Registration Primer and RPI Description

Vincent Garcia

September 14, 2012

Contents

Chapter 1

Images

Contrary to the classical notion of image employed in image processing, a medical image corresponds to the acquisition of an 3D (anatomical) volume[1]. The volume is discretized by first defining a virtual grid in the volume to acquire. Each element of the grid, called voxel for *volume element*, receive a value computed from the integration of a given signal on an elementary volume centered on the considered voxel. The grid and its associated values are then stored into a 3D array.

Considering the 3D array to represent a volume has been norm the for long. However, over the years, research studies proved that adding geometrical and anatomical information allowed to greatly improve computations. For instance, the information relative to the voxel size – distance between two adjacent voxels – has been proved to be very useful if not essential. Recent acquisition devices provide more geometrical (or positioning) information, such as image origin and image orientation, that may really help any process on images (see Figure 1.1).
As a consequence, for each image, we have two different coordinate systems. First, the image coordinate system allow to access to the voxel value using classical indexes (*e.g.* $I(i, j, k)$). Second, the world coordinate system position each voxel according to a coordinate system shared by all images. These coordinate systems and the projection from one system to the other one are presented in section 1.1.

Most if discretization implies a lost of information. The notion of image interpolation, well know in the image processing community, tries to recover the information lost during this discretization process. We present briefly in section 1.2 some interpolation algorithms used in medical imaging.

1.1 Coordinate systems

Let I be a 3D image. If we consider I as an array in a 3-dimensional space, the element of the array in voxel coordinates can be accessed by $I(i, j, k)$ where i, j, and k are natural numbers including 0. Given the voxel coordinates (i, j, k), how to project these coordinates into the world coordinates, and conversely? In this section, we answer this simple but essential question.

[1]We consider here only 3D scalar images. Generalization to other dimensions and other modalities can be trivially deduced from the 3D case.

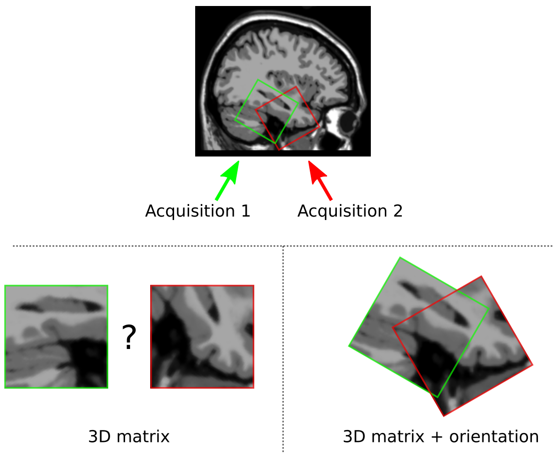

Figure 1.1: Acquisition of two images at different angles of incidence. Without position information (bottom left picture), the two acquisition cannot be superimposed and are difficult to compare. With the position information (bottom right picture), the two acquisition are trivially superimposed.

1.1.1 Projection of coordinates

Given a 3D image I, let A be a matrix 3×3 and T be a vector in the 3-dimensional space. A and T contain the necessary information to position the 3D array (grid) into the world coordinates. Let (i, j, k) be voxel coordinates, the corresponding position in the word coordinates is given by:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = A. \begin{bmatrix} i \\ j \\ k \end{bmatrix} + T \tag{1.1}$$

where

$$T = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{1.2}$$

and

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,1} \\ a_{2,1} & a_{2,2} & a_{2,1} \\ a_{3,1} & a_{3,2} & a_{3,1} \end{bmatrix}. \tag{1.3}$$

T contains the position of the image origin in world coordinates while A contains information on voxel size and image orientation. This projection from the voxel to the world coordinates usually uses a 4×4 matrix in homogeneous coordinates:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = M. \begin{bmatrix} i \\ j \\ k \\ 1 \end{bmatrix} \tag{1.4}$$

where

$$M = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,1} & t_x \\ a_{2,1} & a_{2,2} & a_{2,1} & t_y \\ a_{3,1} & a_{3,2} & a_{3,1} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.5}$$

Both projections (1.1) and (1.4) are identical and we will use in this report one or the other.

Let us now consider (x, y, z) a 3D position in the world coordinates. The projection of this point into the voxel coordinates (i, j, k) is given by:

$$\begin{bmatrix} i \\ j \\ k \end{bmatrix} = A^{-1}. \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} - T \right) \tag{1.6}$$

One should note that i, j, k value are not necessarily integer. If so, the value $I(i, j, k)$ is then obtained by image interpolation. This notion will be discussed on section 1.2. If we consider a 4×4 matrix in homogeneous coordinates, the projection from world to voxel coordinates is given by:

$$\begin{bmatrix} i \\ j \\ k \\ 1 \end{bmatrix} = M^{-1}. \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{1.7}$$

For clarity purpose, we may write in this report $I(x, y, z)$ (or simply $I(\mathbf{x})$ where $\mathbf{x} = (x, y, z)$) where (x, y, z) are world coordinates. The notation $I(x, y, z)$ is abusing and we define

$$I(x, y, z) \triangleq I(i, j, k) \tag{1.8}$$

where (i, j, k) is the projection of (x, y, z) into the voxel coordinates. The nature of the coordinates will indicate which notation is used.

1.1.2 Positioning information in `itk::Image` object

The ITK image format (`itk::Image`) contains the three following information to position any image into the world coordinated:

- position of image origin
- voxel size
- image orientation (called *direction cosine*)

These information are respectively retrieved using functions `GetOrigin()`, `GetSpacing()`, and `GetDirection()`, functions inherited from objet `itk::ImageBase`. Image origin and voxel size (respectively noted T and S) are stored as vector in a 3-dimensional space (`itk::Vector`):

$$T = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{1.9}$$

$$S = \begin{bmatrix} s_x \\ s_y \\ s_z \end{bmatrix} \tag{1.10}$$

Image orientation is stored as a 3×3 matrix (`itk::Matrix`) noted here D:

$$D = \begin{bmatrix} d_{1,1} & d_{1,2} & d_{1,1} \\ d_{2,1} & d_{2,2} & d_{2,1} \\ d_{3,1} & d_{3,2} & d_{3,1} \end{bmatrix} \tag{1.11}$$

To be consistent with notations introduction in Eq. (1.1), we define the 3×3 matrix A as follows:

$$A = D.S_{diag} \tag{1.12}$$

where

$$S_{diag} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}. \tag{1.13}$$

Vector T and matrix A allow to project voxel to world coordinates, and conversely (see Eq. (1.1)).

1.1.3 Positioning information in Inrimages image format

The Inrimage format contains all the information needed to position any image into the world coordinated:

- TX, TY, and TZ is the position of the image origin (resp. along the X-, Y-, and Z-axes),
- VX, VY, and VZ give the voxel size,
- RX, RY, and RZ correspond to the image orientation.

