Rotational Registration of Spherical Surfaces Represented as QuadEdge Meshes

Release 1.00

Luis Ibanez[1], Michel Audette[1], Thomas Yeo[2], Polina Goland[2]

June 4, 2009

[1]Kitware Inc., Clifton Park, NY
[2]CSAIL MIT, Boston, MA

Abstract

This document describes a contribution to the Insight Toolkit intended to support the process of registering two Meshes. The methods included here are restricted to Meshes with a Spherical geometry and topology, and with scalar values associated to their nodes.

This paper is accompanied with the source code, input data, parameters and output data that we used for validating the algorithm described in this paper. This adheres to the fundamental principle that scientific publications must facilitate **reproducibility** of the reported results.

Contents

1 Introduction

The Insight Toolkit already provides methods for registering

- Image to Image
- PointSet to Image
- PointSet to PointSet

but it lacks methods for registering one Mesh versus another Mesh.

In this paper we contribute new classes that can be used for performing registration between two spherical meshes, although not all of of the classes in this contribution are restricted to be used on spherical meshes.

2 Overview

The design of these classes follows very closely the one of the Image Registration Framework in ITK. In particular, we have the usual components

- Optimizer
- Metric
- Transform
- Interpolator

and we have the two objects to be registered, in this case Meshes instead of Images. The two main components that must be provided in order to support Mesh registration are Iterpolators and Metrics.

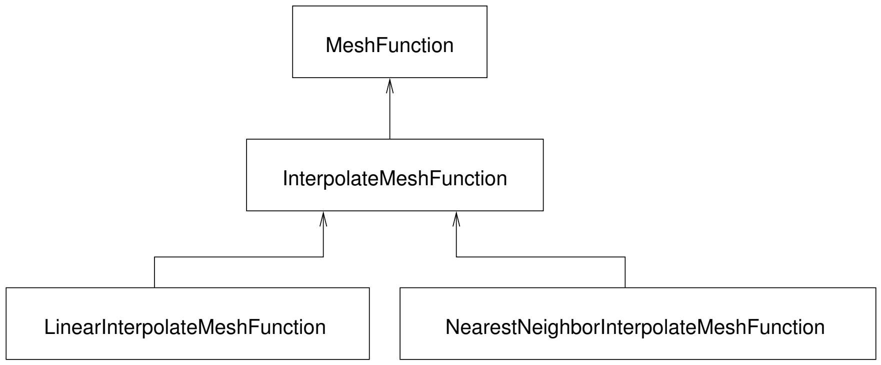
MeshFunction
InterpolateMeshFunction
LinearInterpolateMeshFunction
NearestNeighborInterpolateMeshFunction

Figure 1: Hierarchy of new Mesh Interpolator Classes.

2.1 Mesh Interpolators

Mesh-based Interpolators are designed here following the same structure of the image interpolators. In particular, they derive from the `itk::Function` class, and implement a sequence of

- `itk::MeshFunction`
- `itk::InterpolateMeshFunction`

and finally, the siblings

- `itk::LinearInterpolateMeshFunction`
- `itk::NearestNeighborInterpolateMeshFunction`

The diagram in Figure 1 presents the hierarchy of mesh interpolators that are included in this contribution.

2.2 Mesh Metrics

Mesh metrics follow a similar structure to the existing image metrics in ITK. Their hierarchy starts with the `itk::MeshToMeshMetric` that derives from the existing ITK class `itk::SingleValuedCostFunction`. They currently comprise an abstract base class and a concrete class, as follows:

- `itk::MeshToMeshMetric`
- `itk::MeanSquaresMeshToMeshMetric`

It is expected that variations of Mesh metrics will be added as siblings of the mean squares metric, just as variations of image metrics thrived in ITK.

The diagram in Figure 2 presents the hierarchy of mesh interpolators that are included in this contribution.

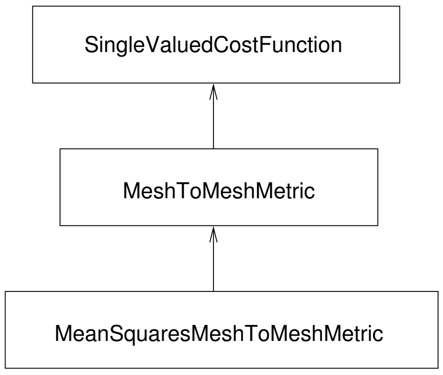

Figure 2: Hierarchy of new Mesh Metric Classes.

