SimpleElastix Documentation

Release 0.1

Kasper Marstal

May 11, 2017

Contents

SimpleElastix is a medical image registration library that makes state-of-the-art image registration really easy to do in languages like Python, Java and R. For example, using the following line of code (Python),

```
import SimpleITK as sitk
resultImage = sitk.Elastix(sitk.ReadImage("fixedImage.nii"), sitk.ReadImage(
↪"movingImage.nii"))
```

we can register two human brain images:

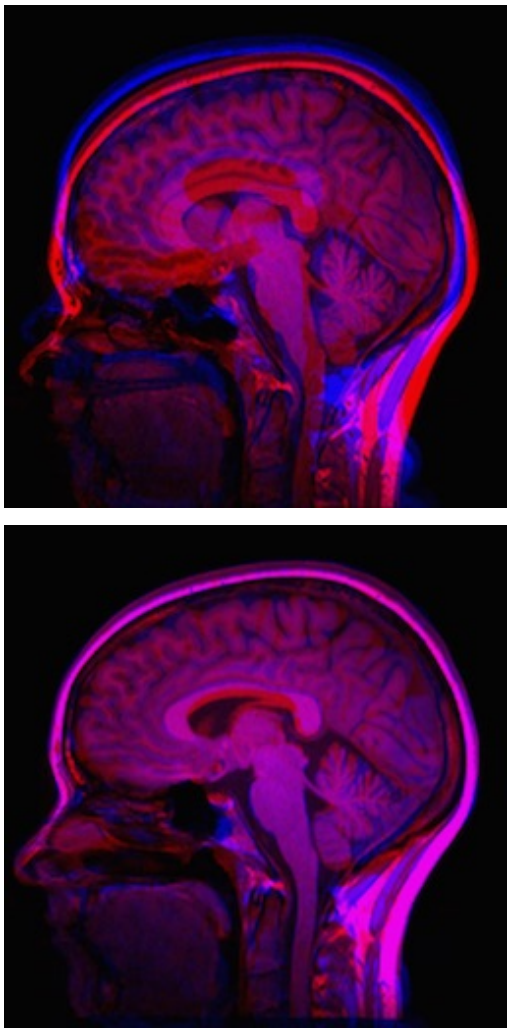


Figure 1. Original mean image of two different subjects (left) and registered mean image (right) using the line of code above. Purple areas indicate good alignment and blue and red areas indicate misalignment.

The *Getting Started* section explains how to clone the Github repository and compile SimpleElastix on *Linux*, *Mac OS X* and *Windows*. The subsequent sections present how to use elastix, and gradually introduce more advanced features and use cases. If you have not used elastix or transformix before, we highly recommend that you familiarize yourself with the Introduction, Hello World example and Parameter Maps sections before working through the examples.

CHAPTER 1

Getting Started

This page explains how to install SimpleElastix. The process involves compiling the C++ project and linking against a target language from which you would like to use SimpleElastix. SimpleElastix can be linked against Python, Java, R, Ruby, Octave, Lua, Tcl and C#.

SuperBuild On Linux

SimpleElastix can be compiled with the SuperBuild. The SuperBuild is a script that automatically downloads and install any dependencies so you can don't have to install elastix, ITK or SWIG on beforehand. The only thing you need is CMake, git and a compiler toolchain. To build SimpleElastix, use the following commands to download the code and start the build:

```
$ git clone https://github.com/kaspermarstal/SimpleElastix
$ mkdir build
$ cd build
$ cmake ../SimpleElastix/SuperBuild
$ make -j4
```

When the project has been built, we can find language packages in the `${BUILD_DIRECTORY}/SimpleITK-build/Wrapping` directory. For example, to install the python module onto your system, navigate to

```
${BUILD_DIRECTORY}/SimpleITK-build/Wrapping/Python/Packaging
```

and run the following command:

```
$ sudo python setup.py install
```

This will install the SimpleITK python module with SimpleElastix unto your system, which can then be imported into your scripts like any other python module.

Target language dependencies need to be pre-installed. The relevant `apt-get` packages are

```
python python-dev monodevelop r-base r-base-dev ruby ruby-dev tcl tcl-dev tk tk-dev
```

Note that this project takes around an hour to build on a quad-core machine. SimpleElastix has been tried and tested on Ubuntu 14.10 using GCC 4.9.2 and Clang 3.4.0, Mac OSX Yosemite using Apple Clang 600.0.56 and Windows 8.1 using Microsft Visual Studio 2010 C++ compiler.

Warning: Be careful not to run out of memory during the build. A rule of thumb is that we need 4GB of memory per core. For example, if we compile SimpleElastix with 4 cores (e.g. `make -j4`) we need a machine with at least 16GB of RAM.

SuperBuild On Mac OS X

The Mac OS X installation procedure is identical to that of Linux, so simply follow the Linux installation steps above to install SimpleElastix. Mac OS X comes with Python and Tcl preinstalled. Other target-language dependencies need to be installed seperately. This can be done with Macports or Homebrew.

It is assumed that a CMake, git and a compiler toolchain has already been installed. We can check for a working compiler by opening the OS X terminal and run `make`. OS X will know if the Xcode Command Line Tools is missing and prompt you to install them if this is the case.

SuperBuild On Windows

SimpleElastix can be compiled with the SuperBuild. The SuperBuild is a script that automatically downloads and install any dependencies so you can don't have to install elastix, ITK or SWIG on beforehand. The only thing you need is CMake, git and a compiler toolchain.

Using the command line

We will use CMake to generate build files and the msbuild.exe program to compile the project (which is also what Visual Studio uses under the good).

1. **Download CMake, git and code, and setup directories.**
    - Download and install CMake GUI. Be sure to select *Add CMake to the system PATH* option.
    - `git clone https://github.com/kaspermarstal/SimpleElastix` into a source folder of your choice. You can install GitHub Desktop and use the accompanying command line tool with git automatically added to its path.
    - Make a new directory named *build* and cd into it by typing *cd build*. Here we will assume that the build directory and the source directory is in the same folder.
2. **Compile the project.**
    - Open "Developer Command Prompt for VS2015" (or equivalent depending on your version of Visual Studio)
    - Run *cmake ../SimpleElastix/SuperBuild*.
    - Run *msbuild /p:configuration=release ALL_BUILD.vcxproj*.
3. **Enable x64 bit build (Optional).**

- Prior to running the *cmake* command in step 2, navigate to *C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC>* (or equivalent depending on your version of Visual Studio) and *vcvarsall amd64*.

Using Visual Studio

Will use CMake to generate build files and the Visual Studio compiler to compile the project.

1. **Download CMake, git and code, and setup directories.**
    - Download and install CMake GUI.
    - `git clone https://github.com/kaspermarstal/SimpleElastix` into a source folder of your choice.
    - Point the CMake source directory to the `SimpleElastix/SuperBuild` folder inside the source directory.
    - Point the CMake build directory to a clean directory. Note that Visual Studio may complain during the build if the path is longer than 50 characters. Make a build directory with a short name at the root of your harddrive to avoid any issues.
2. **Select compiler.**
    - Press configure to bring up the compiler selection window.
    - Check whether our target languages are installed as 32-bit or 64-bit. For example, if our Python installation is 64-bit, we will need to build the 64-bit version of SimpleElastix to link it. If at all possible, we choose the 64-bit version since the build may run out of memory on 32-bit platforms.
    - Choose a compiler and click next. CMake will find the selected compiler for us if we leave the "Use default native compiler" option checked.

Tip:

- If we need a compiler other than the default system option, we select "Specify native compilers". If we don't know what this means or what we need, we leave the "Use default native compiler" option checked.
- If CMake complains that a compiler cannot be found, we install the free Visual Studio Community Edition .
- If CMake does not pick up our target language, we can set the paths manually. For example, to manually configure CMake Python paths, tick "Advanced" and specify `PYTHON_EXECUTABLE`, `PYTHON_INCLUDE_DIR` and `PYTHON_LIBRARY`. See Troubleshooting section for details.

    - Press generate.
3. **If you are comfortable with the commandline, this is by far the easiest and least error-prone way of compiling the project.**
    - Open x64 native tools command prompt. On Windows 10, open the stat menu and click on "All Apps". Find the Visual Studio folder (e.g. "Microsoft Visual Studio 2012"), open it, and click on "Open VS2012 x64 Native Tools Command Prompt."
    - Navigate to the build folder specified in CMake.
    - Run `msbuild ALL_BUILD.vcxproj /p:Configuration=Release`.
    - Done. Ignore steps 4 and 5.