As in section 1.1.2, we define the vector T and S as follows:

$$T = \begin{bmatrix} TX \\ TY \\ TZ \end{bmatrix} \tag{1.14}$$

$$S = \begin{bmatrix} VX \\ VY \\ VZ \end{bmatrix} \tag{1.15}$$

The image orientation (noted D in section 1.1.2) is computed from values RX, RY et RZ. First, we define the vector R as follows:

$$R = \begin{bmatrix} RX \\ RY \\ RZ \end{bmatrix}. \tag{1.16}$$

Then, let N be the normalized vector

$$N = \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} = \begin{bmatrix} RX/\phi \\ RY/\phi \\ RZ/\phi \end{bmatrix} \tag{1.17}$$

where ϕ is the Euclidean norm of R. Finally, the orientation matrix D is given by:

$$D = cos\phi \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + (1 - cos\phi) \begin{bmatrix} n_x^2 & n_x n_y & n_x n_z \\ n_x n_y & n_y^2 & n_y n_z \\ n_x n_z & n_x n_y & n_z^2 \end{bmatrix} + sin\phi \begin{bmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{bmatrix} \tag{1.18}$$

As in section 1.1.2, the 3×3 matrix A defined in Eq. (1.1) an necessary to project voxel coordinates into the world coordinates is given by:

$$A = D.S_{diag} \tag{1.19}$$

1.2 Image interpolation

Let I be an image of size $[w, h, d]$ (resp. width, height, and depth). As explained in section 1.1, the image is the representation (discretization) of a volume in the world coordinate system. Let $\mathbf{x} = (x, y, z)$ be the coordinates of a point in the world coordinate such that $\mathbf{x}$ is included into the acquired volume. In other words, if $\mathbf{i} = (i, j, k)$ is the projection of $\mathbf{x}$ in the voxel coordinates (see section 1.1.1), we have:

$$\mathbf{i} \in [0, w-1] \times [0, h-1] \times [0, d-1] \tag{1.20}$$

If $\mathbf{x}$ is randomly drawn in the image volume, the coordinates (i, j, k) are probably not integer and $I(i, j, k)$ is not defined. The image interpolation consists in computing the image value at any index (i, j, k) – integer or not – usually using the local neighborhood of (i, j, k). Many algorithms have been proposed in the literature and we present in this section a very short list of the most classical interpolation methods. In the case of (i, j, k) are out of the image bound is usually treated differently from the case where (i, j, k) are in the image bound. This case will not be treated below.

1.2.1 Nearest neighbor interpolation

The interpolation to the nearest neighbor is the fastest and the simplest interpolation method. Given (i, j, k) a non integer voxel coordinates, the image value at (i, j, k) is given by

$$I(i, j, k) \triangleq I(\underline{i}, \underline{j}, \underline{k}) \tag{1.21}$$

where $(\underline{i}, \underline{j}, \underline{k})$ are respectively the round to the nearest integer of (i, j, k). One particularity of the nearest neighbor interpolation – which can be considered as an issue – is that the image value is constant on an elementary volume centered around the integer coordinates ; if (i, j, k) are here integer coordinates, we have

$$I(i + \epsilon_1, j + \epsilon_2, k + \epsilon_3) = I(i, j, k) \tag{1.22}$$

for $\epsilon_{i=\{1,3\}} \in (-0.5, 0.5)$. Note that in the case where $|\epsilon_{i=\{1,3\}}| = 0.5$, the selected nearest neighbor depends on the implementation of the rounding procedure. Usually, the nearest neighbor of $i + 0.5$ is $i + 1$, and, as a consequence, the nearest neighbor of $i - 0.5$ is i.

1.2.2 Linear interpolation

Given a a real number, we define $\lceil a \rceil$ and $\lfloor a \rfloor$ respectively as the round up (towards plus infinity) andthe round down (towards minus infinity) of a. Let (i, j, k) be non integer voxel coordinates, and let α, β, γ be the weights defined as:

$$\alpha = \lceil i \rceil - i \tag{1.23}$$
$$\beta = \lceil j \rceil - j \tag{1.24}$$
$$\gamma = \lceil k \rceil - k \tag{1.25}$$
$$\tag{1.26}$$

The coordinates (i, j, k) belongs by definition to the intervale $[\lfloor i \rfloor, \lceil i \rceil] \times [\lfloor j \rfloor, \lceil i \rceil] \times [\lfloor i \rfloor, \lceil k \rceil]$. This intervale corresponds to a cuboid in the world coordinates where each one of its height corners is an element of the grid. The tri-linear interpolation uses the image values at these corners, the value at a given corner being simply weighted by a function of the distance between of the corner and the index (i, j, k):

$$\begin{aligned} I(i,j,k) \triangleq\ & I(\lfloor i \rfloor, \lfloor j \rfloor, \lfloor k \rfloor) * \alpha\beta\gamma \\ & + I(\lceil i \rceil, \lfloor j \rfloor, \lfloor k \rfloor) * (1-\alpha)\beta\gamma \\ & + I(\lfloor i \rfloor, \lceil j \rceil, \lfloor k \rfloor) * \alpha(1-\beta)\gamma \\ & + I(\lfloor i \rfloor, \lfloor j \rfloor, \lceil k \rceil) * \alpha\beta(1-\gamma) \\ & + I(\lceil i \rceil, \lceil j \rceil, \lfloor k \rfloor) * (1-\alpha)(1-\beta)\gamma \\ & + I(\lceil i \rceil, \lfloor j \rfloor, \lceil k \rceil) * (1-\alpha)\beta(1-\gamma) \\ & + I(\lfloor i \rfloor, \lceil j \rceil, \lceil k \rceil) * \alpha(1-\beta)(1-\gamma) \\ & + I(\lceil i \rceil, \lceil j \rceil, \lceil k \rceil) * (1-\alpha)(1-\beta)(1-\gamma \end{aligned} \tag{1.27}$$

One interesting property of this interpolation method is that, contrary to the nearest neighbor interpolation method, there are no discontinuity in terms of values in the image volume. Moreover, the formula (1.27) is still valid even if (i, j, k) are valid integer coordinates.
However, if the initial image was coded for instance on integer (*e.g.* `short`), the interpolation introduce non integer value that may be not adapted for some processing or visualization. In this case, the nearest neighbor interpolation is more adapted.

