will call the *BuildConvolutionKernel()* function of each image convolver to actually build the convolution kernels. All convolutions are based on a pre-computed convolution kernel which is then used by the *Convolve()* and *ConvolveTranspose()* functions when applying the convolution to images. These functions are already implemented by the *vImageConvolver* abstract class, for stationary kernels. If one wants to implement a non-stationary convolution, one has to overload these functions in the child convolver. The *Convolve()* and *ConvolveTranspose()* are used by the *oImageConvolverManager* to apply the convolution when the user asked for during the execution of the program. More specifically, the *ConvolverTranspose()* function is applied on the back-projected correction terms and on the sensitivity

image, when used within iterative algorithms. In any other cases, the *Convolve()* function is applied.

Below is a more detailed description of how the image convolvers are used and how to add your own.

3   Add your own image convolver

3.1   Basic concept

To add your own image convolver, you only have to build a specific class that inherits from the abstract class *vImageConvolver*. Then, you just have to implement a bunch of pure virtual functions that will correspond to the specific stuff you want your new image convolver to do. Please refer to the *CASToR__add_new_modules.pdf* guide in order to fill up the mandatory parts of adding a new module (your new image convolver is a module); namely the auto-inclusion mechanism, the interface-related functions and the management functions. Right below are some instructions to help you fill the specific pure virtual functions of your image convolver.

To make things easier, we provide an example of template class that already implements all the squeleton. Basically, you will have to change the name of the class and fill the functions up with your own code. The actual files are *include/image/iImageConvolverTemplate.hh* and *src/image/iImageConvolverTemplate.cc* and are actually already part of the source code.

3.2   Implementation of the specific functions

There are 3 functions of interest:

BuildConvolutionKernel() : This function is pure virtual in *vImageConvolver* so it has to be implemented by the specific image convolver inheriting from it.

Convolve() : This function is virtual but has an implementation within *vImageConvolver*. However it is only designed for stationary kernels. It has to be overloaded in the case of non-stationary kernels.

ConvolveTranspose() : This function is virtual but has an implementation within *vImageConvolver*. However it is only designed for stationary kernels. It has to be overloaded in the case of non-stationary kernels.

All information and the tools needed to implement these functions are fully described in the template source file *src/image/iImageConvolverTemplate.cc*, so please refer to it.

To be as generic as possible, a convolution is not performed in the Fourrier space but is based on convolution kernels. The kernels built by the *BuildConvolutionKernel()* function are computed during the initialization. The kernels are described using the following members of *vImageConvolver*:

Listing 1: Variables members of vImageConvolver describing the convolution kernels.

```
1   // The number of kernels (1 if stationary, more otherwise
2   INTNB m_nbKernels;
3   // The dimension of each kernel along X
4   INTNB* mp_dimKernelX;
5   // The dimension of each kernel along Y
6   INTNB* mp_dimKernelY;
7   // The dimension of each kernel along Z
8   INTNB* mp_dimKernelZ;
9   // The actual kernels, first pointer for the number of kernels, second ↩
       pointer for the kernel values
10  FLTNB** mpp_kernel;
```

The member $m\_nbKernels$ specifies the number of kernels. In the stationary case, this number is equal to 1. In the non-stationary case, it is greater than 1. For each kernel, its dimensions along each axis are specified by the members $mp\_dimKernelX$, $mp\_dimKernelY$ and $mp\_dimKernelZ$. Finally, the values of each kernel are specified by the member mpp_kernel. The values for a kernel i are specified in the table $mpp\_kernel[i]$. These values are organized as for the images: the 3 spatial dimensions are flattened in a single dimension. Basically, if one wants the kernel value for $X=x$, $Y=y$ and $Z=z$, this is done through:

Listing 2: How to access kernel value for X=x, Y=y and Z=z of kernel i

```
1  INTNB kernel_index = z*mp_dimKernelX[i]*mp_dimKernelY[i] + y*↵
    mp_dimKernelX[i] + x;
2  FLTNB kernel_value = mpp_kernel[i][kernel_index];
```

So inside the *BuildConvolutionKernel()* function, one has to allocate and specify all these members describing the convolution kernels. If the convolution is stationary, this is the only thing to do. In the non-stationary case, the organization of the actual kernels values and positions is up to the user who has to overload the *Convolve()* and *ConvolveTranspose()* functions. Their implementation is also up to the user, so as to be in agreement with how the kernels are organized.

To speed up the convolution operations, the image to be convolved is copied into a padded buffer. Note that this is automatically done by the *oImageConvolverManager* before calling any of the *Convolve()* or *ConvolveTranspose()* functions. The padded buffer is the actual image with some null (zero) values added all around. The amount of zeros added along each dimension is based on the maximum kernel size with respect to this dimension. In the convolution code, all loops can thus be made without any checks, speeding up the execution. To see how it is done for stationary kernels, look at the implementation of the *vImageConvolver::Convolve()* function. So, when implementing the convolution for non-stationary kernels, the image to be convolved is in *mp_paddedImage* member of *vImageConvolver*. Its dimensions and pad offsets can be found in the following members:

Listing 3: Variables members of vImageConvolver describing the padded image buffer.

```
1   // The actual padded buffer image
2   FLTNB* mp_paddedImage;
3   // The offset of the padded image along X
4   INTNB m_offsetX;
5   // The offset of the padded image along Y
6   INTNB m_offsetY;
7   // The offset of the padded image along Z
8   INTNB m_offsetZ;
9   // The number of voxels of the padded image along X
10  INTNB m_dimPadX;
11  // The number of voxels of the padded image along Y
12  INTNB m_dimPadY;
13  // The number of voxels of the padded image along Z
14  INTNB m_dimPadZ;
15  // The number of voxels of the padded image in a slice
16  INTNB m_dimPadXY;
17  // The total number of voxels of the padded image
18  INTNB m_dimPadXYZ;
```

The convolved image has to be written inside the output image provided as a parameter of the *Convolve()* or *ConvolveTranspose()* functions.