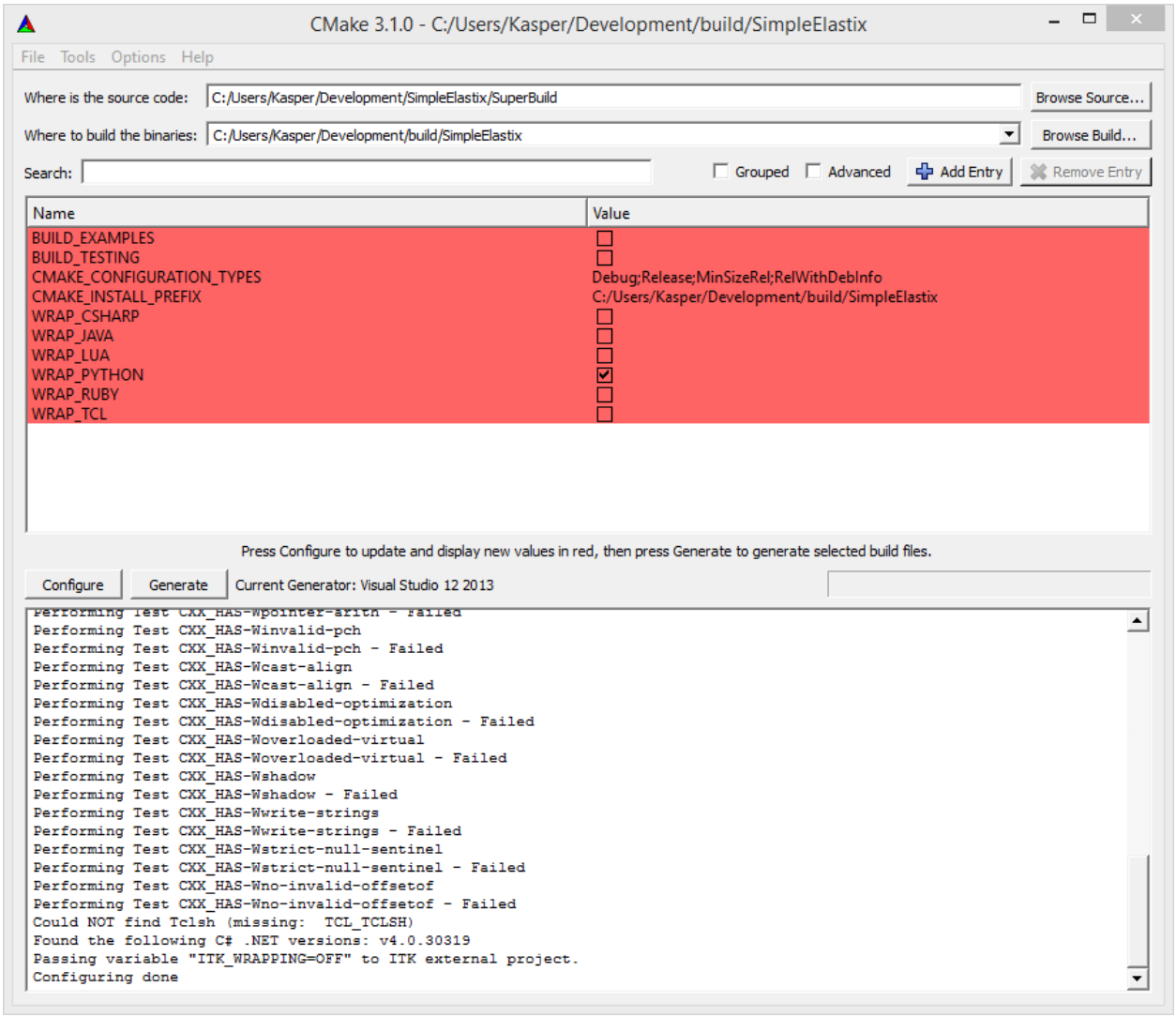


Fig. 1.1: Figure 3: Configure CMake.

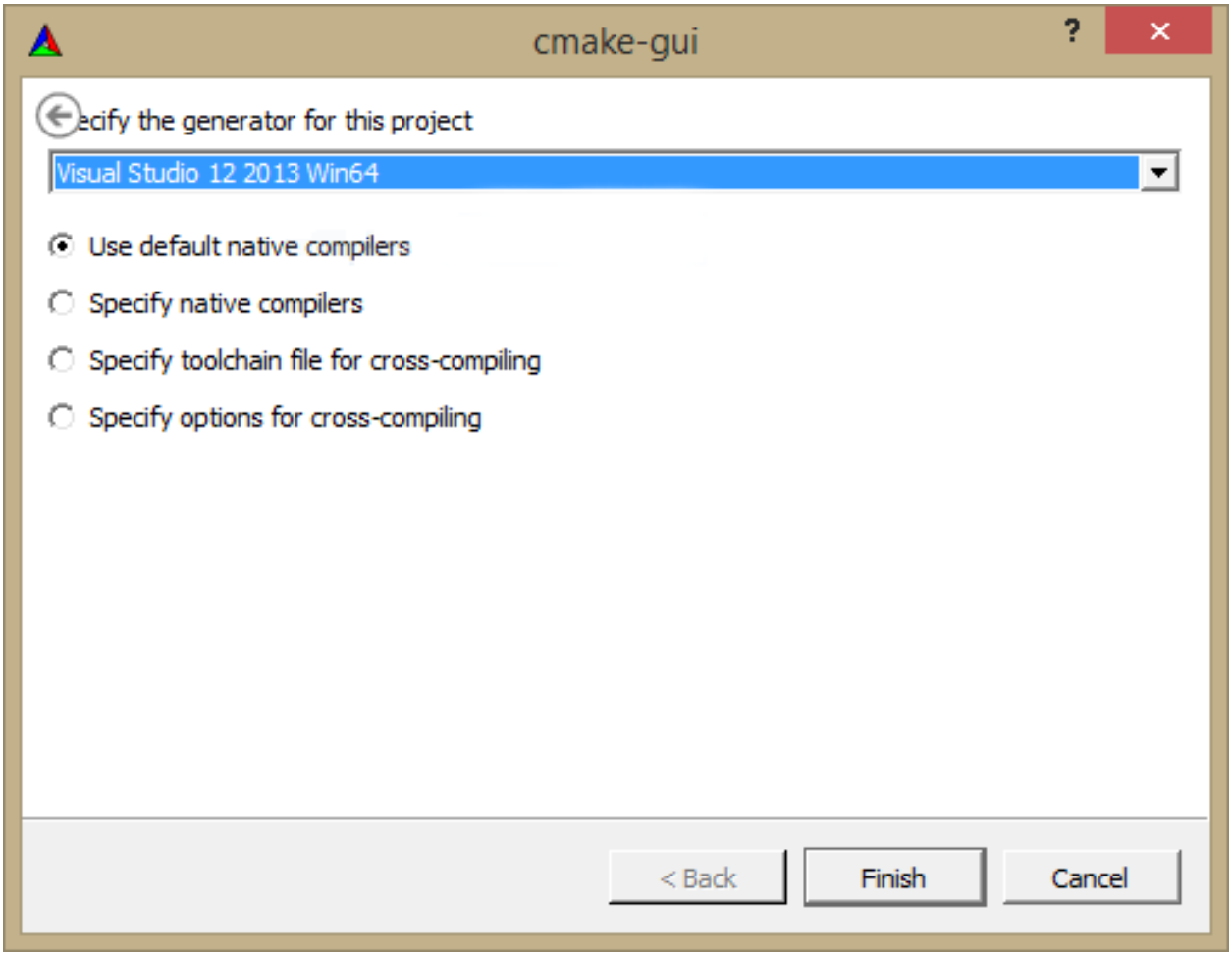


Fig. 1.2: Figure 4: Select compiler.

4. Open Visual Studio, select File -> Open Project/Solution -> Open and choose `SuperBuildSimpleITK` solution.

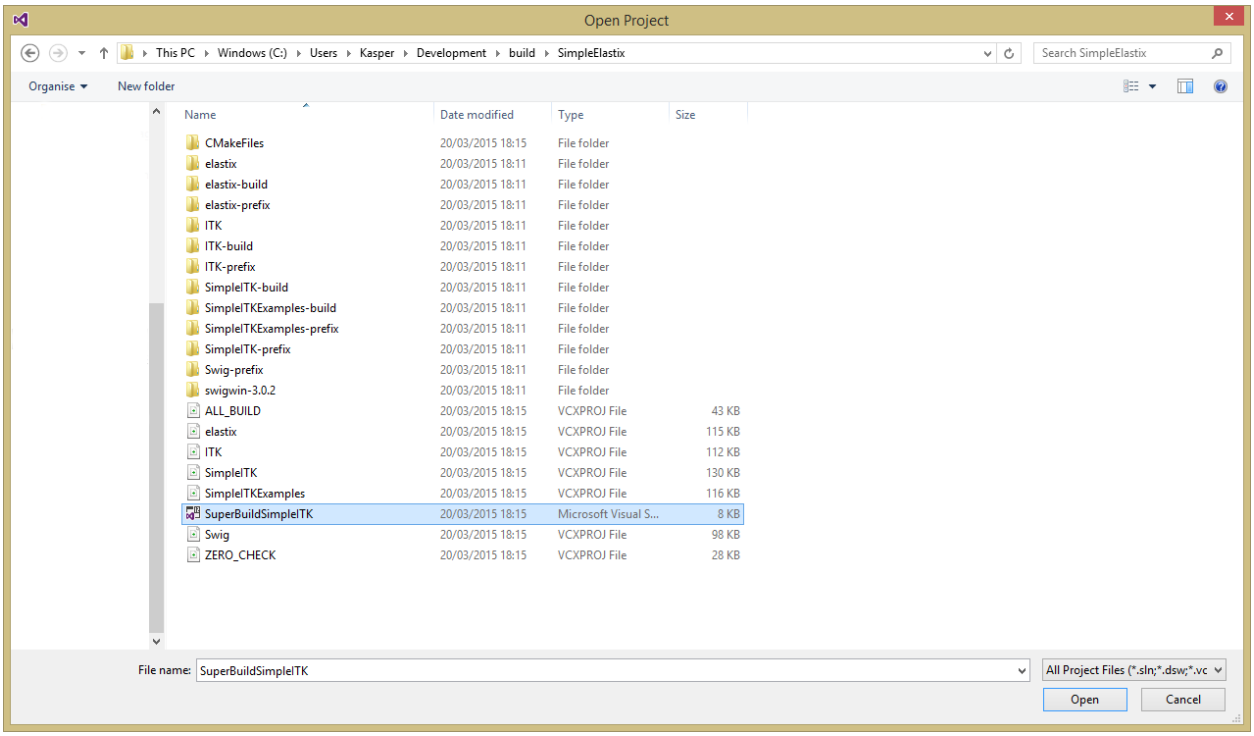


Fig. 1.3: Figure 5: Open the solution in Visual Studio.

