Índice general

The source code for this section can be found in the file

.

The finite element (FEM) library within the Insight Toolkit can be used to solve deformable image registration problems. The first step in implementing a FEM-based registration is to include the appropriate header files.

$$x^2 + 2x + 1 = 0 \tag{1}$$

```
#include "itkFEM.h"
#include "itkFEMRegistrationFilter.h"
```

Next, we use typedefs to instantiate all necessary classes. We define the image and element types we plan to use to solve a two-dimensional registration problem. We define multiple element types so that they can be used without recompiling the code.

```
typedef itk::Image<unsigned char, 2>                       fileImageType;
typedef itk::Image<float, 2>                               ImageType;
typedef itk::fem::Element2DC0LinearQuadrilateralMembrane   ElementType;
typedef itk::fem::Element2DC0LinearTriangularMembrane      ElementType2;
```

Note that in order to solve a three-dimensional registration problem, we would simply define 3D image and element types in lieu of those above. The following declarations could be used for a 3D problem:

```
typedef itk::Image<unsigned char, 3>                     fileImage3DType;
typedef itk::Image<float, 3>                             Image3DType;
typedef itk::fem::Element3DC0LinearHexahedronMembrane    Element3DType;
typedef itk::fem::Element3DC0LinearTetrahedronMembrane   Element3DType2;
```

Here, we instantiate the load types and explicitly template the load implementation type. We also define visitors that allow the elements and loads to communicate with one another.

```

typedef itk::fem::FiniteDifferenceFunctionLoad<ImageType,ImageType> ImageLoadType;
template class itk::fem::ImageMetricLoadImplementation<ImageLoadType>;

typedef ElementType::LoadImplementationFunctionPointer     LoadImpFP;
typedef ElementType::LoadType                              ElementLoadType;

typedef ElementType2::LoadImplementationFunctionPointer    LoadImpFP2;
typedef ElementType2::LoadType                             ElementLoadType2;

typedef itk::fem::VisitorDispatcher<ElementType,ElementLoadType, LoadImpFP>
                                                           DispatcherType;

typedef itk::fem::VisitorDispatcher<ElementType2,ElementLoadType2, LoadImpFP2>
                                                           DispatcherType2;
```

Once all the necessary components have been instantiated, we can instantiate the FEMRegistrationFilter, which depends on the image input and output types.

```
typedef itk::fem::FEMRegistrationFilter<ImageType,ImageType> RegistrationType;
```

The fem::ImageMetricLoad must be registered before it can be used correctly with a particular element type. An example of this is shown below for ElementType. Similar definitions are required for all other defined element types.

```
ElementType::LoadImplementationFunctionPointer fp =
  &itk::fem::ImageMetricLoadImplementation<ImageLoadType>::ImplementImageMetricLoad;
DispatcherType::RegisterVisitor((ImageLoadType*)0,fp);
```

In order to begin the registration, we declare an instance of the FEMRegistrationFilter. For simplicity, we will call it registrationFilter.

```
RegistrationType::Pointer registrationFilter = RegistrationType::New();
```

Next, we call registrationFilter-¿SetConfigFileName() to read the parameter file containing information we need to set up the registration filter (image files, image sizes, etc.). A sample parameter file is shown at the end of this section, and the individual components are labeled.

In order to initialize the mesh of elements, we must first create “dummy” material and element objects and assign them to the registration filter. These objects are subsequently used to either read a predefined mesh from a file or generate a mesh using the software. The values assigned to the fields within the material object are arbitrary since they will be replaced with those specified in the parameter file. Similarly, the element object will be replaced with those from the desired mesh.

```
// Create the material properties
itk::fem::MaterialLinearElasticity::Pointer m;
m = itk::fem::MaterialLinearElasticity::New();
m->GN = 0;                    // Global number of the material
m->E = registrationFilter->GetElasticity();  // Young's modulus -- used in the membrane
m->A = 1.0;                   // Cross-sectional area
m->h = 1.0;                   // Thickness
m->I = 1.0;                   // Moment of inertia
m->nu = 0.;                   // Poisson's ratio -- DONT CHOOSE 1.0!!
m->RhoC = 1.0;                // Density

// Create the element type
ElementType::Pointer e1=ElementType::New();
e1->m_mat=dynamic_cast<itk::fem::MaterialLinearElasticity*>( m );
registrationFilter->SetElement(e1);
registrationFilter->SetMaterial(m);
```

Now we are ready to run the registration:

```
registrationFilter->RunRegistration();
```

To output the image resulting from the registration, we can call WriteWarpedImage(). The image is written in floating point format.

```
registrationFilter->WriteWarpedImage(
      (registrationFilter->GetResultsFileName()).c_str());
```

We can also output the displacement fields resulting from the registration, we can call WriteDisplacementField() with the desired vector component as an argument. For a $2D$ registration, you would want to write out both the x and y displacements, and this requires two calls to the aforementioned function.

```
if (registrationFilter->GetWriteDisplacements())
  {
  registrationFilter->WriteDisplacementField(0);
  registrationFilter->WriteDisplacementField(1);
  // If this were a 3D example, you might also want to call this line:
  // registrationFilter->WriteDisplacementField(2);

  // We can also write it as a multicomponent vector field
  registrationFilter->WriteDisplacementFieldMultiComponent();
  }
```