2.3 Mesh Point Locator

One of the challenging aspects of implementing an interpolator in Meshes is the fact that given a point in space, it is not trivial to find the Mesh cell that contains this point in its domain. The method used here involves the use of a `PointLocator` class that internally constructs a Kd-Tree in order to provide a spatial classification of the Mesh points.

When the registration method is mapping points from the Fixed Mesh into the Moving Mesh, this points are first mapped through the Transform and then they are passed to the interpolator's `Evaluate()` method. The first action of this method is to locate the points from the Moving Mesh that are closest to the point given as argument of Evaluate.

The `PointLocator` class takes advantage of the classes

- `itk::KdTreeGenerator`
- `itk::PointSetToListSampleAdaptor`

that are already provided by the ITK Statistics framework. The KdTree is constructed from the Moving Mesh during the initialization process of the interpolator, that is triggered by the Initialize() method of the Metric itself.

2.4 Triangle Basis System

The `TriangleBasisSystem` is an auxiliary class that holds a vector basis for a triangular cell of a Mesh. This vector basis is then used as a coordinate system in order to compute barycentric coordinates of points in a given cell. These barycentric coordinates are commonly used as weights for a linear interpolation of values assigned to the nodes of the triangular cell.

In a Mesh of topological dimension M that is embedded in a N-Dimensional space, this class will contains M vector of N components each. This class is essentially a data container and it does not provide computational capabilities.

2.5 Triangle Basis System Calculator

The `TriangleBasisSystemCalculator` is an auxiliary class that computes the TriangleBasisSystem of a cell from an M-Dimensional Mesh embedded in a N-Dimensional space. Given a set of N points, the calculator will compute the vectors of the basis.

2.6 Triangle List Basis System Calculator

The `TriangleListBasisSystemCalculator` is an auxiliary class that will compute the triangle basis systems for all the cells of an input mesh.

Given that in the context of Mesh to Mesh registration, the triangle basis of all the cells have to be computed eventually it results more efficient to pre-compute these basis at the beginning of the process, instead of having them computed on-demand as the points are sampled.

Note however that this is a design decision based on the assumption that users will deal most of the time with meshes that are not very large, and for which computation time is the main concern. If you are working with very large meshes, then it may be desirable to modify this code to avoid the memory allocation required for storing all the vector basis, and instead pay for the computation time of these vector set at every call of the interpolator `Evaluate()` method.

2.7 FreeSurfer Binary Surface Reader

FreeSurfer is and application developed by the Athinoula A. Martinos Center for Biomedical Imaging. Developement was supported in part by the National Center for Research Resources (P41-RR14075, R01 RR16594-01A1 and the NCRR BIRN Morphometric Project BIRN002, U24 RR021382), the National Institute for Neurological Disorders and Stroke (R01 NS052585-01), the National Institute of Biomedical Imaging and Bioengineering, as well as the Mental Illness and Neuroscience Discovery (MIND) Institute and is part of the National Alliance for Medical Image Computing (NA-MIC) funded by the National Institutes of Health through the NIH Roadmap for Medical Research, Grant U54 EB005149.

Although FreeSurfer binaries can be freely downloaded, the application is unfortunately not Open Source, and it uses is restricted to non-commercial applications.

For details, see http://surfer.nmr.mgh.harvard.edu/.

The application for which these Mesh registration classes were developed was the registration of brain surfaces that have been mapped to spheres. Surface data of this type is available as part of the binary download of Freesurface.

In order to facilitate the use of data generated from FreeSurfer as input to the Mesh to Mesh registration framework we provide here a reader class that read FreeSurfer binary files and convert them into itkQuadEdgeMeshes.

The minimalist use of the FreeSurferBinarySurfaceReader class is illustrated by the source code below.

```

#include "itkMesh.h"
#include "itkFreeSurferBinarySurfaceReader.h"

typedef itk::Mesh<float, 3>                  MeshType;
typedef itk::FreeSurferBinarySurfaceReader< MeshType >  ReaderType;
```

```

ReaderType::Pointer  surfaceReader = ReaderType::New();

surfaceReader->SetFileName("lh.pial");
surfaceReader->SetDataFileName("lh.curv.pial");

surfaceReader->Update();

MeshType::ConstPointer mesh = surfaceReader->GetOutput();
```

Where

- "lh.pial" is the file containing the geometry and topology of the mesh
- "lh.curv.pial" is the file containing the scalar field associated with points of the mesh

In the `Examples` directory of the accompanying source code you will find an example called

`ConvertFreeSurferBinarySurfaceToVTKSurface.cxx`

That, as the name indicates, can read a pair of FreeSurface files and read them into an itkQuadEdgeMesh, to then save the surface as a VTK legacy file.