Chapter 2

Transformations

2.1 Linear transformation

A linear transformation is a function between two vector spaces (3-dimensional spaces in our case) that preserves the operations of vector addition and scalar multiplication. A linear transformation in a 3-dimensional space (input and output vector space) is usually given as a 4×4 matrix in homogeneous coordinates:

$$M = \begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.1}$$

Let p be a point in $\mathbb{R}^3$ and M be a linear transformation in the same space. The application of M to p

$$p' = M.p \tag{2.2}$$

transform p into a new point p'.

2.1.1 Rigid transformation

A rigid transformation preserves distances between every pair of points (isometry). In particular, any object will have the same shape and size before and after a rigid transformation.
A rigid transformation (in a 3-dimensional space) is parametrized by 6 parameters: 3 parameters for the rotation (resp. around the X-, Y-, and Z-axes) and 3 parameters for the translation. The parametrization of a rigid transformation is not unique. A parametrization is given relatively to a specific interpretation of these parameters. In ITK, there is at least two rigid transformation objects: `RigidTransform` and `Euler3DTransform`. The 3 rotation parameters correspond to a rotation vector for `RigidTransform` and to Euler's angles for `Euler3DTransform`. Given the 6 parameters and their associate interpretation, one can compute the 4×4 matrix as presented in 2.1.

2.1.2 Affine

An affine transformation preserves the collinearity relation between points and the ratios of distances along a line.
An affine transformation is parametrizes by 12 parameters: 3 for the rotation, 3 for the translation, 3 for the scaling, and 3 for the shearing. One may notice that the number of

parameters of an affine transformation is equal to the number of parameters of a linear transformation (see (2.1). Along with the parameters, an associated interpretation is needed in order to generate the corresponding transformation matrix.
In practice the parameters of an affine transformation are usually directly the parameters of the corresponding linear transformation (see (2.1)) since the number of parameters is similar. In this case, no parameter interpretation is required. This is the case for instance for the parameters of any ITK `AffineTransform` object. However, in the case of optimization for instance, one may still want to consider the initial parametrization (e.g. rotation, translation, etc. separated) in order to increase the precision and the robustness of the parameter estimation by adding some *a priori* on parameters.

2.2 Displacement field

In the context of medical imaging, a displacement field (DF for short) corresponds usually to a 3D grid defined into the 3D vector space. A 3D displacement vector is assigned to each element of the grid, hence the name *displacement field.* Like for images, each element of the grid can be projected to the world coordinate system using the position and orientation information associated to the field. The transformation is defined in a finite number of points and one should use a field interpolation (such as linear interpolation) to get the transformation in any point of the vector space.

A DF is usually stored into a vector image since the data organization is similar. The difference is indeed the interpretation of the content which is the integration of a signal on an elementary volume for an image, and a displacement for a DF. Storing a DF into an image is therefore very convenient.
If we consider now the programming point of view, considering a DF as an image is a mistake since an image is not a transformation. Mixing both notions may be quite confusing and dangerous. For instance, let us consider the ITK way of dealing with DFs. DFs are stored are store into ITK image objects. If one wants to resample an image using a DF, on should use the `itk::WarpImageFilter` which apply an image – the considered DF – to another image, and this clearly doesn't make sense. Moreover, displacement field is not the only field-type transformation (e.g. stationary velocity field). By considering a field as an image, a programmer cannot know what kind of field he is dealing with, unless the variable referencing the DF is correctly named which is usually too much to ask.
To overcome this problem, we propose a new ITK transformation object named `itk::DisplacementFieldTransform` which inherits from the `itk::Transform` class. In particular, `itk::DisplacementFieldTransform` allows to transform any point of the vector space (using linear interpolation) and is able to compute the spatial Jacobian (matrix) in any point. Because `itk::DisplacementFieldTransform` is an actual `itk::Transform` object, one may directly use `itk::ResampleImageFilter` to resample an image, which is much more convenient than using `itk::WarpImageFilter`. Finally, we propose the method `GetParametersAsVectorField` which returns a pointer the DF as an `itk::Image` in order to use the already implemented ITK filters specialized for DF.

2.3 Stationary velocity field

A stationary velocity field (SVF for short) is very similar to a displacement field, the difference being in the interpretation of the information (vector) assigned to each element of the grid. For SVFs, the vector represents a velocity which has to be integrated over time

to deduce a displacement.
Similarly to DFs, SVFs are usually stored into `itk::Image` objects which brings the same benefits and problems as for DFs (see section 2.2). Once again, we propose a solution with a new ITK transformation object named `itk::StationaryVelocityFieldTransform`. As for `itk::DisplacementFieldTransform`, `itk::StationaryVelocityFieldTransform` inherits from the `itk::Transform` class. For instance, the class allows to transform any point of the vector space and the method `GetLogSpatialJacobianDeterminant` computes the log determinant of the spatial Jacobian matrix.
The `itk::StationaryVelocityFieldTransform` can be used to resample an image using the `itk::ResampleImageFilter` filter. However, one should never use an `itk::StationaryVelocityFieldTransform` to do it because it will literally take days to complete the resampling. Instead, one should generate a DF directly from the considered SVF. Let `svf` be a `itk::StationaryVelocityFieldTransform` (of dimension 3 and coded on `float`) that we want to convert into a `itk::DisplacementFieldTransform`:

```
typedef itk::DisplacementFieldTransform<float,3> DFType;
typename DFType::Pointer df = DFType::New();
df->SetParametersAsVectorField(svf->GetDisplacementFieldAsVectorField()
    );
```

The methods `GetLogSpatialJacobianDeterminant` and `GetDisplacementFieldAsVectorField` depend on a `scheme` parameter. The *scaling and squaring* scheme is fast but tends to be noisy and not reliable if the transformation is too big. The *forward Euler* scheme is longer to compute but it is much more stable and reliable.

Chapter 3

Registration API

3.1 Goal

MIPS[1] is a set of tools (C++ libraries and executables) offering a suite of functionalities for processing and visualizing medical images and geometric meshes. The image registration tools, which now represent a major part of MIPS, have been developed for years by the different members of the Epidaure and Asclepios teams. Due to different causes (multiple image formats, evolution of registration technics, etc.) it is quite hard to compare two methods. For the same reasons, using two different methods into a registration workflow or simply switch from one method to another one is a complicated task. At best, one usually has to write huge and complex conversion procedures which is not trivial and is very time consuming. The goal of the registration API[2] is to address these issues by proposing a simple an intuitive interface shared by all registration methods. This common interface greatly simplifies the user experience while helping the author of a new registration method to focus on the algorithm, not on the user interface. We call this interface RPI for *Registration Programming Interface*.

3.2 RPI Specifications

3.2.1 `RegistrationMethod` class

RPI defines a C++ programming class named `RegistrationMethod`. This class gives to a method author a strict but simple programming framework. The `RegistrationMethod` class is an abstract class and does not allow to register images. Its purpose is to provide a common interface which has been designed to cover most of registration methods. In this section, we will briefly present the `RegistrationMethod` class and its key features. The presented code has been intentionally simplified to clarify the explanation. One should read the source code for a complete and accurate specification of RPI.

