Information-Theoretic Directly Manipulated Free-Form Deformation Labeled Point-Set Registration

Nicholas J. Tustison, Suyash P. Awate and James C. Gee

February 3, 2009

Penn Image Computing and Science Laboratory, University of Pennsylvania, USA

Abstract

Our previous contributions to the ITK community include a generalized B-spline approximation scheme [3] as well as a generalized information-theoretic measure for assessing point-set correspondence known as the Jensen-Havrda-Charvat-Tsallis (JHCT) divergence [6]. In this submission, we combine these two contributions for the registration of labeled point-sets. The transformation model which uses the former contribution is denoted as *directly manipulated free-form deformation* (DMFFD) and has been used for image registration [5]. The information-theoretic approach described not only eliminates exact cardinality constraints which plague exact landmark matching algorithms, but it also incorporates the local point-set structure into the similarity measure calculation. Although theoretical discussion of these two components is deferred to other venues, the implementation details given in this submission should be adequate for those wishing to use our algorithm. Visualization of results is aided by another of our previous contributions [4]. Additionally, we provide the rudimentary command line parsing classes used in our testing routines which were written in the ITK style and also available to use consistent with the open-source paradigm.

Contents

1 Introduction

Various algorithms have been developed for the registration of various geometric primitives, e.g. surfaces, curves, and points. One such algorithm included in the ITK library is the well-known iterative closest point (ICP) algorithm of Besl and McKay [2]. Since points are perhaps the simplest of all geometric primitives, the utility of a good point-set registration algorithm is significant.

There are two theoretical contributions that we make in this work which will be only mentioned briefly as the theoretical details are, or hopefully will be, delineated elsewhere. In [5], we explain that traditional gradient-based FFD image registration schemes are intrinsically susceptible to problematic energy topographies. In the same publication we derive a preconditioned gradient which provides a significant improvement over previous algorithms. We denote this new FFD registration scheme as *directly manipulated free-form deformation* (DMFFD) image registration. Since these susceptibilities are general they also apply to point-set registration schemes for which the DMFFD remedy is also applicable.

In addition to defining a novel transformation model, we employ a generalized information-theoretic point-set measure known as the Jensen-Havrda-Charvat-Tsallis (JHCT) divergence [6]. This measure allows for controlling the emphasis between robustness and sensitivity with a single tunable parameter. Also considered in the assessment of point-set correspondence is the local structure of the point-sets themselves. Theoretical details as well as the results from extensive testing on data from medical imagery will (hopefully) be forthcoming.

2 Command Line Parser

Antecedent to discussion of the point-set algorithm itself we give a brief discussion of our ITK command line parsing classes as we use them in our testing. In addition to our testing routine, they were originally developed for and are currently being used in an open-source image registration suite known as Advanced Normalization Tools (ANTs)[1] which is also ITK-based.[1]

The two relevant command line parsing classes are:

- `CommandLineOption`
- `CommandLineParser`

The first class contains the necessary functionality for an individual command line option. An option can have 0 or more values with each value holding 0 or more parameters. Each option also has a short name

[1]N.B. We are not encouraging people to use our command line parsing routines (although they are certainly welcome to do so). However, It is merely our preference for the discussed style of command line parsing, as opposed to the routines in getopt.h already available, that we designed our own. Those who prefer the getopt.h routines are perfectly welcome to wrap the point-set registration classes in such a framework and it should be straightforward to do so. That is one of the benefits of ITK.

of type `char` (invoked by ‘-’), a long name of type `std::string` (invoked by ‘–’), and a description of type `std::string`.