3 How to Perform Registration

This section illustrates how to put together a Mesh to Mesh registration example. The code presented in this section is available in the subdirectory Examples, in the file

`MeanSquaresMeshToMeshMetricRegistration1.cxx`

3.1 Including Headers

We start by including the headers of the main components of the registration framework

- Metric
- Transform
- RegistrationMethod
- Interpolator
- Mesh

```
#include "itkMeanSquaresMeshToMeshMetric.h"
#include "itkMeshToMeshRegistrationMethod.h"
#include "itkLinearInterpolateMeshFunction.h"
#include "itkVersorTransformOptimizer.h"
#include "itkVersorTransform.h"
#include "itkQuadEdgeMesh.h"
```

Then we include headers for auxiliary classes. In particular,

- The Command class that will be used to observe the progress of the registration process via events.
- The Mesh reader that will load the meshes from files.

```
#include "itkCommand.h"
#include "itkVTKPolyDataReader.h"
```

3.2 Defining a Command Observer

The command observer class is defined in a very similar way to what you may have already seen in the image registration framework. The code would be almost identical to what is shown in the ITK Software Guide [**?**]

```
class CommandIterationUpdate : public itk::Command
{
public:
  typedef  CommandIterationUpdate   Self;
  typedef  itk::Command             Superclass;
  typedef itk::SmartPointer<Self>   Pointer;

  itkNewMacro( Self );

protected:
  CommandIterationUpdate()
   {
   iterationCounter = 0;
   }

public:
  typedef itk::VersorTransformOptimizer  OptimizerType;
  typedef   const OptimizerType   *      OptimizerPointer;

  void Execute(itk::Object *caller, const itk::EventObject & event)
    {
    Execute( (const itk::Object *)caller, event);
    }

  void Execute(const itk::Object * object, const itk::EventObject & event)
    {
    OptimizerPointer optimizer = dynamic_cast< OptimizerPointer >( object );

    if( ! itk::IterationEvent().CheckEvent( &event ) )
      {
      return;
      }

    std::cout << " Iteration " << ++iterationCounter;
    std::cout << "  Value " << optimizer->GetValue() << "   ";
    std::cout << "  Position " << optimizer->GetCurrentPosition() << std::endl ;
    }
private:
  unsigned int iterationCounter;

};
```

3.3 Declaring the Mesh Types

The fundamental type used in the registration process is the Mesh. In this case we chose to use the `QuadEdgeMesh` because we are dealing with 2D manifolds embedded in a 3D space.

```
  typedef itk::QuadEdgeMesh< float, 3 >   FixedMeshType;
  typedef itk::QuadEdgeMesh< float, 3 >   MovingMeshType;
```

We are using here two Meshes of the same type, but in principle, we could have used two different mesh types, as long as their space dimension is the same.

3.4 Reading the Meshes from Files

Using the Mesh types as template parameters, we can instantiate the types of the mesh readers and proceed to load the meshes into memory. Note that this is optional, since you could be inserting the registration code into an application in which the Meshes are generated through means different from file reading.

```
  typedef itk::VTKPolyDataReader< FixedMeshType >     FixedReaderType;
  typedef itk::VTKPolyDataReader< MovingMeshType >    MovingReaderType;

  FixedReaderType::Pointer fixedMeshReader = FixedReaderType::New();
  fixedMeshReader->SetFileName( argv[1] );

  MovingReaderType::Pointer movingMeshReader = MovingReaderType::New();
  movingMeshReader->SetFileName( argv[2] );

  try
    {
    fixedMeshReader->Update( );
    movingMeshReader->Update( );
    }
  catch( itk::ExceptionObject & exp )
    {
    std::cerr << exp << std::endl;
    return EXIT_FAILURE;
    }
```

Note that, as usual, the `Update()` methods are called inside a `try/catch` block in order to deal with potential Exceptions that may result from problems in the process of reading the Mesh files.

After updating the readers we can extract the fixed and moving meshes and assign them to temporary variables.

```
  FixedMeshType::ConstPointer  meshFixed  = fixedMeshReader->GetOutput();
  MovingMeshType::ConstPointer meshMoving = movingMeshReader->GetOutput();
```

3.5 Declaring the Registration Method

We instantiate the registration method by using the Mesh types as template parameters

```
typedef itk::MeshToMeshRegistrationMethod<
                                  FixedMeshType,
                                  MovingMeshType >    RegistrationType;

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

3.6 Declaring the Metric

We instantiate the metric type by using the Mesh types as template parameters. The we construct one metric and connect it to the registration method.