`RegistrationMethod` is a templated class:

```
template < class TFixedImage ,
           class TMovingImage ,
```

[1]Medical Image Processing and Simulation

[2]Application Programming Interface

```
              class TTransformScalar > class RegistrationMethod
```

`TFixedImage` and `TMovingImage` respectively represent the types of the fixed and the moving image. `TFixedImage` and `TMovingImage` must be `itk::Image` objects. As you may know, `itk::Image` is templated over the the image dimension and the pixel type. Dimension is usually 3 while pixel type can be either a scalar type (e.g. `short`, `float`, etc.) or can be a `itk::Vector` for diffusion tensor images (DTI for short). Type `TTransformScalar` defines the type of the transformation parameters (e.g. type of each element of a affine transformation matrix). It should be either `float` or `double`. Some registration methods support different types of images and/or different types of transformation parameters (e.g. Optimus method) while other methods only support one type of images (e.g. Log-demons) or transformation parameters. In all cases, the image and transformation parameter types supported by a given registration method must clearly be stated in the code documentation. It is not of the responsibility of the user to guess the type of images and transformation parameters supported.

The `RegistrationMethod` class provides all the basic necessary functions to register two images. First, the fixed and the moving images can be set and get using the following accessors:

```
TFixedImage::ConstPointer   GetFixedImage(void);
void                        SetFixedImage(const TFixedImage * image);
TMovingImage::ConstPointer  GetMovingImage(void);
void                        SetMovingImage(const TMovingImage * image);
```

Images must contain all the information necessary to orient and positioned images in the real word coordinates. One should note that the notion of smart pointer used here (`Pointer` and `ConstPointer`) is very common in ITK. The registration process can be started using the function:

```
virtual void                StartRegistration(void) = 0;
```

As the reader may notice, this function is pure virtual since this class is only meant to be inherited. Once the registration process is complete, one can get the output transformation using the function:

```
TransformPointerType        GetTransformation(void) const;
```

where

```
typedef itk::Transform< TTransformScalarType,
                        TFixedImage::ImageDimension,
                        TMovingImage::ImageDimension > TransformType;

typedef typename TransformType::Pointer TransformPointerType;
```

The computed transformation must be a transformation in the real world coordinates allowing to resample the moving image, *i.e.* the transformation from the fixed image to the moving image.

Images and transformations are stored into ITK containers. This choice was decided by the Asclepios team members and was justify by the following points:

- ITK is an open-source library.
- The ITK code is maintained by hundreds of developers.
- All the registration methods recently developed by the team use ITK objects for images and transformations.
- The IO procedures to read/write images and transformations are provided by ITK.
- Most of image format are supported by ITK.
- ITK provides useful tools to manipulate images and transformations.

Any transformation computed by any registration method of RPI must inherit from the `itk::Transform` class (see section2).

ITK also provides a registration framework and one may ask why RPI is not included into this framework. The ITK registration framework is too specific and does not allow to easily integrate few of Asclepios' registration methods such as *Baladin* and *SuperBaloo*. As a consequence, it has been decided not to base the registration API on this framework but to design a new framework more simple and more generic. Only the ITK containers were used for the reasons given previously.

3.2.2 Creating a new registration class

Any registration method of RPI must inherit from the super class `RegistrationMethod`. By definition, a registration method (*i.e.* derived class) inherits all the necessary functions to set inputs, start registration, and get the output transformation. Most of registration methods use parameters to modify the method behavior. The parameters (class members) as well as the corresponding accessors must be implemented into the method – derived – class. Several registration methods share parameters such as the maximum number of iterations. In order to have methods as consistent as possible, parameters should be consistent as well (same member's name, same accessor's name). Finally, the code must be clean, should use the latest programming standards, and must be well documented (doxygen style).

3.2.3 Libraries

So far, most of registration methods of MIPS unfortunately can be called only using a specific executable ; one of them offered a C++ library that could eventually be used within a third party application such as MedInria. In RPI, every single registration method must be compiled into a library. By following all the specifications presented in this document (inheritance from `RegistrationMethod`, use ITK images and transformations, *etc.*), a library corresponding to a given registration method (called here `MyMethod`) can then be trivially used into an external application:

```
typedef MyMethod<TFixedImage,TMovingImage,TransformScalarType> Method;

MyMethod * method = new MyMethod();
```

```
method->SetFixedImage(fixed);
method->SetMovingImage(moving);
method->StartRegistration();
transform = method->GetTransformation();

// Use the transformation to do what you need to do...

delete method;
```

This minimal code register the two images with default parameters. To modify the method parameters, one should use the correct accessors.

3.2.4 Executables

Along with the library, each registration method must provide an executable based on its library. The executable workflow should be (more or less) the following:

1. Read input images (paths given in the command line arguments).
2. Parse method parameters from command line arguments.
3. Create a registration object.
4. Set input images and method parameters.
5. Start registration.
6. Get and save the transformation into an output file.
7. Resample the moving image using the transformation and save it into an output file.

An important part of a usable executable is the parsing of the command line arguments. Among all existing tools, we propose to use the TCAL library. TCLAP provides a simple command line parser and automatically generates the usage information ("help") using options `--help` or `-h`. The author must pay an extra attention to the description of the usage information. It is usually the only available help a user may have to learn how to correctly use a given executable.

All registration executables must share the following options:

- `-f` and `--fixed-image` allows to specify the path to the fixed image.
- `-m` and `--moving-image` allows to specify the path to the moving image.
- `-t` and `--output-transform` allows to specify the path to the output transformation.
- `-i` and `--output-image` allows to specify the path to the output image (moving image resampled).

One easily understands that since these options define the minimal input of any registration method, it becomes trivial to switch from one registration method to another one. Only advanced options must be modified.

Finally, since only a single library is shared by all applications (executable and third party application), there is no duplication of the registration code. All contributors work on the same code. The code is then easier to maintain and contains necessarily less bugs. The transformation computed by an executable or by third party application (same method and same parameters) is by definition identical.

3.2.5 TRex

In order to help researchers and engineers to develop and integrate their registration method into RPI, we have developed a toy example. This example named TRex – for *Toy Registration EXample* – is a registration method fully based on ITK. The transformation computed is a rigid transformation and the optimization method is a simple gradient descent. This example shows how to write a registration method (class) inheriting from `RegistrationMethod`, how to compile the class into a library, and how to write the code which will be compiled into a usable executable with the appropriate usage information. This example is simple, well documented and very easy to understand. The method has only one parameter.