5. Make sure "Release" build type is selected and build the `ALL_BUILD` project. If the "Debug" build type is used instead of "Release" mode, we will experience a significant performance penalty and may not be able to build language packages that are distributed without development binaries.

Manually Building On Linux

The following approach allows us to use a locally installed version of ITK and/or elastix.

1. **Setup the prerequisites**
    - *sudo apt-get install cmake swig monodevelop r-base r-base-dev ruby python python-dev tcl tcl-dev tk tk-dev*.
2. Install SWIG >= 3.0.5
3. **Install ITK. Configure CMake using the same approach as above.**
    - Clone ITK from github.com/InsightSoftwareConsortium/ITK.
    - Configure CMake. Set the following CMake variables: BUILD_SHARED_LIBS=OFF, Module_ITKReview=ON, ITK_WRAP_*=OFF.
    - Compile ITK. Make sure to note the build settings, e.g. Release x64.
4. **Build elastix.**

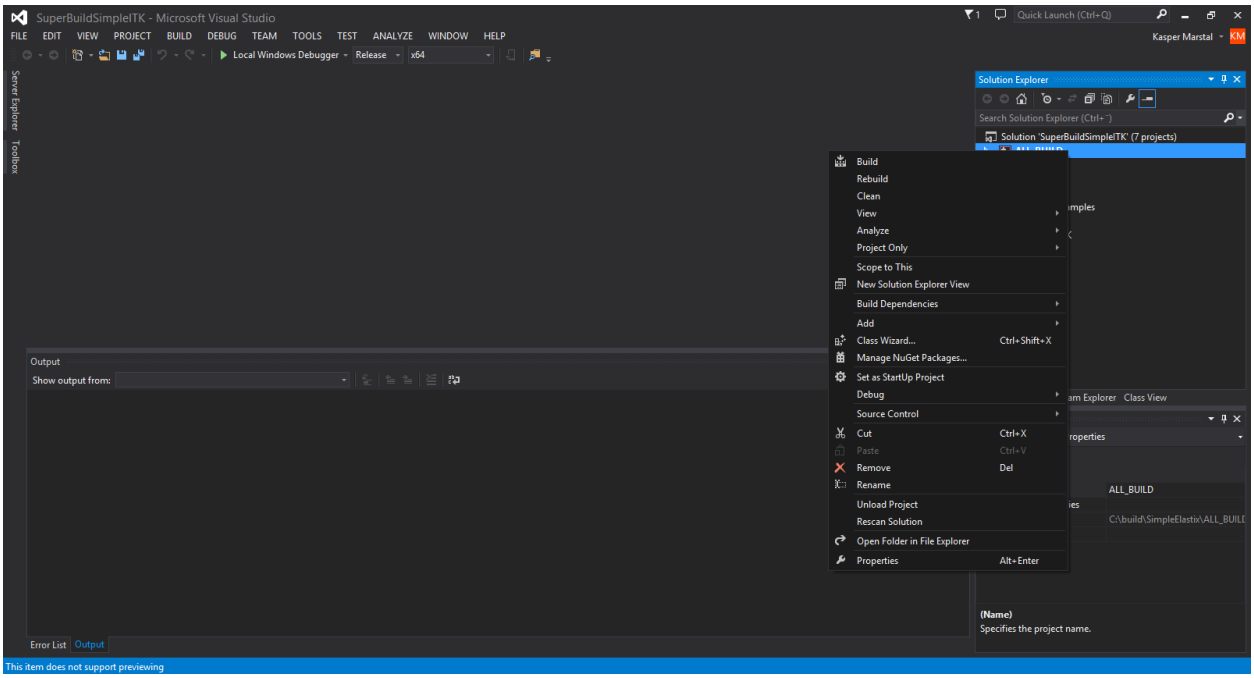


Fig. 1.4: Figure 6: Right-click on `ALL_BUILD` and click `Build`.

- Clone elastix from github.com/kaspermarstal/elastix.
- Set ITK_DIR to the location of the ITK build directory
- Configure CMake. Set the following CMake variables: BUILD_EXECUTABLE=OFF, USE_KNNGraphAlphaMutualInformationMetric=OFF
- Set appropriate ELASTIX_IMAGE_2/3/4D_PIXELTYPES and any components that you might require.
- If you are developing your own elastix components, make sure they are properly registered by the elastix build system.
- Compile elastix. Make sure to configure the build settings exactly the same as ITK e.g. Release x64.

5. **Build SimpleElastix.**

   - Clone SimpleElastix from github.com/kaspermarstal/SimpleElastix.
   - Configure CMake. Point ITK_DIR to the location of the ITK build directory and ELASTIX_DIR to the location of the elastix build directory, specifically the src/ directory in it.
   - Build SimpleElastix. Make sure to configure the build settings exactly the same as ITK e.g. Release x64.

Troubleshooting

I have installed a target language but CMake cannot find it

The language package may be configured incorrectly or the necessary folders may not have been added to your `$PATH` environment variable during installation. Two solutions are available.

- **Solution 1: Add the necessary folders to your `$PATH` environment variable.**
  - Linux and Mac OS X: Add `export PATH=${PATH}:/path/to/folder` to your `$HOME/.bash_profile` (or `$HOME/.profile` depending on your system) and restart the terminal.
  - Windows: Go to Control Panel -> System -> Advanced tab -> Environment Variables and add the target language installation directory to the `PATH` variable.
- **Solution 2: Set the paths manually in CMake (quick and dirty fix). For example, specify `PYTHON_EXECUTABLE`, `PYTHON`**
  - Linux and Mac OS X: Run `$ ccmake .` in the build directory and press `t` on your keyboard to see these options.
  - Windows: Tick "Advanced" in the CMake GUI to see these options.
  - You will have to repeat this procedure every time you setup a new build of SimpleElastix, so we recommend that you configure your `$PATH` environment variable as described in solution 1 above.

If you are still experiencing problems at this point, re-install the language package or consult Google or Stack Overflow.

Visual Studio throws `LNK1102 out of memory` error even though I selected the 64-bit compiler

While Visual Studio targets 64-bit platforms when we select a 64-bit compiler, the Visual Studio toolchain itself will be 32-bit by default. We may therefore experience an out-of-memory error even though you compile a 64-bit vesion of elastix, especially during the linking stage. There are (at least) two ways we can try switch to a 64-bit toolchain ("try" because these methods work, someimes they don't).

- Solution 1: Set the environment variable `_IsNativeEnvironment=true` in command prompt, then call the Visual Studio executable from command line. For example, in the case of VS2013:

```
start "c:\Program Files (x86)\Microsoft Visual Studio 12.
↪0\Common7\IDE\devenv.exe" c:\SimpleElastix\build\SimpleITK-build\SimpleITK.
↪sln
```

- Solution 2: In Visual Studio, edit your .vcxproj file and insert the following after the `<Import... Microsoft.Cpp.Defaults>` line:

```
<Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />
<PropertyGroup>
    <PreferredToolArchitecture>x64</PreferredToolArchitecture>
</PropertyGroup>
```

The SuperBuild throws `Server SSL certificate verification failed: certificate has expired` during checkout of elastix

- Solution 1: We run `svn info https://svn.bigr.nl/elastix/trunkpublic/` in out terminal. It will then prompt us to "(R)eject, accept (t)emporarily or accept (p)ermanently?" the certificate. Hit 'p' and then enter. It may then ask us for our SVN username, which is `elastixguest`, and password, which is also `elastixguest`. Then we restart the build. You may have to change permissions recusively in ~/.subversion/auth. This is done by doing "chmod -R 777 ~/.subversion/auth". If all else fails, our last port of call will be to delete ~/.subversion/auth/svn.ssl.server and then carry out the steps in Solution 1. The commands noted here are linux-specific.

