How to add a custom optimization algorithm into CASToR and/or another penalty

April 27, 2018

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 picture of the project, as well as the programming guidelines *CASToR_HowTo__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_HowTo__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 optimization algorithm into CASToR and how to define penalties. CASToR is mainly designed to be modular in the sense that adding a new feature should be as easy as possible. This guide begins with a brief description of the optimizer part of the CASToR architecture that explains the chosen philosophy. Then follows one step-by-step guide that explains how to add a new optimizer by simply adding a new class with few mandatory requirements, or a new penalty in the same way.

2 The optimizer architecture

The optimizer part of the code is based on 2 main classes: *oOptimizerManager* and *vOptimizer*. The *oOptimizerManager*, being the manager, is in charge of reading command line options and instantiating a *vOptimizer* with respect to what the user asks for. The *vOptimizer* is an abstract class so only its children can be used as actual optimizers. It corresponds to the optimization algorithm used within the iterative process, such as MLEM for example. In CASToR, all iterative algorithms are decomposed in two steps that we call *data update step* and *image udpate step*. The *data update step* corresponds to the operations performed on each event in the data space, used to compute the correction terms in this space that are then backward projected into a correction image. The *image update step* corresponds to the operations performed on each voxel of the image space, where the correction terms gathered in the image space during the *data update step* are used to compute the new voxel value also based on the sensitivity and the current voxel value. Back to the code, the main program will instantiate and initialize the *oOptimizerManager*, and during the reconstruction process, four functions of the *oOptimizerManager* will be used:

PreDataUpdateStep() : This function is called at the beginning of a subset, before the loop over all events. It basically calls the eponym function of *vOptimizer* which itself calls the *PreDataUpdateSpecificStep()* function. This last function does nothing on purpose but being virtual, so any specific optimizer can overload it to perform specific operations at this step.

DataUpdateStep() : This function is called inside the loop over all events, for each event. It calls a series of functions of *vOptimizer* that split up the update step in the data space in smaller steps: 1. forward projection, 2. optional step, 3. backward projection of the sensitivity for histogram data, 4. optional step, 5. compute corrections, 6. optional step, 7. backward projection of corrections, 8. compute FOMs. See below for a description of these functions.

PostDataUpdateStep() : This function is called at the end of a subset, after the loop over all events. It basically calls the eponym function of *vOptimizer* which itself calls the *PostDataUpdateSpecificStep()* function. This last function does nothing on purpose but being virtual, so any specific optimizer can overload it to perform specific operations at this step.

ImageUpdateStep() : This function is called at the end of a subset, after the *PostDataUpdateStep()* function, and is used to perform the image update step by calling the eponym function from *vOptimizer*. Its aim is to apply the image correction factors computed from all the calls to the *DataUpdateStep()* function.

Within the *oOptimizerManager::DataUpdateStep()* function, the following functions from *vOptimizer* are called in this order:

DataStep1ForwardProjectModel() : This function performs the forward projection of the current image, taking all dynamic dimensions through their basis functions into account. It applies all multiplicative corrections included in the system matrix and add the additive terms to the result, so that the latter is directly comparable to the recorded data. It can deal with emission or transmission data natively.

DataStep2Optional() : This function does nothing but being virtual, so that it can be overloaded by specific optimizers if needed.

DataStep3BackwardProjectSensitivity() : This function calls the pure virtual *SensitivitySpecificOperations()* function to compute the weight associated to the current LOR and backward projects this weight into the sensitivity image, taking all multiplicative terms from the system matrix into account. It is called only when using histogram data.

DataStep4Optional() : This function does nothing but being virtual, so that it can be overloaded by specific optimizers if needed.

DataStep5ComputeCorrections() : This function calls the pure virtual *DataSpaceSpecificOperations()* function to compute the correction terms associated to the current event.

DataStep6Optional() : This function does nothing but being virtual, so that it can be overloaded by specific optimizers if needed.