3.2.6 Observers

Depending on the registration method, the set of parameters used, and the input images, the registration process may be very long task. If the considered method is included into an application such as MedInria, it can be very useful to know the progression of the registration process. For instance, the software is usually waiting for the process to be finished before updating the current view. Therefore, we have developed an event observer. An observer is an object which is linked to the considered registration method. When the progression of the registration process changes (*e.g.* registration starts, stops, *etc.*), all the observers are notified. Each notified observer does a specific task defined by the user, for instance update the current view if the registration process is finished.
The observer class in RPI is named `rpi::Observer` and is based on a very simple but efficient design pattern. A user that would want to define an observer must create a class that inherits from `rpi::Observer` and implement the `Update()` method which will be called by the registration method during the notification to observers:

```
template < class TFixedImage,
           class TMovingImage,
           class TTransformScalarType=double >
class MyObserver : public rpi::Observer<TFixedImage,
                                        TMovingImage,
                                        TTransformScalarType>
{
public:

  typedef rpi::RegistrationMethod<TFixedImage,
                                  TMovingImage,
                                  TTransformScalarType> Method;

  void Update(void)
  {
    if (this->m_registrationMethod->GetRegistrationStatus()==
      Method::REGISTRATION_STATUS_PROCESSING)
    {
      std::cout << "I am processing..." << std::endl;
    }
    else if (this->m_registrationMethod->GetRegistrationStatus()==
      Method::REGISTRATION_STATUS_STOP)
    {
      std::cout << "I have finished!" << std::endl;
```

```
        }
    }
};
```

The observer is a templated class since the observer object must contain an instance of the `rpi::RegistrationMethod` object. The signification of the templates is similar too the templates used by `rpi::RegistrationMethod`. In the example given previously, one understand what does the considered observer: the `Udpate()` method displays "I have finished!" if the registration process is finished, and "I am processing..." otherwise.
To attach an observer to a registration method, simply use the `AttachObserver()` method. The class `rpi::RegistrationMethod` also provides the methods `GetRegistrationStatus()` and `SetRegistrationStatus()` which respectively gets and sets the status of the registration process. Two status are provided: `REGISTRATION_STATUS_PROCESSING` and `REGISTRATION_STATUS_STOP`. These two status represent the minimum information on the registration status one could expect. However, if a registration method provides a better granularity of the registration progression, this fineness may be used by the observer. In this case, the considered method updates a specific instance variable at each iteration of the algorithm and notify its associated observer. Then, the observers can access – directly in the `Update()` method – to the *improved* progression status using the adapted accessor. This can be very useful for progression bar for instance.

3.3 RPI methods

In this section, we briefly present all the registration methods implemented into RPI. For a complete and accurate description of the method and of the parameters used, please refer to the corresponding code and publication (if any).

3.3.1 Baladin

Baladin [ORPA00] is a registration method for scalar images and based on a pyramidal block-matching algorithm. Blocks are built in the moving image and, for each block, the most similar block is searched in the fixed image (exploration). Each paring of blocks defines a displacement vector. The set of displacement vector is finally used to estimate a linear transformation from the fixed to the moving image. The linear transformation computed can be a rigid transformation, a similitude, or an affine transformation.
The output transformation computed by the library is represented as an `itk::AffineTransform<ScalarType,3>` object where `ScalarType` is either `float` or `double`. The output transformation computed by the executable is represented by an `itk::AffineTransform<double,3>` stored into a text file.
Baladin can be initialized with a linear transformation (rigid, similitude, or affine) computed using Baladin, Baloo, or Optimus. If no initial transformation is provided, the initial transformation is simply the identity. If the type of the initial transformation is different from the type specified, the output transformation may not be of the type specified. The reason is that Baladin computes a linear transformation of the specified type that will be fused to the global transformation. For instance, if the initial transformation is the identity, the composition of rigid transformations is a rigid transformation. However, if the initial transformation is affine, the composition of affine and rigid transformations is an affine transformation.

3.3.2 Optimus

Optimus is a registration method for scalar images and based on the New UOA optimizer of Powell. The transformation computed by the registration method is a rigid transformation represented as an `itk::Euler3DTransform<double,3>` object and written into a text file for the executable. Optimus can only be initialized by an `itk::Euler3DTransform<double,3>` transformation. So far, the only registration in RPI able to compute an `itk::Euler3D-Transform<double,3>` is Optimus.

3.3.3 Baloo

Baloo is a registration method for scalar and tensor images (2 classes) and based on a pyramidal block-matching algorithm. The transformation computed by the registration method is a linear transformation among rigid, similitude, or affine transformation. The output transformation computed by the library is represented as an `itk::Affine-Transform<ScalarType,3>` object where `ScalarType` is either `float` or `double`. The output transformation computed by the executable is represented by an `itk::Affine-Transform<double,3>` stored into a text file. Baloo cannot be initialized by any transformation.

3.3.4 SuperBaloo

Similarly to Baloo, SuperBaloo is a registration method for scalar and tensor images (2 classes) and based on a pyramidal block-matching algorithm. However, SuperBaloo computes a displacement field transformation.
The output transformation computed by the library is represented as an `itk::Displacement-FieldTransform<ScalarType,3>` object where `ScalarType` is either `float` or `double`. The executable writes the output transformation into a vector image (NifTi, Inrimage, NRRD, etc.) where values are coded as `float`. SuperBaloo cannot be initialized by any transformation.

3.3.5 Diffeomorphic Demons

Diffeomorphic demons is a registration method for scalar images and based on the demons algorithm. The difference with the classical demons is that the transformation computed – a displacement field – is a diffeomorphism.
The output transformation computed by the library is represented as an `itk::Displacement-FieldTransform<ScalarType,3>` object where `ScalarType` is either `float` or `double`. The executable writes the output transformation into a vector image (NifTi, Inrimage, NRRD, etc.) where values are coded as `float`.
The library can be initialized only by an `itk::DisplacementFieldTransform<SclaraType,-3>` object. The executable can be initialized by a displacement field (vector image) or by an (ITK) linear transformation. In the case of linear transformation, the input transformation is converted into a displacement field which can initialize the corresponding library.

3.3.6 Log-Demons

Log-Demons is a registration method for scalar images and based on the demons algorithm. The difference with the classical demons is that the transformation computed is a stationary velocity field (SVF for short) and not a displacement field. This type of

transformation is a convenient tool for registration since a SVF can be trivially converted into a displacement field that is a diffeomorphism.
The output transformation computed by the library is represented as an `itk::StationaryVelocityFieldTransform<ScalarType,3>` object where `ScalarType` is either `float` or `double`. The executable writes the output transformation into a vector image (NifTi, Inrimage, NRRD, etc.) where values are coded as `float`.
The library can be initialized only by an `itk::StationaryVelocityFieldTransform<SclaraType,3>` object. The executable can be initialized by a SVF (vector image) or by an (ITK) linear transformation. In the case of linear transformation, the input transformation is converted into a SVF which can initialize the corresponding library.