PCRE (Perl Compatible Regular Expression) build fails on Mac OS X

On recent versions of Mac OS X you may experience the following error when using the SuperBuild:

```
Performing build step for 'PCRE'
make[3]: *** No targets specified and no makefile found. Stop.
make[2]: *** [PCRE-prefix/src/PCRE-stamp/PCRE-build] Error 2
make[1]: *** [CMakeFiles/PCRE.dir/all] Error 2
make: *** [all] Error 2
```

This happens during the SWIG build. We can work around this issue by forcing CMake to use clang. Add the following flags to the CMake configure command and run it in a *clean* directory:

```
cmake -DCMAKE_CXX_COMPILER:STRING=/usr/bin/clang++ -DCMAKE_C_COMPILER:STRING=/usr/bin/
↪clang path/to/SimpleElastix/SuperBuild
```

Ruby build fails on Mac OS X

The Ruby virtual machine cannot accomodate spaces in paths. If you see a path that contains spaces like `/Applications/Apple Dev Tools/Xcode.app/Contents/Developer`, re-install Xcode Command Line Tools to a place with no spaces in the path.

I get compilation errors like `elastixlib.h' and 'transformixlib.h' file not found`

This error may stem from a space in the path of the build directory. For example, if we are building SimplElastix in `/Users/kasper/folder name/build` we should rename it to `/Users/kasper/folder_name/build` or similar.

SimpleElastix takes a long time to build!

The full build take 2+ hours to build on a standard machine. We can speed up compilation by deselecting Examples, Testing and any wrapped languages we don't need. Other than that there is not much we can do. SimpleITK has to compile all filters (including elastix) for all pixel types in order to support runtime selection of the correct template parameters.

I am unable to assign a parameter to a parameter map in a parameter map list

This is a known issue which is possibly related to the SWIG-generated code. If you have a solution please make a pull request!

Contribute

If you are experiencing a problem that is not describes on this page, you are very welcome to open an issue on Github and we will do our best to help you out. Likewise, if you have found a solution to a problem that is not described on this page, we hope you will open a pull request on Github and help fix the problem for everyone.

CHAPTER 2

Introduction

If you are completely new to SimpleElastix, or even medical image registration in general, you might not know where to start or what questions to ask. This section will walk you through the basics.

Image Registration

Image registration is the process of transforming images into a common coordinate system so corresponding pixels represent homologous biological points. For example, registration can be used to obtain an anatomically normalized reference frame in which brain regions from different patients can be compared. Computer scientists and medical doctors use this information to build computational models of disease processes.

In the past decade there has been increasing interest in relating information in different medical images spurred by a growing availability of scanners, modalities and computing power. Clinical applications include segmentation of anatomical structures, computer-aided diagnosis, monitoring of disease progression, surgical intervention and treatment planning.

A significant amount of research has focused on developing the registration algorithms themselves. However, less research has focused on accessibility, interoperability and extensibility of these algorithms. Scientific source code is typically not published, is difficult to use because it has not been written with other researchers in mind or is lacking documentation. This is a problem since image registration is a prerequisite for a wide range of medical image analysis tasks and a key algorithmic component for image-based studies. Open source, user-friendly implementations of scientific software make state-of-the-art methods accessible to a wider audience, promote opportunities for scientific advancement, and support the fundamental scientific principle of reproducibility. To this end, we have developed the SimpleElastix software package.

SimpleElastix

Elastix cite{Klein2010} is an open source, command-line program for intensity-based registration of medical images that allows the user to quickly configure, test, and compare different registration methods. SimpleElastix is an extension of SimpleITK cite{Lowekamp2013} that allows you to configure and run Elastix entirely in Python, Java, R, Octave, Ruby, Lua, Tcl and C# on Linux, Mac and Windows. The goal is to bring robust registration algorithms

to a wider audience and make it easier to use elastix, e.g. for Java-based enterprise applications or rapid Python prototyping.

A lot of research has focused on making SimpleElastix computationally efficient and easy to use. Stochastic sampling (Klein et al. 2007), multi-threading and code optimizations (Shamonin et al 2014) makes registration run fast without sacrificing robustness. A simple parameter interface and modular architecture allows you to configure registration components at runtime and easily try out different registration methods.

Previously, performing these kinds of operations on large datasets would incur a significant workflow overhead from scripting command line invocations and arguments to copying images and transform parameter files across folders. With SimpleElastix and SimpleTransformix this complexity is easier to manage and more memory and disk I/O efficient.

Mathematical Background

The registration concept is schematically depicted in Figure 1. A standard image registration procedure involves two input images: One is defined as the fixed image $I_F(x)$ and the other as moving image $I_M(x)$. x denotes the d-dimensional position in the image. If you are registering images taken with your iPhone, $d = 2$ and x will have two components, one for each dimension of the 2D plane. If you are registering body parts taken with a CT scanner, $d = 3$ and x will have three components, one for each of spatial dimension. You may also encounter time series which are four-dimensional (3D+time).

A transform $T(x)$ represents the spatial mapping of points from the fixed image p to points in the moving image q. This establishes a correspondence for every pixel in the fixed image to a position in the moving image.

A similarity metric provides a measure of how well the fixed image matches the moving image. This measure forms a quantitative criterion to be optimized by an optimizer over the search space defined by the parameters of the transform. In general, the registration procedure is formulated as an optimization problem in which a cost function C is minimized with respect to T. Mathematically, $I_M(x)$ is deformed to match $I_F(x)$ by finding a coordinate transformation $T(x)$ that makes $I_M(T(x))$ spatially aligned with $I_F(x)$. This simply means that the optimizer adjusts the parameters of the transform in a way that minimizes the difference between the two images.

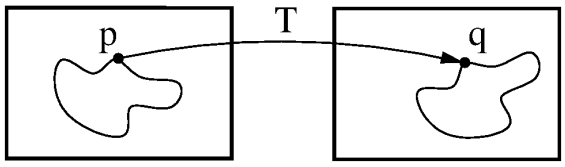


Fig. 2.1: Figure 2: Image registration is the act of deforming moving image points p to biologically corresponding points q in the fixed image domain.

The metric is a key component in the registration process. It uses information from the fixed and moving image to compute a similarity value. The derivative of this value tells us in which direction we should move the moving image for better alignment. The moving image is moved in small steps, and thes process is repeated until a convergence criteria is met. The metric can use pixel intensities, point positions, pre-computed image features or anything we might want to optimize. We just have to define a metric for it.

Sometimes a regularisation term is added to the cost function to penalize unwanted transformations. For example, if we expect your transform to be reasonably smooth, we can penalize sharp deformations. Further, a multi-resolution approach is typically adopted where we start with a high level of smoothing and gradually sharpen the images. This increases the capture range and makes registration much less prone to local minima.

Many different transforms, metrics and optimizers are available in SimpleElastix. In theory, the combinatorial number of choices of registration components can be daunting. In practice, you will develop an intuition about which algorithms are better suited to different types of problems. Choosing the right method for a particular problem will always be a trial-and-error process even for experts, but with the right tools you can focus more on the problem and less on implementation.

Registration Components

In this section we introduce common terminology and some of the choices for different types of components. For a technical discussion and equations see the elastix manual. For documentation of source code see the elastix doxygen pages where you will also find a complete list of available parameters.

Image Pyramids

A multi-resolution pyramid strategy improves the capture range and robustness of the registration. There are three types of pyramids available in elastix: SmoothingImagePyramid, RecursiveImagePyramid and ShrinkingImagePyramid. The FixedImagePyramid and the MovingImagePyramid have identical options. The SmoothingImagePyramid smoothes the images with a Gaussian kernel at different scales. The RecursiveImagePyramid smoothes and downsamples the image. The ShrinkingImagePyramid merely downsamples the image. In general, you should be using the SmoothingImagePyramid together with a random sampler, since it will not throw away valuable information and a few thousead samples is often sufficient for a good approximation to the true gradient. It may consume quite some memory for large images and many resolution levels, however.

Two parameters have to be set to define the multi-resolution strategy: `NumberOfResolutions` and `ImagePyramidSchedule`. The pyramid schedule defines the amount of blurring and/or down-sampling in each direction x, y, z and for each resolution level. If you only set the `NumberOfResolutions`, a default schedule will be used that smoothes the fixed image by a factor of 2 in each dimension, starting from $\sigma = 0.5$ in the highest resolution. This schedule is usually fine. If you have highly anisotropic data, you might want to blur less in the direction of the largest spacing.

In general, 3 resolutions are sufficient. If the fixed and moving image are initially far away, you can increase the number of resolution levels to, say, 5 or 6. This way more attention is paid to register large, dominant structures in the beginning. The resolution schedule is specified as follows:

```
(NumberOfResolutions 4)
(FixedImagePyramidSchedule 8 8 8 4 4 4 2 2 2 1 1 1)
```

In this example, 4 resolutions are used for a 3D image. At resolution level 0 the image is blurred with $\sigma = 8/2$ voxels in each direction (σ is half the pyramid schedule value). At level 1 $\sigma = 4/2$ is used, and finally at level 4 the original images are used for registration.

SimpleElastix can automatically configure these options for you if you use the default parameter maps.

Masks

You may encounter problems where you are more interested in aligning substructures than global anatomy. For example, 4D CT images of lungs vary considerably due to breating motion and you may not want to align the more static rib cage at the expense of lung overlap. One possibility is to crop the image, but this approach restricts the Region Of Interest (ROI) to be a square (2D) or cube (3D) only. If you need an irregular shaped ROI, you can use masks. A mask is a binary image filled with 0's and 1's. Intensity values are only sampled within regions filled with 1's.