For example, suppose we were creating an image registration program which has several transformation model options such as ‘rigid’, ‘affine’, and ‘deformable’. An instance of the command line option could have a long name of “transformation”, a short name of ‘t’, and the description “Transformation model—rigid, affine, or deformable”. The values for this option would be “rigid”, “affine”, and “deformable”. Each value would then hold parameters that correspond to that value. For example, a possible subsection of the command line for calling this fictional image registration program could be

```
--transformation rigid[parameter1]
```

or

```
-t affine[parameter1,parameter2,parameter3]
```

The parsing class, `CommandLineParser`, takes as input the standard `argc`, `argv` variables, parses the input, and stores them in a data structure of options. This class also contains routines for converting types including `std::vector` using ‘x’ as a delimiter. For example, I can specify the 3-element `std::vector` {10, 20, 30} on my command line as “10x20x30”.

Setting up possible options can perhaps best be understood by viewing the calls in our testing routine `itkDMFFDLabeledPointSetRegistrationFilterTest.cxx`. After instantiating the command line parser, we set the command and command descriptions (for documentation purposes)

```
itk::CommandLineParser::Pointer parser = itk::CommandLineParser::New();
parser->SetCommand( argv[0] );
parser->SetCommandDescription( "DMFFD Labeled Point-Set Registration" );
```

and initialize the command line options. For example, setting up the “optimization” option is carried out as follows:

```
std::string description =
  std::string( "[maximumNumberOfIterationsAtEachLevel," ) +
  std::string( "<gradientScalingFactor(s)>,<lineSearchIterations>," ) +
  std::string( "<lineSearchMaximumStepSize>]" );

OptionType::Pointer option = OptionType::New();
option->SetLongName( "optimization" );
option->SetShortName( 'z' );
option->SetDescription( description );
parser->AddOption( option );
```

Once the parser and command line options are initialized, the user can then extract the values and parameters from the parser itself. Again, we return to our “optimization” example. After the command line is read, the code snippet for reading the “optimization” values and parameters is given in lines 394-449.

```
394  /**
395   * Set optimization variables
396   */
397  if( verbose )
398    {
399    std::cout << "Setting optimization variables." << std::endl;
400    }
401
402  typename itk::CommandLineParser::OptionType::Pointer optimizationOption =
403      parser->GetOption( "optimization" );
404    if( !optimizationOption || optimizationOption->GetNumberOfParameters() < 1 )
405      {
406      std::cerr << "Incorrect optimization option specification." << std::endl;
407      std::cerr << "   " << optimizationOption->GetDescription() << std::endl;
408      return EXIT_FAILURE;
409      }
410    std::vector<unsigned int> numIterations = parser->template
411      ConvertVector<unsigned int>( optimizationOption->GetParameter( 0 ) );
412    typename RegistrationFilterType::ResizableUIntArrayType numberOfIterations;
413    numberOfIterations.SetSize( numIterations.size() );
414    for( unsigned int i = 0; i < numIterations.size(); i++ )
415      {
416      numberOfIterations[i] = numIterations[i];
417      }
418    registrationFilter->SetMaximumNumberOfIterations( numberOfIterations );
419    if( optimizationOption->GetNumberOfParameters() > 1 )
420      {
421      std::vector<typename RegistrationFilterType::RealType> gradFactors =
422        parser->template ConvertVector<typename RegistrationFilterType::RealType>(
423        optimizationOption->GetParameter( 1 ) );
424      typename RegistrationFilterType::ResizableRealArrayType
425        gradientScalingFactors;
426      gradientScalingFactors.SetSize( numIterations.size() );
427      if( gradFactors.size() != numIterations.size() )
428        {
429        gradientScalingFactors.Fill( gradFactors[0] );
430        }
431      else
432        {
433        for( unsigned int i = 0; i < gradFactors.size(); i++ )
434          {
435          gradientScalingFactors[i] = gradFactors[i];
436          }
437        }
438      registrationFilter->SetGradientScalingFactor( gradientScalingFactors );
439      }
440    if( optimizationOption->GetNumberOfParameters() > 2 )
441      {
442      registrationFilter->SetLineSearchMaximumIterations( parser->template
443        Convert<unsigned int>( optimizationOption->GetParameter( 2 ) ) );
444      }
445    if( optimizationOption->GetNumberOfParameters() > 3 )
446      {
447      registrationFilter->SetLineSearchMaximumStepSize( parser->template
448        Convert<float>( optimizationOption->GetParameter( 3 ) ) );
449      }
```