3.3.7 Incompressible Log-Demons

Incompressible Log-demons – or iLogDemons for short – is a registration method for scalar images and based on the Log-demons algorithm. The transformation computed is a stationary velocity field (SVF for short). The difference with the Log-demons is that the method accept an input mask image ; the iLogDemons adds a constraint on the computed SVF so that the volume of the zone defined into the mask is similar before and after applying the transformation.
The output transformation computed by the library is represented as an `itk::StationaryVelocityFieldTransform<ScalarType,3>` object where `ScalarType` is either `float` or `double`. The executable writes the output transformation into a vector image (NifTi, Inrimage, NRRD, etc.) where values are coded as `float`.
The library can be initialized only by an `itk::StationaryVelocityFieldTransform<SclaraType,3>` object. The executable can be initialized by a SVF (vector image) or by an (ITK) linear transformation. In the case of linear transformation, the input transformation is converted into a SVF which can initialize the corresponding library.

3.4 Image resampling

3.4.1 Theory

Let F and M be respectively the fixed and the moving images. If we want to register M on F, we want to modify M so that the modified – or transformed – version of M can be superimposed on F. Instinctively, one would want to estimate the transformation from M to F in order to apply this transformation to M. Well, in fact, we need the transformation from F to M to resample M in the geometry of F. This is not trivial and we will try to convince you in this section.
Let $\widetilde{M}$ be the resampled version of M in the geometry of F. Theoretically, $\widetilde{M}$ and F can be superimposed and then compared for diagnosis purpose for instance. First, we create an empty image $\widetilde{M}$ which has exactly the same geometrical properties of F (image size, voxel size, orientation, and origin). Second, we have to fill $\widetilde{M}$ with information **taken** from image M. Let (i,j,k) be an index (voxel coordinates) of $\widetilde{M}$ and let (x,y,z) be the corresponding world coordinates. Then, let $T_{F\to M}$ be the transformation from F to M. The value assigned to $\widetilde{M}(x,y,z)$ (world coordinates) is given by:

$$\widetilde{M}(x,y,z) = M({}_{F\to M}(x,y,z)) \tag{3.1}$$

If the world coordinates $T_{F\to M}(x,y,z)$ are not defined on the grid of M, $M({}_{F\to M}(x,y,z))$ is estimated by image interpolation (see section 1.2).

In other words, to resample M, we go through the voxels of $\widetilde{M}$. For each voxel of $\widetilde{M}$ we deduce the position of the corresponding voxel into M using the transformation ${}_{F\rightarrow M}$, and then we **take** the value at this position in M and assigned it to the initial position in $\widetilde{M}$.

Still not convince? Let's see now what would happen if we had the transformation $T_{M\rightarrow F}$ from M to F. As before, we first create an empty image $\widetilde{M}$ which has exactly the same geometrical properties of F. We go through M and for each voxel of M, we send its corresponding value to $\widetilde{M}$ using $T_{M\rightarrow F}$. If (x, y, z) is the world coordinates of a voxel of M, we have:

$$\widetilde{M}(T_{M\rightarrow F}(x, y, z)) = M(x, y, z) \tag{3.2}$$

First difficulty here, if coordinates $T_{M\rightarrow F}(x, y, z)$ are not defined on the grid of $\widetilde{M}$, how to assign – or distribute – the value $M(x, y, z)$ in $\widetilde{M}$? For the next problem, let us assume that coordinates $T_{M\rightarrow F}(x, y, z)$ are always defined on the grid of $\widetilde{M}$ in order to simplify the explanation. Let us consider the case where M contains n voxels while F contains $2n$ voxels. Using the second scheme (transformation from M to F), one understands that only half of voxels of $\widetilde{M}$ will receive a value from M. The other half won't be modified and consequently $\widetilde{M}$ will have "holes".

3.4.2 `rpiResampleImage`

The `rpiResampleImage` tool resample an input scalar image given an input transformation. The supported transformations are identity, linear transformations, displacement field transformations, and stationary velocity field transformations computed using the registration methods of RPI (but not only). The image specified using the `--geometry` option is used to set the resample image geometry. If the `--geometry` option is not used, the geometry of the resampled image will be either similar to the input image geometry if the input transformation is a linear transformation, or will be similar to the geometry of the input field is the input transformation is a displacement field or a stationary velocity field. Several image interpolation methods are proposed.

3.5 Composition of transformations

3.5.1 Order of composition

Let us consider two images I_F and I_M (resp. fixed and moving images). We want to register I_M to I_F and a common way to do it is to apply successively different registration methods. In this example, we use three registration methods (e.g. rigid, affine, and non-linear). First, we apply the first registration method between images I_F and I_M. We obtain a transformation (form the fixed to the moving image) T_0 and an image I_0 (I_M re-sampled in the geometry of I_F):

$$\text{method}_1(I_F, I_M) \rightarrow \{T_1, I_1\} \tag{3.3}$$

Then we apply the second registration method between images I_F and I_1. By considering the image I_1 instead of I_M, we initialize the registration process by starting the algorithm *closer* to the optimal transformation. We obtain a transformation T_2 and an image I_2 (I_1 re-sampled in the geometry of I_F):

$$\text{method}_2(I_F, I_1) \rightarrow \{T_2, I_2\} \tag{3.4}$$

Finally, we apply the third registration method between images I_F and I_2. We obtain a transformation T_3 and an image I_3 (I_2 re-sampled in the geometry of I_F):

$$\text{method3}(I_F, I_2) \rightarrow \{T_3, I_3\} \tag{3.5}$$

If we assume that the successive registration processes improved the image registration, the image I_3 should be the image the most similar to I_F, then I_2, I_1, and finally I_M. An important issue here is that I_3 is the result of three consecutive re-samplings of the initial moving image I_M. In order to avoid the multiple approximations due to these re-samplings, one may want to re-sample only once I_M using the global transformation from I_F to I_M. However, this global transformation is not T_3 but the composition of T_1, T_2, and T_3. Indeed, T_3 is the transformation from I_F to I_2 ; the image I_M re-sampled using T_3 is probably not correctly registered onto I_F.