You should use a mask when:

- Your image contains an artificial edge that has no real meaning. The registration might be tempted to align these artificial edges, thereby neglecting the meaningful edges. The conic beam edge in ultrasound images is an example of such an artificial edge.
- The image contains structures in the neighbourhood of your ROI that may influence the registration within your ROI as in the lung example.

Masks can be used both for fixed and moving images. A fixed image mask is sufficient to focus the registration on a ROI, since sample positions are drawn from the fixed image. You only want to use a mask for the moving image when your moving image contains highly pertubed grey values near the ROI.

In case of multi-resolution registation you need to set `(ErodeMask "true")` if you do not want information from the artificial edge to flow into you ROI during the smoothing step. If the edge around your ROI is meaningful, as in the lung example where the edges of lungs needs to be aligned, you should set it to false, because the edge will help to guide the registration.

Transforms

The choice of transform is essential for successful registration and, perhaps more importantly, what we perceive as "successful". The transform reflects the desired type of transformation and constrain the solution space to that type of deformation. For example, in intra-subject applications it may be sufficent to consider only rigid transformations if you are registering bones, while a cross-sectional study demands more flexible transformation models to allow for normal anatomical variability between patients.

The number of parameters of the transform corresponds to the degrees of freedom (DOF) of the transformation. This number varies greatly from 3 DOFs for 3D translation and 12 DOFs for 3D affine warping to anywhere between hundreds and millions of DOFs for b-spline deformation fields and non-parametric methods.

The number of DOFs is equal to the dimensionality of the search space and directly proportional to the computational complexity of the optimization problem. The computational complexity affects running time and likelihood of convergence to an optimal solution. Notice that there is a distinction between convergence to an optimal solution and a good registration result. If we use a 2D translation transform embedded in a multi-resolution approach, chances are we will find the global optimal solution. That does not garuantee a good level of anatomical correspondence, however, which will most likely require a more complex deformation model. On the other hand, registering complex anatomical structures using a b-spline deformation without proper initialization is most likely going to fail. Therefore it is often a good idea to start with simple transforms and propagate solutions through transforms of gradually increasing complexity.

Some common transforms are (in order of increasing complexity) translation, rigid (rotation, translation), Euler (rotation, translation), affine (rotation, translation, scaling, shearing), b-spline (non-rigid), Spline-Kernel Transform (non-rigid) and weighted combinations of any of these.

In elastix, the transform is defined from the fixed image to the moving image. It may seem counter-intuitive that the transform is defined in this direction, since it is the moving image we want to transform. Would it not be more logical to map each pixel in the moving image to its new position in fixed image? Perhaps, but then two pixels from the moving image might be mapped to the same pixel on the fixed grid and some pixels in the fixed image might not be mapped to at all. The chosen convention allows us to iterate over the fixed image and pick a pixel from the moving image for every pixel in the fixed image.

Metrics

The similarity metric measures the degree of similarity between the moving and fixed image and is a key component in the registration process. The metric samples intensity values from the fixed and transformed moving image and evaluates the fitness value and derivatives, which are passed to the optimizer.

Selecting an appropriate metric is highly dependent on the registration problem to be solved. For example, some metrics have a large capture range while others require initialization close to the optimal position. In addition, some metrics are only suitable for comparing images obtained from the same imaging modality, while others can handle inter-modality comparisons. There are no clear-cut rules as to how to choose a metric and it may require a trial-and-error process to find the best metric for a given problem.

The Mean Squared Difference (SSD) metric computes the mean squared pixel-wise intensity differences between the fixed and moving images. The optimal value of the metric is zero. Poor matches are result in large values of the metric. The metric samples intensity values from the fixed and transformed moving image and evaluates the fitness value and

derivatives, which are passed to the optimizer. This metric relies on the assumption that intensity representing the same homologous point must be the same in both images and only suited for two images with the same intensity distributions, i.e. for images from the same modality.

Normalized Correlation Coefficient (NCC) computes pixel-wise cross-correlation normalized by the square root of the autocorrelation of the images. The metric is invariant to linear differences between intensity distributions and is therefore particularly well suited for intra-modal CT registration where intensity scales are always related by a linear transform even between scanners. This metric produces a cost function with sharp peaks and well-defined minima, but therefore has a relatively small capture radius.

The Mutual Information (MI) measure computes the mutual information between two images and is more general. MI is a measure of how much information one random variable (image intensity in one image) tells about another random variable (image intensity in the other image). This corresponds to measuring the dependency of the probability density distributions (PDF) of the intensities of the fixed and moving images without having to specify the actual form of the dependency. It is therefore well suited for multi-modal image pairs as well as mono-modal images. Normalized Mutual Information is likewise suitable for both mono- and multi-modality registration. Some studies seem to indicate better performance with NMI than MI in some cases.

Mattes Mutual Information is an implementation where the same pixels are sampled in every iteration. Using a fixed set of discrete positions to evaluate the marginal and joint PDFs makes the path in search space more smooth.

The Kappa Similarity metric measures the overlap of segmented structures and is developed specifically to register binary images (segmentations). In most cases however, it is better to convert the binary images to a distance map and apply one of the usual metrics.

Optimizers

The optimizer estimates the optimal transform parameters in iterative fashion. Many optimizers are available in elastix including Gradient Descent (GD), Robbins-Monroe (RM), Adaptive Stochastic Gradient Descent (ASGD), Conjugate Gradient (CG), Conjugate Gradient FRPR, Quasi-Newton LBFGS, RSGD "Each parameter apart", Simultaneous Perturbation (SP), CMAEvolutionStrategy and FullSearch which samples the entire search space. If you do not have any immediate preferences, stick to the Adaptive Stochastic Gradient Descent (Klein 2009) which requires less parameters to be set and tends to be more robust. For an elabore discussion see the elastix manual.

Samplers

The sampler is responsible for selecting locations in input images for the metric to evaluate. In general, it is sufficient to evaluate only a subset of randomly sampled voxels. SimpleElastix comes with a grid, random, random coordinate and full sampling strategies selected using the `ImageSampler` parameter.

The grid sampler defines a regular grid on the fixed image and selects the coordinates onthe grid. Effectively, the image is down-sampledwithout smoothing. The size of the grid or downsampling factor is taken as input. The random sampler randomly selects a user-specified number of voxels. Every voxel has equal chance to be selected and a sample may be selected more than once. The random coordinate sampler is not limited to voxel positions but may sample positions between voxels. The off-grid grey-level values are obtained via in interpolation.

Interpolators

The metric typically compares intensity values in the fixed image against the corresponding values in the transformed moving image. When a point is mapped from one space to another by a transform, it will generally be mapped to a non-grid position. Interpolation is required to evaluate the image intensity at the mapped off-grid position.

Several methods for interpolation exist, varying in quality and speed. The `NearestNeighborInterpolator` returns the value of the spatially closest voxel. This technique is low in quality but require little resources. You will want to use this method for binary images.

The `LinearInterpolator` returns a weighted average of surrounding voxels using distances as weights. In elastix, this method is highly optimized and very fast. In general, you will probably want to use this method together with the random coordinate sampler during the optimization process for best performance (in the time sense).

The `BSplineInterpolator` (or the more memory effecient `BSplineInterpolatorFloat`) interpolates pixel values using b-spline approximations of user-defined order N. First order b-splines corresponds to linear intepolation in which case you might as well use the linear interpolator. To generate the final result image a higher-order interpolation is usually required for which $N = 3$ is recommended. The final interpolator is called a ResampleInterpolator. Any one of the above methods can be used, but you need to prepend the name with Final, for example `FinalBSplineInterpolatorFloat`

Images

As a final note in this section, it is important to know the appropriate definitions and terms when working with medical images. In particular, information associated with physical spacing between pixels and position on the image grid with respect to world coordinate system is extremely important. Improperly defined spacing and origins will most likely result in inconsistent results. The main geometrical concepts associated with an image object are depicted in Figure 7.

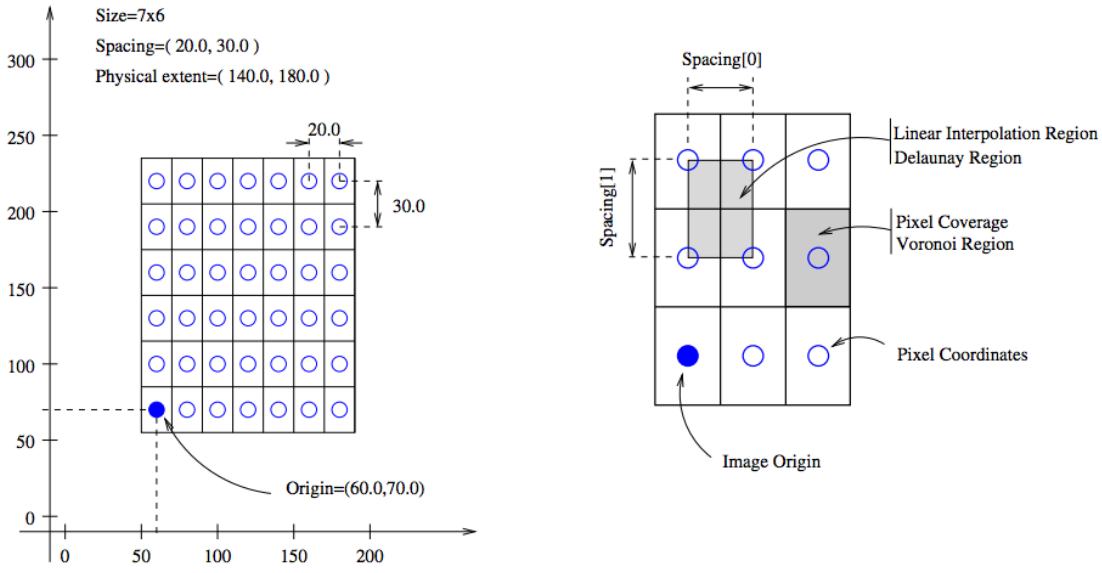


Fig. 2.2: Figure 7: Geometrical concepts associated with the ITK image. Adopted from Ibanez et al. (2005).