As seen on lines 402-403, we obtain the “optimization” option directly from the parser which has been assigned values and parameters based on what was actually written on the command line. Note that we could have obtained the option using the short name, i.e. `optimizationOption = paser->GetOption( ’z’ )`. We also include functions to convert types including vector types delimited by the character ‘x’. For example, in our optimization framework, we allow for multiple B-spline resolution levels. This means that I can start out with a relatively coarse B-spline grid and increase the resolution during the registration. Suppose I wanted 20 iterations at the first level, 10 iterations at the second level, and 5 iterations at the third and final level. The way that this would be specified on the command line would be definecolorlistbackgroundgray1.0 `--optimization [30x20x10]` The vector corresponding to this parameter could be obtained by using the `ConvertVector` function as seen on lines 410-418. Similarly, we have a `Convert` function for non-vector types as shown on lines 442-443.

3 Point-Set Registration

In continuation with previous discussion, we first describe a typical command line call taken from one of the tests contained in the accompanying `CMakeLists.txt` file. We then give an accounting of the classes included in this contribution.

3.1 Command Line Call for the Testing Routine

A sample command line call from our set of tests is the following:

```
itkDMFFDLabeledPointSetRegistrationFilterTest 3 --point-sets [${CMAKE_SOURCE_DIR}/Data/sphere.vtk,${
CMAKE_SOURCE_DIR}/Data/square.vtk] --transformation [3,1x1x1,${CMAKE_SOURCE_DIR}/Data/square.nii.gz] --labels
[1,0.5,1] --similarity [1.5,0.97,10,0,50,1,5,10] --optimization [20x10x5,4] --output ${CMAKE_BINARY_DIR}/psr_2 --
verbose
```

The first two arguments are the executable and the point-set dimension. The subsequent arguments are as follows (optional parameters are enclosed by <>):

- `--point-sets [fixedPointSet,movingPointSet,<numberOfFixedSamples>,<numberOfMovingSamples>]`
  - `fixedPointSet`: file containing the designated fixed point set. This file can be in the form of a label image (e.g. ITK-snap segmentation image), a vtk file with scalar labels, or one can read in a simple text file (the filename suffix must be '.txt') with the following format:
    $$\begin{array}{cccc} 0 & 0 & 0 & 0 \\ x_1 & y_1 & z_1 & label_1 \\ x_2 & y_2 & z_2 & label_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & z_n & label_n \\ 0 & 0 & 0 & 0 \end{array}$$
    For 2-D point-sets, the z-value is simply ignored. Specified as a file name.
  - `movingPointSet`: file containing the designated moving point set. Same file format possibilities as for the fixed point-set. Specified as a file name.
  - `<numberOfFixedSamples>`: number of samples to be generated from the fixed point set. If not specified the samples are the fixed points themselves. Specified as a scalar.
  - `<numberOfMovingSamples>`: number of samples to be generated from the moving point set. If not specified the samples are the moving points themselves. Specified as a scalar.
- `--transformation [splineOrder,meshResolution,<domainImage>]`
  - `splineOrder`: designates the order of the B-spline basis functions. Default is cubic (3) but projective and piecewise projective are possible with order = 1, quadratic with order = 2. Higher orders > 3 are also possible. Specified as a scalar.
  - `meshResolution`: specifies the B-spline mesh resolution at the coarsest resolution level which doubles for each resolution level. A 2-D 3×5 mesh resolution would be specified as “3x5” whereas a 3-D $3 \times 4 \times 5$ mesh resolution would be specified as “3x4x5”. The resolution at the next higher level would be “6×8” (2-D) and “$6 \times 8 \times 10$” (3-D). Specified as a vector.

