How to use the itk::NaryElevateImageFilter and itk::UnaryRetractImageFilter for the post-processing of medical 4D data

Stefan Roettger
Siemens Medical MR

1.Introduction

The subject of the paper is to introduce a new class of 4D filters to itk.While it is very easy to load a 3D volume stored as a series of DICOM images with ITK, some convenience functions that are already available for 3D volumes are not yet available for the 4D case. Especially for MR data (and to a certain limit also for CT data) it is very important to look at the full 4D series, since the radiologist usually does not make a diagnosis based on a single volume but on the full time series of volumes which shows the temporal flow of the contrast agent. As an example, for breast imaging the radiologist typically examines parameter maps like MIPt (maximum intensity over time), TTP (time to peak) or WI (Wash-In). These parameter maps are the 3D result of post-processing the entire 4D time series. The goal of proposing the denoted 4D filters is to provide ITK with functionality that enables a postprocessing job to handle the entire 4D series as one entity and not as a loose collection of 3D volumes.

2. Overview

In version 3.0 of the toolkit ready-to-use functionality is missing to take a series of 3D volumes and combine them into a single 4D image. The proposed itk:NaryElevateImageFilter is designed for this particular job. Its name comes from the fact that it is closely related with the itk::NaryFunctorImageFilter with the difference that it also elevates the dimension of the input data (e.g. 3) to the next higher dimension (e.g. 4D). In this context, a postprocessing job would be just another filter operation on the generated 4D output image with a 3D volume as the final result. Several subsequent filter operations can be exectuted after each other on the same 4D input image. The proposed itk::UnaryRetractImageFilter is performing one of these post-processing jobs which means that it retracts the input data from the input dimension (e.g. 4) to the next lower dimension (e.g. 3). It is also closely related to the itk::UnaryFunctorImageFilter with the difference that the input and output dimension do not need to match.

The post-processing is understood to be independent for each voxel meaning that the value of the output voxel is the result of a computation on the list of the voxel's values for each time point. This list of values is passed to a template functor which performs the actual computation. The retract filter calls the functor for each voxel, thus processing the entire 4D volume. For interface completeness, the functor also receives the list of time points.

Both proposed filters borrow streaming support from the super-class. Streaming saves memory by processing the intermediate 4D image chunk by chunk. This can be achieved by appending an itk::StreamingFilter to the end of the pipline. Caution needs to be taken if the input volumes are loaded with the built-in ITK DICOM reader. This reader is not (yet) able to stream, therefore the

streaming will be disabled for the entire pipeline. The general pipeline setup is shown in the following figure:

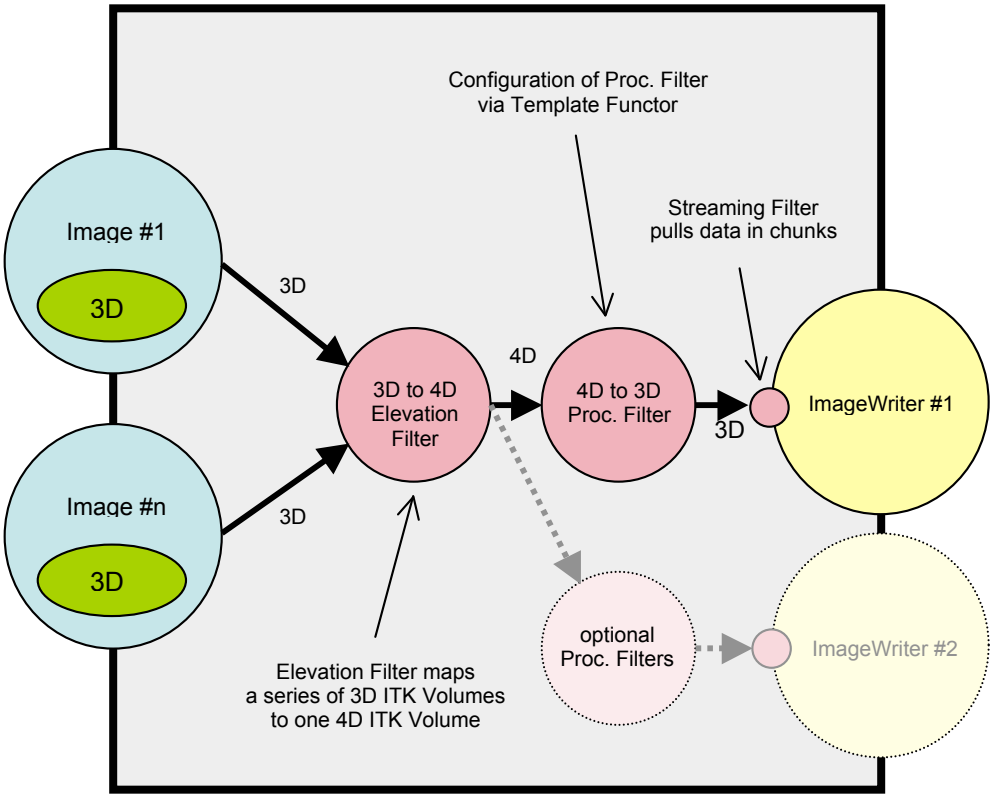

Figure 1: Schematic 4D Pipeline Setup

3. Example

In this chapter we show how to set up a simple 4D processing pipeline to generate a MIPt parametric map.

First we declare the image types we are going to use:

```
// declare images
typedef signed short PixelType;
typedef itk::Image<PixelType, 2> Image2DType;
typedef itk::Image<PixelType, 3> Image3DType;
typedef itk::Image<PixelType, 4> Image4DType;
```

Then we analyse the 4D input data and separate all DICOM images with a different time stamp from each other. Foreach volume with a different time stamp a series id is generated so that the 4D series is represented as a list of series ids. An approriate DICOM tag for the separation is the aquisition number:

```
// create name generator and attach to reader
typedef itk::GDCMSeriesFileNames NamesGeneratorType;
NamesGeneratorType::Pointer nameGenerator=NamesGeneratorType::New();
nameGenerator->SetUseSeriesDetails(true);
nameGenerator->AddSeriesRestriction("0020|0012"); // acquisition number
nameGenerator->SetDirectory(argv[1]);
```

```
  // get series IDs
  typedef std::vector<std::string> SeriesIdContainer;
  const SeriesIdContainer &seriesUID=nameGenerator->GetSeriesUIDs();
```

Now we load the time series by generating an array of DICOM readers (one reader for each volume):

```
  // create reader array
  typedef itk::ImageSeriesReader<Image3DType> ReaderType;
  ReaderType::Pointer *reader=new ReaderType::Pointer[seriesUID.size()];
```

After that we declare the elevation filter which will consume the output of the reader array:

```
  // create elevation filter
  typedef itk::NaryElevateImageFilter<Image3DType, Image4DType> NaryElevateFilterType;
  NaryElevateFilterType::Pointer elevateFilter=NaryElevateFilterType::New();
```

Then we connect each reader with the appropriate series id and also connect the output of the readers with the inputs of the elevation filter:

```
  // declare series iterators
  SeriesIdContainer::const_iterator seriesItr=seriesUID.begin();
  SeriesIdContainer::const_iterator seriesEnd=seriesUID.end();
  int seriesNum=0;

// connect each series to elevation filter
  while (seriesItr!=seriesEnd)
  {
    reader[seriesNum]=ReaderType::New();

    typedef itk::GDCMImageIO ImageIOType;
    ImageIOType::Pointer dicomIO=ImageIOType::New();
    reader[seriesNum]->SetImageIO(dicomIO);

    typedef std::vector<std::string> FileNamesContainer;
    FileNamesContainer fileNames;

    fileNames=nameGenerator->GetFileNames(seriesItr->c_str());
    reader[seriesNum]->SetFileNames(fileNames);

    elevateFilter->SetInput(seriesNum,reader[seriesNum]->GetOutput());
    reader[seriesNum]->ReleaseDataFlagOn();

    reader[seriesNum]->Update();
    reader[seriesNum]->GetOutput()->SetMetaDataDictionary(dicomIO->GetMetaDataDictionary());

    seriesNum++;

    ++seriesItr;
  }
```

