Optimizers and penalties in CASToR

September 18, 2020

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__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 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 and penalty parts of the CASToR architecture that explains the chosen philosophy. Then follows a 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 and penalty architectures

In CASToR, an optimizer is defined as an optimization algorithm specific to an objective function. In other words, the objective function and the optimization algorithm are regarded as a whole and not separately.

The optimizer part of the code is based on 3 main classes: *oOptimizerManager*, *vOptimizer* and *vPenalty*. The *oOptimizerManager*, being the manager, is in charge of reading command line options and instantiating a *vOptimizer* and a *vPenalty* 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, that does not include a penalty term, or OSL (One-Step-Late), that includes a penalty term.

The *vPenalty* is also an abstract class, so only its children can be used as actual penalties. It corresponds to the penalty term included in the objective function, that is added to the data-fidelity term, and that some optimizers can take into account.

In CASToR, all iterative algorithms are decomposed in two main 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 considering the sensitivity, the penalty and the current voxel value.

The main program instantiates and initializes 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.

PreImageUpdateStep(): This function is called between the loop over all events and the image update step. It basically calls the eponym function of *vOptimizer* which itself calls the *PreImageUpdateSpecificStep()* 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. For optimizers that include a penalty term, this is where the penalty's functions are called (see below).

ImageUpdateStep(): This function is called at the end of a subset, after the *PreImageUpdateStep()* function. It 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 and the potential penalty.

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 first calls the pure virtual *SensitivitySpecificOperations()* function to compute the weight associated to the current event. It then 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 so that an event corresponds to a histogram bin.

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 intrinsic 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 without penalty 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 used 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.

When a penalty is included, some computations specific to the penalty need to be done before the *image update step*. Such computations are implemented by the specific optimizer by overloading the *PreImageUpdateSpecificStep()* function of the *vOptimizer* that does nothing by default. The penalty itself is implemented inside the abstract class *vPenalty* which is instantiated and parameterized by the *oOptimizerManager* class, and then managed by the specific optimizer. Inside the *PreImageUpdateSpecificStep()* function of the specific optimizer, the 5 following functions from the *vPenalty* may be used:

GlobalPreProcessingStep(): This function does nothing by default but being virtual, so it can be overloaded by the specific penalty. It is designed to be called outside of the loops over dynamic and spatial dimensions.

LocalPreProcessingStep(): This function does nothing by default but being virtual, so it can be overloaded by the specific penalty. It is designed to be called inside the loops over dynamic and spatial dimensions as it takes all indices as parameters.

ComputePenaltyValue(): This function is a pure virtual function, so it has to be implemented by the specific penalty. It takes all dynamic and spatial indices as parameters and is supposed to return the value of the penalty term for these indices.

ComputeFirstDerivative(): This function is a pure virtual function, so it has to be implemented by the specific penalty. It takes all dynamic and spatial indices as parameters and is supposed to return the value of the first derivative of the penalty term for these indices.

ComputeSecondDerivative(): This function is a pure virtual function, so it has to be implemented by the specific penalty. It takes all dynamic and spatial indices as parameters and is supposed to return the value of the second derivative of the penalty term for these indices.

Optimizers that include a penalty term may not need the second derivative. Some penalties may also not be twice differentiable. To deal with that, any optimizer that includes a penalty term has to specify the minimum required derivative order of the penalty. Any penalty also has to specify its own derivative order. A compatibility check is then performed during the initialization by the *oOptimizerManager*. Nonetheless, all penalties have to implement the three pure virtual functions

ComputePenaltyValue(), *ComputeFirstDerivative()* and *ComputeSecondDerivative()*. Even if some penalties may not strictly require to compute the penalty value for optimization purposes, this function is used to compute the objective function for information purpose when the user asks for it. If a penalty is not twice differentiable, the *ComputeSecondDerivative()* function is left empty, because it will not be called.

3 Implemented optimizers and penalties

Implemented optimizers that do not admit a penalty term include:

- MLEM (for histogrammed transmission and emission data and list-mode emission data),
- MLTR from Van Slambrouck (for histogrammed transmission data),
- Landweber (for histogrammed emission and transmission data),
- NEGML from Nuyts (for histogrammed emission data),
- AML from Byrne (for histogrammed emission data).

Implemented optimizers that admit a penalty term include:

- One-Step-Late from Green (for histogrammed transmission and emission data and list-mode emission data),
- Penalized Preconditioned Gradient ML from Nuyts (for histogrammed emission data),
- BSREM II from Ahn and Fessler (for histogrammed emission data),
- Modified EM for penalized ML from De Pierro (for histogrammed emission data).

Implemented penalties include:

- Markov Random Field penalizing differences between neighbors (including different neighborhood shapes, proximity factors, Bowsher's weights, and several potential functions),
- Median Root Prior (including different neighborhood shapes).

For a complete and exhaustive list of all available optimizers and penalties, use the related help options 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 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 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 the 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::PreImageUpdateSpecificStep()* respectively. If the optimizer admits a penalty term, you will most likely have to overload the *PreImageUpdateSpecificStep()* function to perform instructions related to the computation of this penalty term.

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, there are different variables that must be set in the constructor, according to what the optimizer can do. First, one must specify if the optimizer 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. Second, one must specify if the optimizer is compatible with emission and/or transmission data. For example, MLEM is compatible with both types of data, but MLTR is only compatible with transmission data and NEGML with emission data. If your optimizer is compatible with emission data, set the boolean member *m_emissionCompatibility* to true in the constructor. If your optimizer is compatible with transmission data, set the boolean member *m_transmissionCompatibility* to true in the constructor. By default, all these booleans are set to false in the constructor of *vOptimizer*. Third, one must specify if the optimizer admits a penalty term, and if so, what is the minimum derivative order that the penalty must admit for being used with the optimizer. If your optimizer admits a penalty term, set the integer member *m_requiredPenaltyDerivativesOrder* to the required minimum derivative order that the penalty must admit. By default, this value is set to -1 in the constructor of *vOptimizer* (which means that the optimizer does not admit a penalty term).

5 Add your own penalty

5.1 Basic concept

To add your own penalty, you only have to build a specific class that inherits from the abstract class *vPenalty*. Then, you just have to implement a bunch of pure virtual functions corresponding to what you want your new penalty to specifically 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 penalty 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 penalty.

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/iPenaltyTemplate.hh* and *src/optimizer/iPenaltyTemplate.cc* and are actually already part of the source code. Also, we recommend that

you take a look at other implemented penalties.

5.2 Implementation of the specific penalty functions

First, one must specify the derivative order of the penalty. This must be done in the constructor by specifying the value of the integer member *m_penaltyDerivativesOrder*.

Then, the functions that you have to implement are the three following ones: *ComputePenaltyValue()*, *ComputeFirstDerivative()* and *ComputeSecondDerivative()*. If the penalty admits strictly less than two derivatives, then the *ComputeSecondDerivative()* function must still be implemented (because it is pure virtual) but can be left empty as it will never be called if the member *m_penaltyDerivativeOrder* is appropriately set. Some information is provided in the template source file *src/optimizer/iPenaltyTemplate.cc*, so please refer to it. See also already implemented penalties to get a good understanding.

Aside these three pure virtual functions, there are two virtual functions whose implementation in *vPenalty* do nothing on purpose, but that can be overloaded to perform other types of actions specific to your penalty. These functions are *GlobalPreProcessingStep()* and *LocalPreProcessingStep()*, as explained above.