- `<domainImage>`: B-spline objects have finite domain support which must be specified. Common practice for us is specifying this domain via the size, origin, and spacing information of an image. **Note that the image directional information is not used.** If the domain image is not specified, the bounding box is calculated from both point sets and used as the finite domain. If the image domain is not specified, we arbitrarily chose to divide the resulting bounding box domain into 100 elements in each dimension and the spacing was calculated accordingly. Please note that this division has no effect on the course of the registration as only the boundary of the domain matters. However, it will affect the output deformation field as it is calculated based on this domain. Specified as a file name.

- `--optimization [maximumNumberOfIterationsAtEachLevel,<gradientScalingFactor(s)>,<lineSearchIterations>,<lineSearchMaximumStepSize>]`
  - `maximumNumberOfIterations`: Specifies the maximum number of conjugate gradient descent iterations at each resolution level. This is designated in vector form, e.g. “10x5x3x1” designates four resolution levels with the coarsest resolution level having a maximum of 10 iterations, the next level having a maximum of 5 iterations followed by levels with respective maximums of 3 and 1 iterations. Specified as a vector for multiple levels or a scalar for a single level.
  - `<gradientScalingFactor(s)>`: The gradient step taken is based on the voxel spacing of the domain. The gradient scaling factor(s) allow one to multiply the gradient by a single scalar factor for all levels of the entire registration or one can specify a set of scalar factors corresponding to the number of levels, e.g. $2 \times 1 \times 0.5 \times 0.5$ specifies a gradient scaling factor of 2 at the base level, 1 at the second level, and 0.5 for each of the top two levels. Specified as a vector or a scalar.
  - `<numberOfLineSearchIterations>`: number of line search iterations for all resolution levels. Specified as a scalar.
  - `<lineSearchMaximumStepSize>`: This scalar acts as a governor for the step size taken during the line search (based on the voxel spacing and gradient scaling factor at the current level). Specified as a scalar.

- `--labels [<whichLabels>,<labelPercentages>,<labelGradientWeights>]`
  - `<whichLabels>`: Suppose we had fixed and moving point-sets with labels ‘1’, ’5’, ’7’, ’3’, and ’12’ but we only want to perform the registration with labels ’1’ and ’12’ and ’3’. This parameter allows one to do that by specifying this parameter as “1x12x3”. Specified as a vector for multiple labels or a scalar for a single label.
  - `<labelPercentages>`: Given the previous example about registering points only with labels “1x12x3”, suppose we only want to use 10% of the points with label ‘1’, 35% of the points with label ‘12’ and 100% of the points with label ‘3’, we can do this by designating this parameter as “0.1x0.35x1.0”. Specified as a vector for multiple labels or a scalar for a single label.
  - `<labelGradientWeights>`: Given the previous example about registering points only with labels “1x12x3”, suppose we want to weight the gradient of points with label ‘12’ twice as much as the other two labels. We can do this by specifying this parameter as “0.5x1.0x0.5” or “1x2x1” or “25.37x50.74x25.37” which should have equal effect. Specified as a vector for multiple labels or a scalar for a single label.

- `--similarity [alpha,annealingRate,pointSetSigma,<useRegularizationTerm>,<evaluationKNeighborhood>,<useAnisotropicCovariances>,<covarianceKNeighborhood>,<kernelSigma>]`: Each of these parameters are explained in greater detail in our previous submission [6]. Each parameter is specified as a scalar.

- `--output filePrefix`: prefix for the output files comprised of deformation field component images (.nii.gz) and warped point vtk file. Specified as a string.
- `--verbose`: If chosen, the course of the registration is printed to the screen.

3.2 Additional ITK Classes

In addition to the command line parsing classes discussed earlier, we provide the following classes:

- `BSplineControlPointImageFilter`: auxiliary class which produces a B-spline object from the control point grid produced as output from our earlier submission [3]. Also contains other useful utilities for B-spline control point grids.
- `DMFFDLabeledPointSetRegistrationFilter`: main class for labeled point set registration.
- `JensenHavrdaCharvatTsallisLabeledPointSetMetric`: wrapper class for our earlier submission [6] to accommodate labeled point-sets.
- `VectorImageFileReader/Writer`: IO classes for reading/writing vector component images.