Let us consider the image re-sampling of I_M using T_1. Let $\mathbf{x} = (x, y, z)$ be a point (in the real world coordinates) in the real grid of the fixed image. The re-sampling is defined as:

$$I_1(\mathbf{x}) = I_M(T_1(\mathbf{x})) \tag{3.6}$$

Now, let us now consider the image re-sampling of I_1 using T_2:

$$\begin{aligned} I_2(\mathbf{x}) &= I_1(T_2(\mathbf{x})) && (3.7)\\ &= I_M(T_1(T_2(\mathbf{x})) && (3.8) \end{aligned}$$

Finally, we re-sample I_2 using T_3:

$$\begin{aligned} I_3(\mathbf{x}) &= I_2(T_3(\mathbf{x})) && (3.9)\\ &= I_1(T_2(T_3(\mathbf{x})) && (3.10)\\ &= I_M(T_1(T_2(T_3(\mathbf{x}))) && (3.11)\\ &\triangleq I_M(T_1 \circ T_2 \circ T_3(\mathbf{x})) && (3.12) \end{aligned}$$

Eq. 3.12 shows that the global transformation allowing to re-sample the image I_M onto I_F using a single global transformation T which is the composition of T_1, T_2, and T_3:

$$T = T_1 \circ T_2 \circ T_3 \tag{3.13}$$

3.5.2 Composing transformations using `rpiFuseTransformation`

The `rpiFuseTransformation` executable is a program that allow to fuse (compose) a list of transformation into a unique transformation. Since it is not possible yet – at least in a general case – to store a list of transformations into a single file, we propose to store the list as a XML file. This XML lists the type of each transformation and the path to the files containing the transformations :

```
<?xml version="1.0" encoding="UTF-8"?>

<listoftransformations>
```

```
    <transformation>
        <type>linear</type>
        <path>t0.txt</path>
        <invert>1</invert>
    </transformation>

    <transformation>
        <type>displacementfield</type>
        <path>t1.nii.gz</path>
        <invert>0</invert>
    </transformation>

</listoftransformations>
```

In this example, we have 2 transformations. The first one (denoted T_0) is a linear transformation stored into the text file “t0.txt”. The second transformation (denoted T_1) is a displacement field stored into the – image – file “t1.nii.gz”. The computed transformation is given by $T_0 \circ T_1$. Thus, the order the transformations written into the XML file is important since the composition of two transformations is usually not commutative (see section 3.5.1). The XML file can contain any strictely positive number of transformations. The meaning of the different tags is:

- Tag `<listoftransformations>` contains a set of transformations. This tag is mandatory.
- Tag `<transformation>` contains a transformation. This tag must be an element of the tag `<listoftransformations>`. This tag is mandatory.
- Tag `<type>` contains the type of the considered transformation. The available types are “linear”, “displacemendfield”, and “stiaionaryvelocityfield”. This tag must be an element of the tag `<transformation>`. This tag is mandatory.
- Tag `<path>` contains the path to the file containing the transformation. A displacement field or a stationary velocity field must be stored into an image file (e.g. Nifty, Analyse, etc.). This tag must be an element of the tag `<transformation>`. This tag is mandatory.
- Tag `<invert>` indicates if the considered transformation has to be inverted before the fusion. The value “1” induces the transformation inversion ; the value “0” means no inversion. This tag is optional. By default, no inversion is performed.

The type of the output transformation depends on the transformations of the list. A list of linear transformations will be fused into a linear transformation. If the list contains at least one displacement field, the output transformation will be a displacement field. If the option ”–force-displacement-field” is used, the ouput transformation will be a displacement field. For now on, if the list contains linear transformations and stationary velocity fields, the computed transformation will be a displacement field. However, this will change in the near future in order to compute a unique stationary velocity field (thanks to Marco Lorenzi).
One should choose the output file extension according to the transformation computed. Indeed, if the output transformation is a displacement field, only image format are supported (e.g. .nii). No checking on the file extension will be performed since the output transformation type is easily predictable.

3.6 Conversion of transformations

In this section, we present the conversion of a transformation of the voxel space into a transformation of the world space, and conversely. We first introduce in Fig. 3.1 a scheme that will be useful to give an intuition of how the conversion procedure works. This array visually represent the projection of a point from one image to another one, or from coordinate system to another one. On the top are represented the fixed and the moving images and on the left hand-side are represented the voxel and the world coordinate systems. We present here two arrows representing the projection of a point from the voxel coordinates to the world coordinates (see Section 1.1.1). In this section, M_{fixed} and M_{moving} will respectively represent the projection (4×4) matrix of the fixed and moving images.

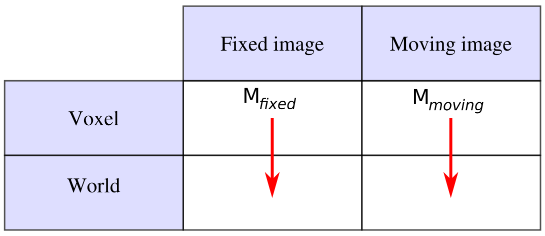

Figure 3.1: Projection matrices from the voxel coordinates to the world coordinates.

3.6.1 Linear transformation: from voxel to world coordinates

Let T_{voxel} be a linear transformation (4×4 matrix) from the fixed image to the moving image and given in the voxel coordinates (see Fig. 3.2). The conversion of T_{voxel} into T_{world}, the corresponding linear transformation in the world coordinates, is given by:

$$T_{world} = M_{moving}.T_{voxel}.M_{fixed}^{-1} \tag{3.14}$$

Fig. 3.3 gives a visual explanation of the conversion. This conversion is used in the registration method Baloo where the initial code returns a linear transformation such as T_{voxel}. In the original version of Baladin, the returned transformation T_{voxel} is a linear transformation from the moving image to the fixed image and given in the voxel coordinates. In this particular case, T_{voxel} must be inverted before computing T_{world} (from fixed to moving image) as proposed in Eq. 3.14.
From Eq. 3.14 one trivially converts a linear transformation from world coordinates to voxel coordinates:

$$T_{voxel} = M_{moving}^{-1}.T_{world}.M_{fixed} \tag{3.15}$$

In the case of Baladin, the final transformation T_{voxel} since Baladin consider transformations from the moving image to the fixed image.

3.6.2 Displacement field: from voxel to world coordinates

The conversion of a displacement field is different from the conversion of a linear transformation. Indeed, a displacement vector is assigned to each element of the field. Let p_{voxel} be an index of the fixed image (point in the voxel coordinates) and let v_{voxel} be its

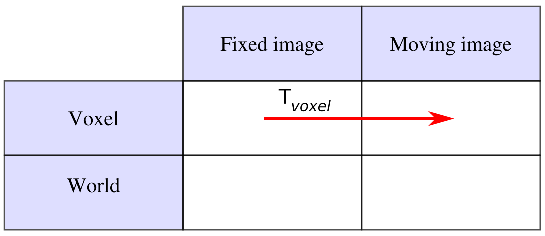

Figure 3.2: Transformation from the fixed image to the moving image and given in the voxel coordinates.