Above, circles are used to represent the centre of pixels. The value of the pixel is assumed to be a Dirac Delta Function located at the pixel centre. Pixel spacing is measured between the pixel centres and can be different along each dimension. The image origin is associated with the coordinates of the first pixel in the image. A pixel is considered to be the rectangular region surrounding the pixel centre holding the data value.

In the next section we introduce the SimpleElastix Hello World example.

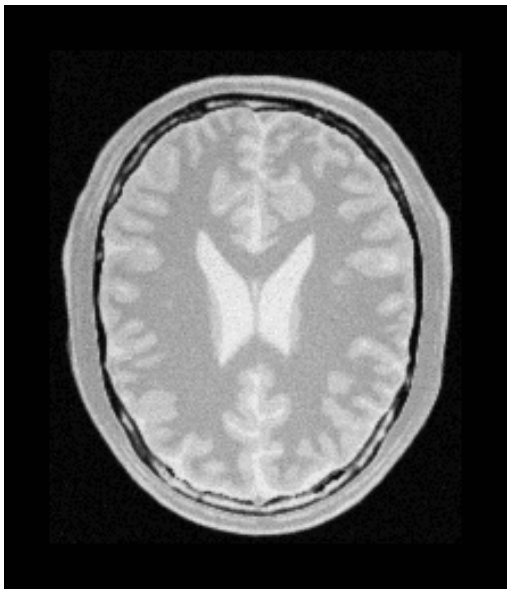
CHAPTER 3

Hello World

This example illustrates how to use SimpleElastix. With a single function call we can specify the fixed image, the moving image and the type of registration you want to perform. SimpleElastix will then register our images using sensible default parameters that work well in most cases. In later examples we shall see how to tweak the default parameters, but for now we keep it simple.

Registration With Translation Transform

Say we want to register the following two brain MRIs.



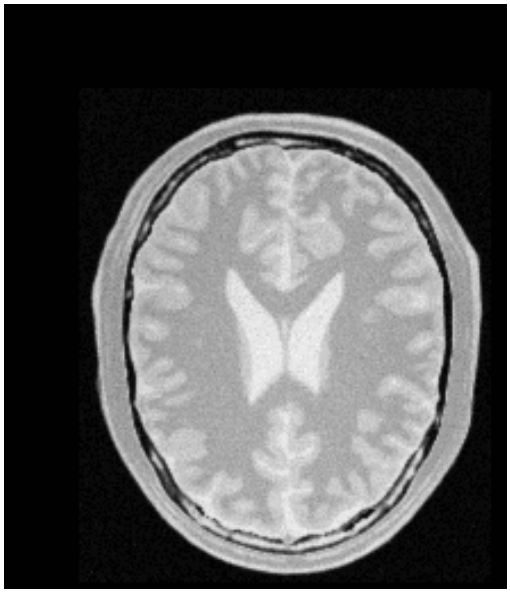


Figure 2. The original image `fixedImage.nii` (left) and translated image `movingImage.nii` (right).

We identify that the objects are related by a simple spatial shift and that a translation transform should be sufficient to align the objects. To correct the misalignment, we run the following lines of python code:

```
import SimpleITK as sitk

resultImage = sitk.Elastix(sitk.ReadImage("fixedImage.nii"), \
                           sitk.ReadImage("movingImage.nii"), \
                           "translation")
```

That's it! We have effectively registered two images using a single line of code. Compare this to the ITK Hello World example. We refer to this short-hand notation as the procedural interface (some people also refer to it as the functional interface) because it consists of functions that accomplish a specific task. The procedural interface is less flexible than the object-oriented interface introduced below, but it is very simple to use. Let's break down what goes on under the hood of this single function call.

First of all, `import SimpleITK as sitk` loads the SimpleITK module from which SimpleElastix is accessed. This assumes that SimpleElastix has been compiled and installed on your machine.

`sitk.ReadImage()` is the generic image file reader of SimpleITK which loads an image from disk and pass a SimpleITK image object to SimpleElastix. You can also apply a SimpleITK filter to an image before passing them to SimpleElastix. For example, you could use the `ResampleImageFilter()` to downsample images that would otherwise consume too much memory during registration.

The final parameter `"translation"` specifies the desired type of registration. In the Parameter Maps section we will take a close look at parameter maps and examine what happens when you specify this parameter. It is obvious from the figure below that a translation transform is sufficient to align these images.

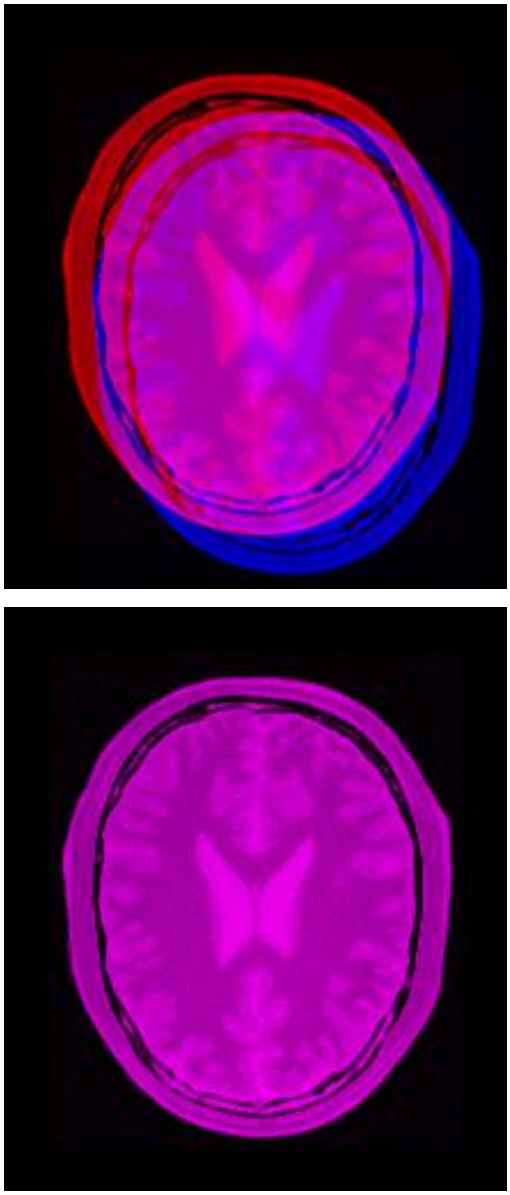


Figure 3. Mean image before registration (left) and mean image after registration (right). The fixed image is shown the red color channel and the moving image is shown in the blue color channel.

Object-Oriented Interface

The example above used procedural interface. While the procedural interface may be useful for rapid prototyping, it trades off flexibility for code simplicity. For example, the final deformation field cannot be retrived and applied to another image. This is a problem if we want to subsequently warp other images, e.g. a label image, using the same transformation. Further, image quality is reduced from resampling the resulting image twice. To this end, SimpleElastix comes with a more flexible object-oriented interface suitable for more advanced use cases and scripting purposes. In the next example, we perform the same registration as above, but this time using the object oriented interface:

```
import SimpleITK as sitk
```

```
fixedImage = sitk.ReadImage('fixedImage.nii')
movingImage = sitk.ReadImage('movingImage.nii')
parameterMap = sitk.GetDefaultParameterMap('translation')

elastixImageFilter = sitk.ElastixImageFilter()
elastixImageFilter.SetFixedImage(fixedImage)
elastixImageFilter.SetMovingImage(movingImage)
elastixImageFilter.SetParameterMap(parameterMap)
elastixImageFilter.Execute()

resultImage = elastixImageFilter.GetResultImage()
transformParameterMap = elastixImageFilter.GetTransformParameterMap()
```

This is more verbose but also a lot more powerful. We can now warp an entire population of images (e.g. binary label images for segmentation of different brain regions) using the same parameter map and a single instance of transformix:

```
transformixImageFilter = sitk.TransformixImageFilter()
transformixImageFilter.SetTransformParameterMap(transformParameterMap)

population = ['image1.hdr', 'image2.hdr', ... , 'imageN.hdr']

for filename in population:
    transformixImageFilter.SetMovingImage(sitk.ReadImage(filename))
    transformixImageFilter.Execute()
    sitk.WriteImage(transformixImageFilter.GetResultImage(), "result_"+filename)
```

The object-oriented interface facilitates reuse of components and dramatically simplifies book-keeping and boilerplate code. We will use the object-oriented interface in the documentation from this point forward.