We have also included the necessary JHCT classes from [6] as well as a grid producing class [4] for deformation visualization purposes and a utility which generates a deformed grid from a deformation field.

3.3 Miscellany

There are a couple of miscellaneous items of which we would like to inform the user. As we get feedback from the users, we expect this section to grow.

- We would recommend that you compile in `Release` mode. This is done in the traditional manner using the cmake menu.
- In specifying the fixed and moving points, the resulting output deformation field will be that sampled field (from a continuous B-spline object) which ‘pushes’ the moving points to the ‘fixed’ points. However, one might want to ‘pull’ one image into another image using point-sets derived from the image. This is easily done by switching the fixed and moving point-sets. The user then has a deformation field which, when applied by the `itk::WarpImageFilter`, can warp the image in the appropriate fashion. This is used in one of our examples.
- We found it easiest to output the deformation field as component images. For example, if one specifies the output prefix to be ‘psr’ the output will consist of two or three scalar images (dependent upon the dimensionality) with the names given as `psrWarpxvec.nii.gz`, `psrWarpyvec.nii.gz`, and possibly `psrWarpzvec.nii.gz`.
- The warped images used in the visualization of the Examples section are created using the `CreateWarpedGridImage` utility that we have included in this submission. It’s fairly easy to use and is derived from our earlier submission [6]. The call is as follows

```
Usage: CreateWarpedGridImage ImageDimension deformationField outputImage [directions,e.g. 1x0x0] [
gridSpacing] [gridSigma]
```

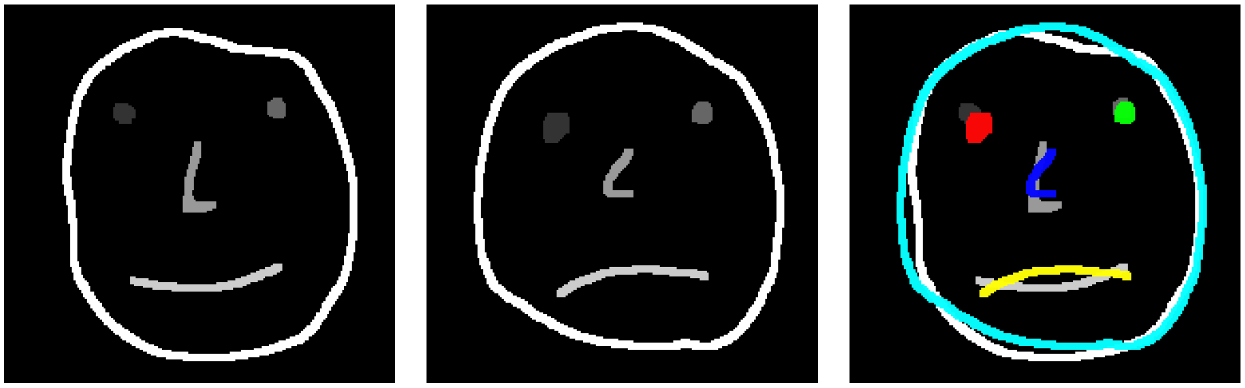

(a) (b) (c)

Figure 1:

So, given our earlier deformation field example, a sample call would be

```
Usage: CreateWarpedGridImage 3 psrWarp.nii.gz psrWarpedGrid.nii.gz
```

where we have omitted the components in our deformation field file name.

4 Examples

The following 2-D and 3-D examples are reproduced using the tests in `CMakeLists.txt`. The second one has been commented out due to timeout concerns during testing. However, the individual user should be able to run the tests by uncommenting the lines or reproducing the appropriate command line calls themselves. The examples should give the user a feel for the approximate parameter values used for various problems although it is by no means exhaustive.