```
typedef itk::MeanSquaresMeshToMeshMetric< FixedMeshType,
                                          MovingMeshType >
                                          MetricType;

MetricType::Pointer  metric = MetricType::New();

registration->SetMetric( metric );
```

3.7 Connecting the input Meshes

The Fixed and Moving input meshes are connected to the registration method. Note that the order of this call is irrelevant, as long as it is done before calling the `StartRegistration()` method.

```
registration->SetFixedMesh( meshFixed );
registration->SetMovingMesh( meshMoving );
```

3.8 Declaring the Transform

Here we use the `itk::VersorTransform` because we only expect to apply rotations to the spherical meshes. The template parameter of the Transform is the type that will be used internally for performing computations. We get this type as a trait from the Metric type, which in turn defined this trait based on the NumericTraits of the Mesh's coordinate representation type.

```
typedef itk::VersorTransform< MetricType::TransformComputationType >   TransformType;

TransformType::Pointer transform = TransformType::New();

registration->SetTransform( transform );
```

3.9 Declaring the Interpolator

As we map points from the Fixed mesh into the Moving mesh, most of these points will land in the middle of the moving mesh cells. For this reason, we need an interpolator in order to compute the values of the Moving mesh scalar field and its derivatives at those locations. Here we select to use the linear interpolator for meshes.

```
typedef itk::LinearInterpolateMeshFunction< MovingMeshType > InterpolatorType;

InterpolatorType::Pointer interpolator = InterpolatorType::New();

registration->SetInterpolator( interpolator );
```

3.10 Initializing Transform Parameters

In some registration processes, the user may have an idea of what could be a good initial Transform. It is very important to convey this information to the registration framework, because it sets the optimizer in a much better location for exploring the cost function (Metric). In this particular case we take the initial values of the Transform from the command line parameters and use them to set the components of the rotation axis and the rotation angle. Then we get the parameters of the transform and pass them to the registration method.

```
TransformType::AxisType  axis;
TransformType::AngleType angle;

axis[0] = atof( argv[3] );
axis[1] = atof( argv[4] );
axis[2] = atof( argv[5] );

angle = atof( argv[6] );

transform->SetRotation( axis, angle );

parameters = transform->GetParameters();

registration->SetInitialTransformParameters( parameters );
```

3.11 Setting the Optimizer

Declaration

The VersorTransform parameters do not form a Vector space due to the fact that the addition operation is not equivalent to composition of rotation in the Versor space. For this reason we have to use here an Optimizer that is aware of this particularity of the space of rotations. Our choice is the `itk::VersorTransformOptimizer`. Here we instantiate the optimizer and connect it to the registration method.

```
typedef itk::VersorTransformOptimizer     OptimizerType;

```

```
OptimizerType::Pointer      optimizer     = OptimizerType::New();

registration->SetOptimizer( optimizer );
```

Transform Parameter Scaling

The parametric space in which the Optimizer explores the cost function (the Metric) is not necessarily isotropic. Most ITK optimizers provide a mechanism for users to specify the different scales of the particular dimensions of the parametric space. In this specific case, however, the three components of the Versor are of similar scale, and therefore we can set all the scaling parameter values to the same number.

```
typedef OptimizerType::ScalesType              ScalesType;

ScalesType    parametersScale( numberOfTransformParameters );
parametersScale[0] = 1.0;
parametersScale[1] = 1.0;
parametersScale[2] = 1.0;

optimizer->SetScales( parametersScale );
```

Optimizer Parameters Fine Tunning

One of the most critical aspects of performing registration is to identify the values for the parameters of the optimizer. These parameters are the ones controlling the behavior of the optimizer as it explores the parametric space of the Transform. Inappropriate values of this parameters will lead to the optimizer getting trapped in local minima, or running for too many iterations, or not enough iterations.

Unfortunately, the process of selecting Optimizer parameters is not an exact science and requires a systematic process of trial and error. This is where most of the effort of registration is spent by users.

The following parameters turn out to be appropriate for registering the FixedMesh.vtk in the data directory with itself.

```
optimizer->MinimizeOn();
optimizer->SetGradientMagnitudeTolerance( 1e-6 );
optimizer->SetMaximumStepLength( 0.05 );
optimizer->SetMinimumStepLength( 1e-9 );
optimizer->SetRelaxationFactor( 0.9 );
optimizer->SetNumberOfIterations( 100 );
```

Connecting the Observer to the Optimizer

The Command observer that we defined at the beginning of this example, is now connected to the Optimizer. This will report the value of the Metric along with the parameters of the Transform at every iteration of the optimization.