Now we basically have the 4D series ready to use. As an example, we can tell the filter to actually load the data into memory (by calling Update() on the filter) and retrieve some useful statistics about the 4D data:

```
  // get statistics from elevate filter
  elevateFilter->Update();
  float dataMin=elevateFilter->GetDataMin();
  float dataMax=elevateFilter->GetDataMax();
  float dataMean=elevateFilter->GetDataMean();
  float noiseThres=elevateFilter->GetNoiseThres();
```

As the second part of the post-processing pipeline we now create a retraction filter that performs a specific processing task on the 4D data. In our example this is a MIPt computation:

```
  // create retraction filter
  typedef itk::UnaryRetractImageFilter<Image4DType, Image3DType,
     itk::Functor::CalcMin<typename Image4DType::PixelType, typename Image3DType::PixelType>>
     UnaryRetractFilterType;
  UnaryRetractFilterType::Pointer retractFilter1=UnaryRetractFilterType1::New();
```

The actual computation is passed to the rectraction filter by means of a template functor. Foreach voxel the functor receives an array of scalar values and an array of time points which describe the temporal behavior of the voxel's measured scalar value. The MIPt processing operation just returns the maximum value of the scalar array (this applies for T1 weighted MR measurements, but for T2* weighted MR measurements one needs to take the minimum):

```
// MIPt (T2* weighted measurement)
template <class TInput, class TOutput>
class CalcMin
{
public:

  typedef typename NumericTraits<TInput>::ValueType ValueType;

  CalcMin() {}
  ~CalcMin() {}

  inline TOutput operator()(const Array<TInput> &A,const Array<double> &T)
  {
    ValueType result=A[0];

    for (unsigned int i=1; i<A.size(); i++)
    {
      ValueType temp=A[i];
      if (temp<result) result=temp;
    }

    return static_cast<TOutput>(result);
  }
};
```

As the last step, we connect the retract filter to the 4D input:

```
  // connect elevation to retract filters
  retractFilter1->SetInput(elevateFilter->GetOutput());
```

And finally we create a DICOM writer that saves the result of the computation as a multiframe DICOM file:

```
  // write data to multiframe DICOM
  typedef itk::ImageFileWriter<Image3DType> WriterType;
  WriterType::Pointer writer=WriterType::New();
  writer->SetFileName("out-MIPt.dcm");
  writer->SetInput(retractFilter->GetOutput());
  writer->Update();
```

For the supplied test data (which is a T2* weighted MR NeuroPerfusion measurement, kindly provided by the University of Würzburg and scanned on a Siemens Symphony MR Scanner), the resulting output 3D DICOM image is depicted below. The image shows slice #5 out of a total of 12 generated slices:

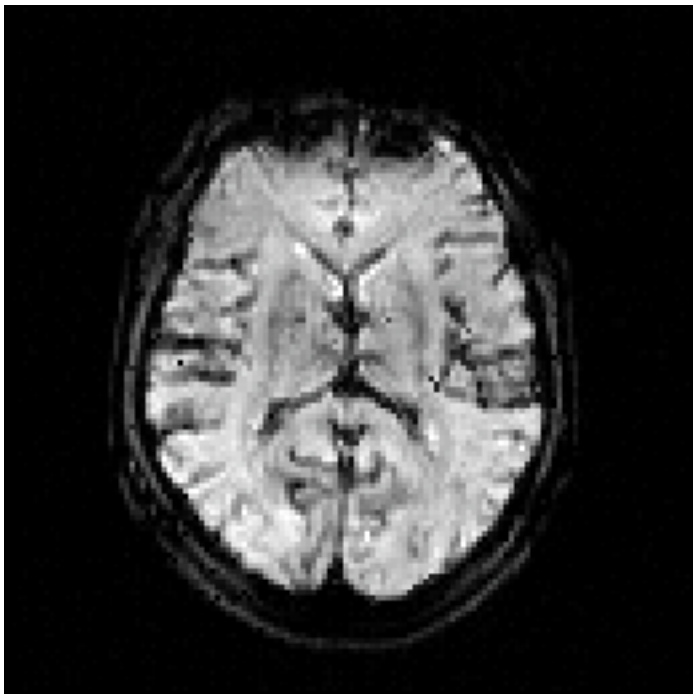

Figure 2: MIPt of NeuroPerfusion MR Data

Voila!

Have Fun
/Stefan