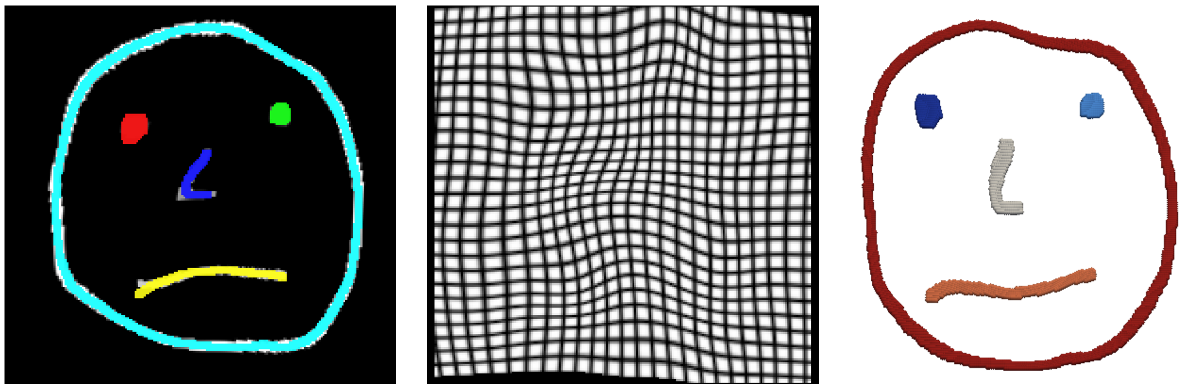
4.1 Smiley vs. Frowney

The following 2-D example (test `PSR_1` in `CMakeLists.txt`) uses label images of a smiley face (Figure 1(a)) and and a frowney face (Figure 1(b)) created using ITK-SNAP. Note that there are five labels corresponding to the sparse anatomical features in both faces. The smiley face is comprised of 3354 total points whereas the frowney face is comprised of 3459 points where each labeled voxel constitutes a point.

After designating the frowney face as the moving point-set and the smiley face as the fixed point-set, we ran the registration and obtained the results given in Figure 2. As we noted in our Miscellany section, we can warp the smiley face image into the frowney face image using the resulting deformation field which is shown in Figure 2(a). The corresponding warped grid image is given in Figure 2(b). We also show the warped frowney face points in Figure 2(c) which were 'pushed' into the space of the smiley face points.

4.2 Sphere to Cube

Our second example illustrates the registration capabilities for two 3-D point-sets. Given the sampled cube and sphere in Figure 3, we choose to register the cube to the sphere such that the cube comprises the 'moving' point-set and the the sphere comprises the 'fixed' point-set. In addition to displaying the registration



(a) (b) (c)

Figure 2:

result in Figure 4(a), we show the resulting warped grid image showing the inverse deformation which pulls the circle into the square.

References

[1] B. B. Avants, N. J. Tustison, G. Song, S. Das, J. Das, J. Pluta, and J. C. Gee. Ants: Advanced open source tools for normalization and neuroanatomy. Available at: http://www.picsl.upenn.edu/ANTS/. 2

[2] Paul J. Besl and Neil D. McKay. A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992. 1

[3] N. J. Tustison and J. C. Gee. N-d C^k B-spline scattered data approximation. *The Insight Journal*, 2005. (document), 3.2

[4] Nicholas J. Tustison, Brian A. Avants, and James C. Gee. Gridding graphic graticules. *Insight Journal*, 2007. (document), 3.2

[5] Nicholas J. Tustison, Brian B. Avants, and James C. Gee. Directly manipulated free-form deformation image registration. *IEEE Transactions on Image Processing*, 2009. accepted for publication. (document), 1

[6] Nicholas J. Tustison, Suyash P. Awate, and James C. Gee. A novel information-theoretic point-set measure based on the Jensen-Havrda-Charvat-Tsallis divergence. *Insight Journal*, 2008. (document), 1, 3.1, 3.2, 3.3