```
CommandIterationUpdate::Pointer observer = CommandIterationUpdate::New();
optimizer->AddObserver( itk::IterationEvent(), observer );
```

3.12 Running the Registration

We are finally ready for running the registration. This is done by calling the `StartRegistration` method. We place this call inside a `try/catch` block in order to be able to manage any potential Exception that may be thrown as a result of an internal error during the registration process.

```
try
  {
  registration->StartRegistration();
  }
catch( itk::ExceptionObject & e )
  {
  std::cerr << "Registration failed" << std::endl;
  std::cout << "Reason " << e << std::endl;
  return EXIT_FAILURE;
  }
```

The command line used for running the registration is

```
MeanSquaresMeshToMeshMetricRegistration1  FixedMesh.vtk MovingMesh.vtk 0.0 0.0 1.0 0.1
```

Where the first three numbers after the MovingMesh.vtk parameter are the components of the rotation axis, and the last numeric parameter is the initial angle of rotation expressed in radians.

3.13 Print out the Registration Results

Naturally, after running the registration, we want to look at the final results. This is done by calling the `GetLastTransformParameters()` method as illustrated in the lines below.

```
OptimizerType::ParametersType finalParameters =
                    registration->GetLastTransformParameters();

const double bestValue = optimizer->GetValue();

std::cout << "final parameters = " << finalParameters << std::endl;
std::cout << "final value      = " << bestValue << std::endl;
```

4 Visually Monitoring the Registration Process

Following the behavior of the registration process is another one of the challenging aspects of performing registration. Often times it is frustrating for users to spend long hours running a registration process without having proper feedback on whether the optimizer is heading in the right direction in the parametric space.

This section illustrates how to use a helper class that provides visual feedback as the iteration process is ongoing.

The code can be found in the Examples directory, in the file

`MeanSquaresMeshToMeshMetricRegistrationWithMonitor1.cxx`

This file is a slightly modified version of the example that we described in the previous section.

4.1 Including the Registration Monitor

The first difference between this example and the previous one is that here we replace the Command observer with a class called `RegistrationMonitor`. This new class provides visual feedback by using VTK classes to display a rendering of the surface as it is mapped by the Transform at every iteration of the registration process.

The RegistrationMonitor class requires vtkPolyData types as inputs, therefore we redundantly load the meshes into vtkPolyData structures by using the vtkPolyDataReader class.

We need to include the following headers

```
#include "RegistrationMonitor.h"
#include "vtkPolyDataReader.h"
#include "vtkPolyData.h"
#include "vtkSmartPointer.h"
```

4.2 Loading Surfaces into PolyData

In order to load the Fixed and Moving meshes into vtkPolyData classes we use two instances of the vtkPolyDataReader class. This is, unfortunately, a duplication of memory from the Meshes that we already read into itkMesh classes.

```
  vtkSmartPointer< vtkPolyDataReader > vtkFixedMeshReader =
    vtkSmartPointer< vtkPolyDataReader >::New();

  vtkSmartPointer< vtkPolyDataReader > vtkMovingMeshReader =
    vtkSmartPointer< vtkPolyDataReader >::New();

  vtkFixedMeshReader->SetFileName( argv[1] );
  vtkMovingMeshReader->SetFileName( argv[2] );

  vtkFixedMeshReader->Update();
  vtkMovingMeshReader->Update();
```

4.3 Instantiating the Registration Monitor

We proceed now to instantiate the RegistrationMonitor, connect the surfaces to it, and set the parameters that define how often the visualization is going to be updated.

```
  RegistrationMonitor visualMonitor;

  visualMonitor.SetFixedSurface( vtkFixedMeshReader->GetOutput() );
  visualMonitor.SetMovingSurface( vtkMovingMeshReader->GetOutput() );
```

```

visualMonitor.SetNumberOfIterationsPerUpdate( 1 );

visualMonitor.Observe( optimizer.GetPointer(), transform.GetPointer() );
```

The most interesting line is the call to the `SetNumberOfIterationsPerUpdate` method. This method defines how many iterations of the optimizer should pass between refreshes of the surface rendering display. For the purpose of this example we render the surface at every iteration, hence the value "1". However, this has the disadvantage that we are probably spending more CPU cycles in performing rendering and visualization than in actually computing the registration.

In a more realistic scenario, we probably should use a value of 10 or 50, to let the optimizer make more progress in the registration before we spend time in updating the surface rendering.

The use of the RegistrationMonitor is mostly an scaffolding intended to assist the user as it is figuring out proper values for all the numeric parameters used in the registration process. Once you have identified good values for all these parameters you probably want to remove the RegistrationMonitor from the code, given that at that point it is not contributing much to bringing the registration to completion.