In the next section, we will take a closer look at the parameter map interface that configures the registration components.

CHAPTER 4

Parameter Maps

In the previous section we saw how to configure an entire multi-resolution registration process with a single parameter. Here, we will examine how SimpleElastix configures registration components internally. Knowing the internal mechanisms is not strictly necessary to register simple anatomical structures, but it will help you solve complex problems that require problem-specific tuning.

Elastix introduces the concept of a parameter map to configure the registration procedure. A parameter map is a collection of key-value pairs that atomically defines the components of the registration and any settings they might require. Only input images and output options need to be specified seperately. Elastix will read a given parameter map and load the specified components at runtime.

The original elastix and transformix command line programs read text files from disk in which parameters are specified according to the following format:

```
(ParameterName ParameterValue)
```

The parameter may be either single- or vector-valued. If the value is of type `string` then the value will be quoted:

```
(ParameterName "value1" ... "valueN")
```

If the value is of numerical type it will be unquoted:

```
(ParameterName 123)
```

SimpleElastix can read these types of files as well but further introduces native data objects for parameter files in all target languages. This means that instead of editing text files you can programmatically configure registration components. For example, we could initialize a parameter object and start populating it with values:

```
import SimpleITK as sitk

p = sitk.ParameterMap()
p['Registration'] = ['MultiResolutionRegistration']
p['Transform'] = ['TranslationTransform']
...
```

and so on. ElastixImageFilter has default settings that allows us to get started right away.

The Default Parameter Maps

In the Hello World example we obtained a registered image by running

```
resultImage = sitk.Elastix(sitk.ReadImage('fixedImage.nii'),  \
                           sitk.ReadImage('movingImage.nii'), \
                           'translation')
```

Internally, ElastixImageFilter uses a pre-configured parameter map to register images. The parameter map is obtained by an internal call to `sitk.GetDefaultParameterFile('translation')`. This function provides parameter maps for rigid, affine, non-rigid and groupwise registration methods (in order of increasing complexity).

Tip: ElastixImageFilter will register our images with a `translation -> affine -> b-spline` multi-resolution approach by default. We simply leave out the call to `SetParameterMap` to achieve this functionality.

```
import SimpleITK as sitk

# Functional interface
resultImage = sitk.Elastix(sitk.ReadImage('fixedImage.nii'), sitk.ReadImage(
↪'movingImage.nii'))

# Object oritented interface
elastixImageFilter = sitk.ElastixImageFilter()
elastixImageFilter.SetFixedImage(sitk.ReadImage('fixedImage.nii'))
elastixImageFilter.SetMovingImage(sitk.ReadImage('movingImage.nii'))
resultImage = elastixImageFilter.Execute()
```

We can also retrieve the parameter map ourselves and reconfigure it before passing it back to ElastixImageFilter, allowing us to quickly optimize a registration method for a particular problem:

```
parameterMap = sitk.GetDefaultParameterMap('translation')

# Use a non-rigid transform instead of a translation transform
parameterMap['Transform'] = ['BSplineTransform']

# Because of the increased complexity of the b-spline transform,
# it is a good idea to run the registration a little longer to
# ensure convergence
parameterMap['MaximumNumberOfIterations'] = ['512']

resultImage = sitk.Elastix(sitk.ReadImage('fixedImage.nii'),  \
                           sitk.ReadImage('movingImage.nii'), \
                           parameterMap)
```

Tip: We can print parameter maps to console like this:

```
import SimpleITK as sitk
elastixImageFilter = sitk.ElastixImageFilter()
elastixImageFilter.PrintParameterMap()
```

We will study other parameter maps more closely in later examples. For now, we simply print the translation parameter map to console and examine its contents.

```
>>> sitk.PrintParameterMap(sitk.GetDefaultParameterMap("translation"))
ParameterMap 0:
  (AutomaticParameterEstimation "true")
  (CheckNumberOfSamples "true")
  (DefaultPixelValue 0)
  (FinalBSplineInterpolationOrder 2)
  (FixedImagePyramid "FixedSmoothingImagePyramid")
  (FixedImagePyramidSchedule 8 8 8 4 4 4 2 2 2 1 1 1)
  (ImageSampler "RandomCoordinate")
  (Interpolator "LinearInterpolator")
  (MaximumNumberOfIterations 32)
  (MaximumNumberOfSamplingAttempts 8)
  (Metric "AdvancedMattesMutualInformation")
  (MovingImagePyramid "MovingSmoothingImagePyramid")
  (MovingImagePyramidSchedule 8 8 8 4 4 4 2 2 2 1 1 1)
  (NewSamplesEveryIteration "true")
  (NumberOfResolutions 4)
  (NumberOfSamplesForExactGradient 4096)
  (NumberOfSpatialSamples 4096)
  (Optimizer "AdaptiveStochasticGradientDescent")
  (Registration "MultiResolutionRegistration")
  (ResampleInterpolator "FinalBSplineInterpolator")
  (Resampler "DefaultResampler")
  (Transform "TranslationTransform")
  (WriteResultImage "true")
```

The first thing to note is that the parameter map is enumerated. ElastixImageFilter can take a vector of parameter maps and apply the corresponding registrations sequentially. The resulting transform is called a composite transform since the final transformation is a composition of sequentially applied deformation fields. For example, a non-rigid registration is often initialized with an affine transformation (translation, scale, rotation, shearing) to bring the objects into rough alignment. This makes the registration less suscetible to local minima. We can also ask SimpleElastix to add the individual deformation fields and apply them in one go (but make sure you know what you are doing before opting for this apprach).

Tip: We can add mulitple parameter maps to SimpleElastix like this:

```
import SimpleITK as sitk
elastixImageFilter = sitk.ElastixImageFilter()
elastixImageFilter.SetParameterMap(sitk.GetDefaultParameterMap('translation'))
elastixImageFilter.AddParameterMap(sitk.GetDefaultParameterMap('affine'))
```

Note that the first call is a `Set` method. This deletes any prevously set parameter maps. Subsequent calls to `AddParameterMap` appends parameter maps to the internal list of parameter maps

---

Let's examine the parameters above in detail.

Important Parameters

`Registration` is the top-level parameter which in this case has been set to `MultiResolutionRegistration`. A multi-resolution pyramid strategy improves the capture range and robustness of the registration. We will almost always want to use multiple resolutions unless your problem is particularly simple. The basic idea is to first estimate `T(x)` on a low resolution version of the images and then propagate the estimated deformation to higher resolutions. This makes the registration initially focus on larger structures (the

skull and brain hemispheres etc), before focusing on high-frequency information (brain subregions etc) which contain more local minima. `FixedImagePyramid`, `FixedImagePyramidSchedule`, `MovingImagePyramid`, `MovingImagePyramidSchedule`, and `NumberOfResolutions` controls the pyramid strategy.

The `Transform` parameter is set to `TranslationTransform` which it is optimized with an `AdaptiveStochasticGradientDescent` optimizer (Klein et al. 2009). SimpleElastix will use this optimizer together with the `AdvancedMattesMutualInformation` metric by default since this combination work well for a broad range of problems whether mono-modal or multi-modal.

Image intensities are sampled using an `ImageSampler`, `Interpolator` and `ResampleInterpolator`. The sampler is responsible for selecting points in the image to sample. The `RandomCoordinate` simply selects random positions. The interpolator is responsible for interpolating off-grid posititions during optimization. The `LinearInterpolator` used here is very fast and uses very little memory.

A `BSplineInterpolator` of order 2 is used to resample the result image from the moving image once the final transformation has been found. This is a one-time step so the additional computational complexity is worth the trade-off for higher image quality.

Another important parameter is `AutomaticParameterEstimation` which controls whether the `AdaptiveStochasticGradientDescent` optimizer should estimate its own convergence parameters or allow you to set them. Automatically obtained parameters work well in most cases and facilitates a complete hands-off approach which is highly recommended. Optimizers can be tricky to tune by hand.

`DefaultPixelValue` sets value of pixels outside the moving image grid. The rest of the key-value pairs are component specific parameters. There are multiple choices available for each type of component. For example, you can construct an image pyramid with recursive sampling or via Gaussian Smoothing. Each choice has its own pros and cons. Consult the Registration Components secton for a description of all types of available components.

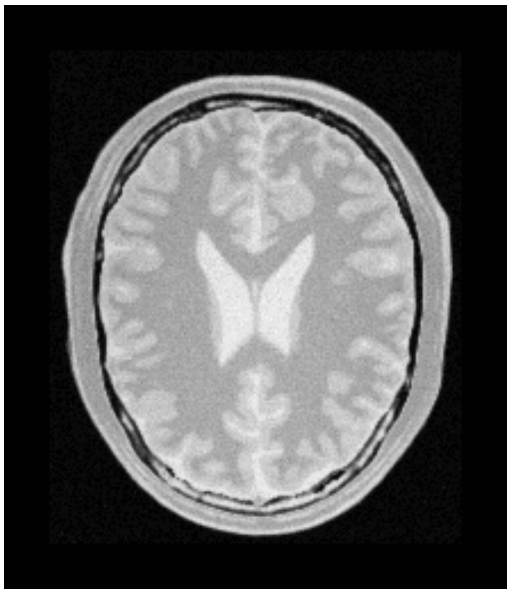
CHAPTER 5

Rigid Registration

A rigid transform can register objects that are related by rotation and translation. For example, if you are registering images of a patient's bones, you can often assume that a rigid transform is sufficient to align these structures. In fact, it is often advantageus to chose a simple transform if problems that allows it, as this constrains the solution space and ensures no spurious non-rigid local minima affect your results. Think of it as a way of embedding expert knowledge in the registration procedure.