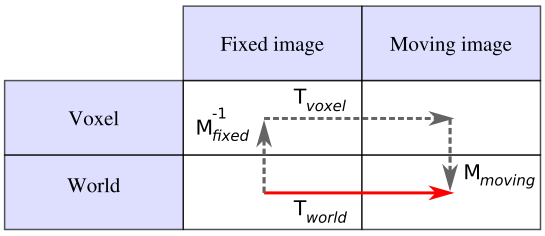

Figure 3.3: Conversion procedure: from voxel to world coordinates.

associated displacement vector given in voxels. Let p_1 be the projection of p_{voxel} into the world coordinates:

$$p_1 = M_{fixed} . p_{voxel} \tag{3.16}$$

By definition, $p_{voxel} + v_{voxel}$ is an index in the moving image (point in the voxel coordinates). Let be p_2 be the projection of $p_{voxel} + v_{voxel}$ into the world coordinates:

$$p_2 = M_{moving} . (p_{voxel} + v_{voxel}) \tag{3.17}$$

Both p_1 and p_2 are points in the world coordinates. The displacement vector, denoted v_{world}, in the world coordinates corresponding to v_{voxel} is simply the difference between p_2 and p_1 (see Fig. 3.4):

$$v_{world} = M_{moving} . (p_{voxel} + v_{voxel}) - M_{fixed} . p_{voxel} \tag{3.18}$$

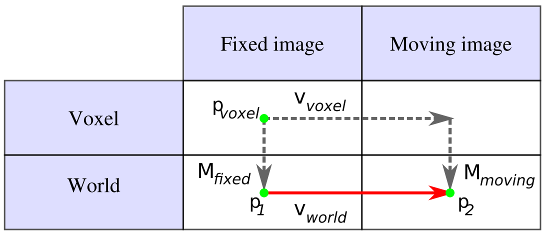

Figure 3.4: Conversion of a displacement field from voxel to world coordinates. Green circles represent (from top to bottom and left to right) p_{voxel}, p_1, and p_2.

3.7 Other useful tools

3.7.1 imageConvert

The **imageConvert** tool converts an image from one format to another. The format of each image is determined by its extension. Scalar images (2D and 3D) as well as DTI (3D) images are supported. Using some appropriate options, one can also modify the image origin, the voxel size, and the image orientation (direction). This will affect only the image header but not the content.

3.7.2 imageDisplayProperties

The **imageConvert** tool displays the properties of an input image. Scalar and vector images (DTI) are supported. 2D and 3D images are supported. See the example below to know what kind of information is displayed:

```
[vgarcia@koch ~]$ imageDisplayProperties ~/Data/T1.nii
Image name        : /user/vgarcia/home/Data/T1.nii
Image dimension   : 3
Image size        : [ 181, 217, 181] voxels
Pixel type        : scalar
Component number  : 1
Component type    : float
Component size    : 4 bytes
Image origin      : [    0.000,    0.000,    0.000 ]
Voxel size        : [    1.000,    1.000,    1.000 ]
Direction cosine  : [    1.000,    0.000,    0.000 ]
                    [   -0.000,   -1.000,    0.000 ]
                    [   -0.000,    0.000,    1.000 ]
```

3.7.3 imageResize

The **imageResize** tool resizes an input image given either the new voxel size or the new image size. Voxel size and image size cannot be set simultaneously. Image orientation is preserved. Image origin may be shifted in order to perserved the borders of the image in the real world coordinates. However, if the voxel size set does not allow to discretize the initial volume into an integer number of voxels, the image size in units (image size in voxels * voxel size) and the image border of the output image will be different from those of the input image.

3.8 Things to do

3.8.1 Tensor support for Baloo and SuperBaloo

Baloo and SuperBaloo are so far the only registration methods of RPI supporting tensor images. However, this support is not guaranteed since Olivier Commowick is still working on tensor registration. The core code of Baloo and SuperBaloo may be updated and therefore Baloo and SuperBaloo (RPI versions) may not work after this update.
The resampling of a tensor image is not as easy as for scalar images. You usually have to interpolate the tensor image and you also have to rotate it. So far, we use the `itk::WarpTensorImageFilter` of TTK which applies a displacement field (as a vector

image) to the tensor image. In the case of Baloo, we estimate a linear transformation. To resample the tensor, we generate a displacement field from the linear transformation, and then we use `itk::WarpTensorImageFilter`. This solution is of course not the correct way to resample a tensor using a linear transformation. A proper function should be written to directly use the linear transformation.

3.8.2 Improve method `GetSpatialJacobian` of the `itk::DisplacementField-Transform` class

The `GetSpatialJacobian` method of the `itk::DisplacementFieldTransform` class computes the gradient using a finite difference and considering the neighborhood along one direction for each element of the matrix. The problem of this method is that the estimation of the spatial Jacobian is incorrect and tends to be noisy. A better way to do it would be to consider a larger neighborhood as proposed in [PFA04].

3.8.3 Improve method `TransformPoint` of the `itk::StationaryVelocity-FieldTransform` class

The `TransformPoint` method of the `itk::StationaryVelocityFieldTransform` class first go through all the image to estimate a value (see Xavier or Marco) and then transform the considered point by composing several time with the velocity field. The problem of this method is that if one uses the `itk::ResampleImageFilter` to resample a given image using a stationary velocity field, the `TransformPoint` method will be called by every voxel of the image to resample. This means that, for each voxel, the whole image is explored. As a consequence, using a stationary velocity field to resample an image can take days. As proposed in section 2.3, one should first compute a displacement field from the stationary velocity field, and then resample the image using the displacement field.

3.8.4 Improve the multi-resolution scheme

The multi-resolution scheme used by the demons-based registration methods use a pyramidal approach where each dimension is divided by 2 from one resolution to another one. One the image size and voxel size are homogeneous, this works perfectly. However, if one consider for instance an image of size $512 \times 256 \times 64$, it may be more interesting to consider the following pyramidal approach:

- $512 \times 256 \times 64$
- $256 \times 256 \times 64$
- $128 \times 128 \times 64$
- $64 \times 64 \times 64$
- $32 \times 32 \times 32$

The goal here is to reduce the size of the image in order to tend to have images of size as isotrope as possible. I haven't tried this, but if the voxels also have an isotropique size, this makes sense.

Bibliography

[ORPA00] S. Ourselin, A. Roche, S. Prima, and N. Ayache. Block matching: A general framework to improve robustness of rigid registration of medical images. In *MICCAI '00*, volume 1935 of *LNCS*, pages 557–566, 2000.

[PFA04] Xavier Pennec, Pierre Fillard, and Nicholas Ayache. A Riemannian Framework for Tensor Computing. Technical report, INRIA, July 2004.