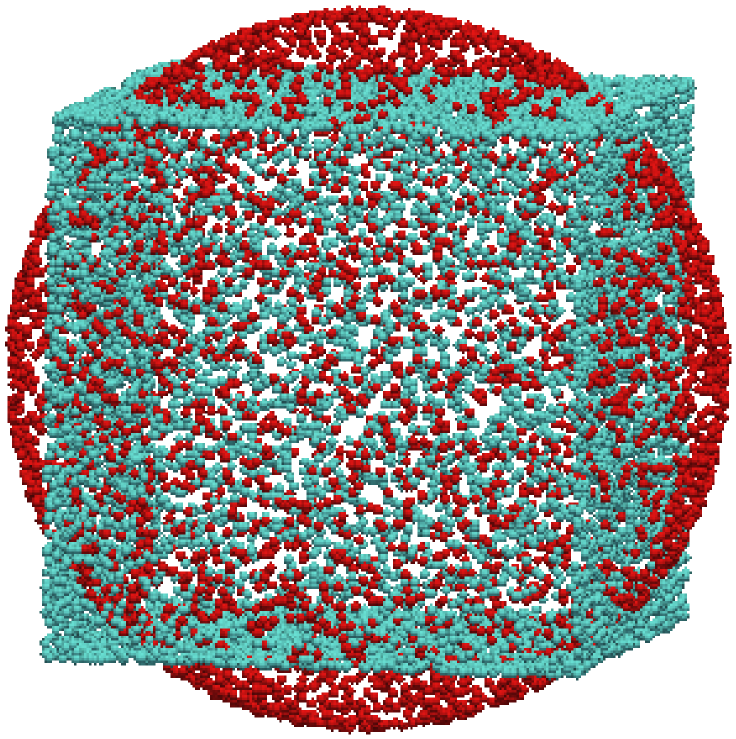
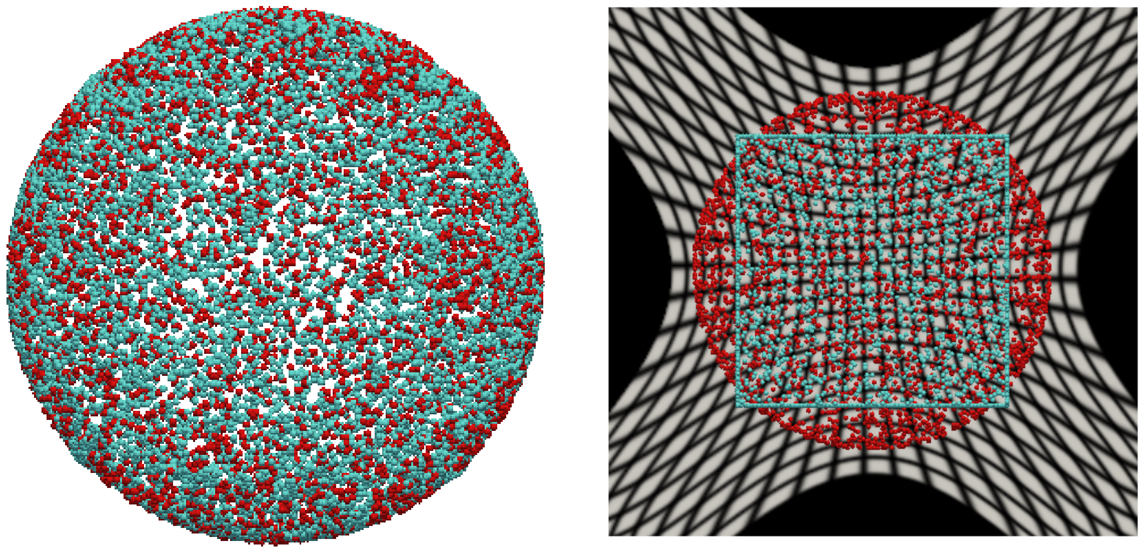


Figure 3: Cube (blue) and sphere (red) point-sets before registration.



(a) (b)

Figure 4: Registration results: (a) The cube (blue) and sphere (red) point-sets after registration. The warped grid image showing the inverse deformation (‘pulling’ the sphere to the square).