DataStep7BackwardProjectCorrections() : This function performs the backward projection of the previously computed correction terms into the so-called backward image to gather the corrections from all events into the image space. It takes all dynamic dimensions through their basis functions into account, as well as all multiplicative corrections included in the system matrix.

DataStep8ComputeFOM() : This function computes some figures-of-merit in the data space, if asked for.

The *vOptimizer::ImageUpdateStep()* function performs the *image update step*. It loops over all dynamic basis functions and for each voxel, compute the sensitivity using the *ComputeSensitivity()* function and calls the pure virtual *ImageSpaceSpecificOperations()* function to compute the new image value.

So basically, all operations specific to an optimizer are performed within the three following pure virtual functions:

SensitivitySpecificOperations() : From the current data, forward model, projection line, etc, it must compute the weight associated to the current event. This function is useful only for histogram data and thus for optimizers compatible with histogram data.

DataSpaceSpecificOperations() : From the current data, forward model, projection line, etc, it must compute the correction term associated to the current event that will be back-projected. It can deal with multiple values as specified by the member variable *m_nbBackwardImages*.

ImageSpaceSpecificOperations() : From the current voxel value, the correction value(s) and the sensitivity, it must compute the new image value.

3 Implemented optimizers

Several optimizers have already been implemented, this includes MLEM (for transmission and list-mode and histogrammed emission data), MLTR (for histogrammed transmission data), Landweber (for histogrammed emission and transmission data), NEGML and AML (for histogrammed emission data). For a complete and exhaustive list of all available optimizers, use the related help option directly within the CASToR program.

Note that the current generic iterative algorithms can use subsets of the data. Any optimizer can thus benefit from the use of subsets. See the general documentation for a detailed description of how the iterative algorithm uses subsets of the data.

4 Add your own optimizer

4.1 Basic concept

To add your own optimizer, you only have to build a specific class that inherits from the abstract class *vOptimizer*. Then, you just have to implement a bunch of pure virtual functions corresponding to what you want your new optimizer to specifically realize. Please refer to the *CASToR_HowTo__add_new_modules.pdf* guide in order to fill up the mandatory parts of adding a new module (your new optimizer 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 optimization functions of your optimizer.

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/optimizer/iOptimizerTemplate.hh* and *src/optimizer/iOptimizerTemplate.cc* and are actually already part of the source code. Also, we recommend that you take a look at other implemented optimizers.

4.2 Implementation of the optimization functions

The optimization functions that you have to implement are the three ones mentionned in the previous section: *SensitivitySpecificOperations()*, *DataSpaceSpecificOperations()* and *ImageSpaceSpecificOperations()*. All information and the tools needed to implement these functions are fully described in the template source file *src/optimizer/iOptimizerTemplate.cc*, so please refer to it. Aside these three pure virtual functions, there are many virtual functions whose implementation in *vOptimizer* do nothing on purpose, but that can be overloaded to perform other types of actions specific to your optimizer. There are all optional functions that are included in the *data update step*; their name are *vOptimizer::DataStepXOptional()*, where X is the sub-step number defining when they are called in the *data update step* process. To perform specific operations inside a subset before and/or after the loop on all events, there are the functions *vOptimizer::PreDataUpdateSpecificStep()* and *vOptimizer::PostDataUpdateSpecificStep()* respectively.

For optimizers highly differing from the way the *vOptimizer* was thought, all functions related to the *data update step* and the *image update step* are virtual, so they can be overloaded to implement alternative behaviours. For details about that, look at the doxygen documentation contained in the *include/optimizer/vOptimizer.hh* file.

Finally, for any optimizer, one must specify in the constructor if it is compatible with list-mode and/or histogram data. For example, MLEM is compatible with both types of data, but NEGML is only compatible with histogram data. If your optimizer is compatible with list-mode data, set the boolean member *m_listmodeCompatibility* to true in the constructor. If your optimizer is compatible with histogram data, set the boolean member *m_histogramCompatibility* to true in the constructor. By default, these two booleans are set to false in the constructor of *vOptimizer*.

5 Add your own penalty

This feature is not yet implemented in the CASToR code.