Tip: Rigid registration is one of the simplest of methods in the catagory of linear transformation models and is often used as initialization for affine- and non-rigid transforms.

The rigid transform is selected using `(Transform "EulerTransform")`. Consider the images in Figure 8.



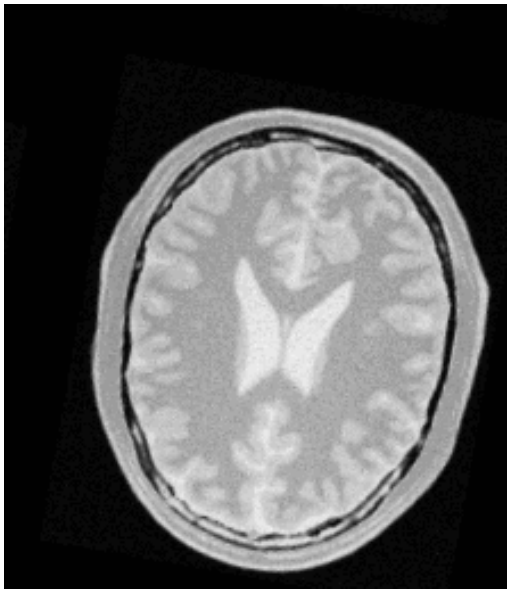


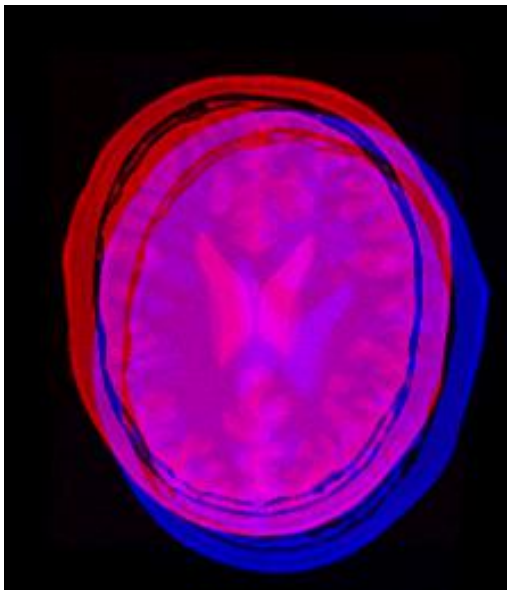
Figure 8. The original image `fixedImage.nii` (left) and translated and rotated image `movingImage.nii` (right).

The image on right has been rotated 10 degrees and translated 13 pixels in the x-direction and 17 pixels in the y-direction. Using the `EulerTransform` we may correct for this misalignment.

```
import SimpleITK as sitk

elastixImageFilter = sitk.ElastixImageFilter()
elastixImageFilter.SetFixedImage(sitk.SetFixedImage("fixedImage.nii"))
elastixImageFilter.SetMovingImage(sitk.SetFixedImage("movingImage.nii"))
elastixImageFilter.SetParameterMap(sitk.GetDefaultParameterMap("rigid"))
elastixImageFilter.Execute()
sitk.WriteImage(elastixImageFilter.GetResultImage())
```

It is clear from the result mean image on right in Fig. 9 that registration was successful.



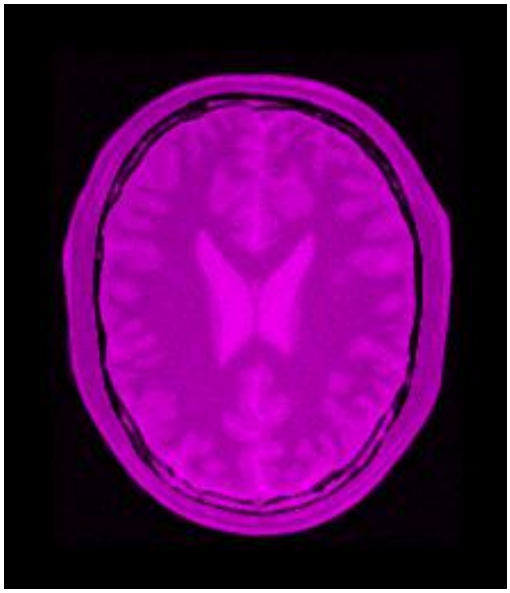


Figure 9. Mean image before registration (left) and mean image after registration (right).

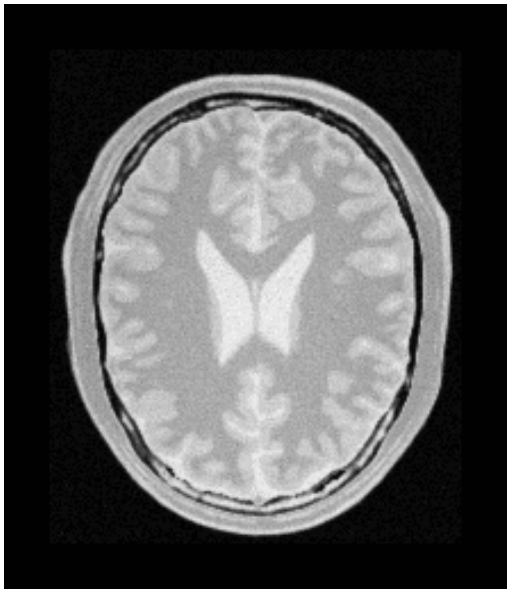
In the next example we will introduce scaling and shearing into the registration.

CHAPTER 6

Affine Registration

The affine transform allows for shearing and scaling in addition to the rotation and translation. This is usually a good choice of transform for initialization of non-rigid transforms like the B-Spline transform. The affine transform is selected using `sitk.GetDefaultParameterMap("affine")`.

Consider the images in Figure 10.



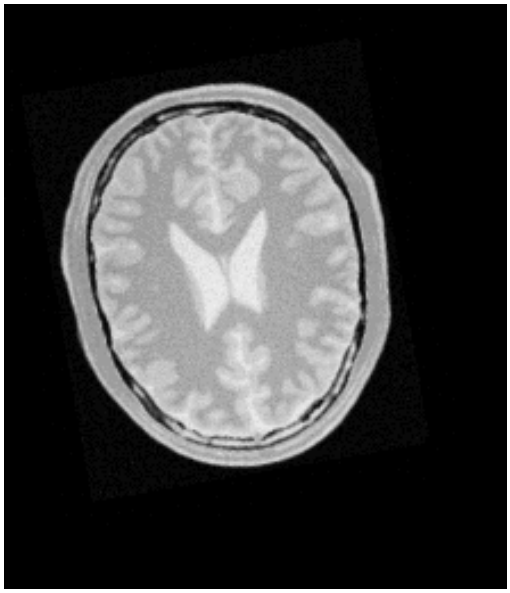


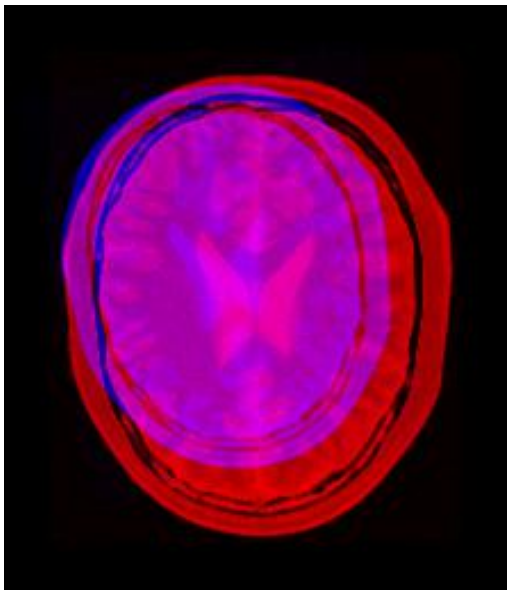
Figure 10. The original image `fixedImage.nii` (left) and translated and rotated image `movingImage.nii` (right).

The image on the right has been sheared, scaled 1.2x, rotated 10 degrees and translated 13 pixels in the x-direction and 17 pixels in the y-direction. Using the `AdvancedAffineTransform` we may correct for this misalignment.

```
import SimpleITK as sitk

elastixImageFilter = sitk.ElastixImageFilter()
elastixImageFilter.SetFixedImage(sitk.ReadImage("fixedImage.nii")
elastixImageFilter.SetMovingImage(sitk.ReadImage(movingImage.nii")
elastixImageFilter.SetParameterMap(sitk.GetDefaultParameterMap("affine"))
elastixImageFilter.Execute()
sitk.WriteImage(elastixImageFilter.GetResultImage())
```

It is clear from the result mean image on right in Fig. 11 that registration was successful.



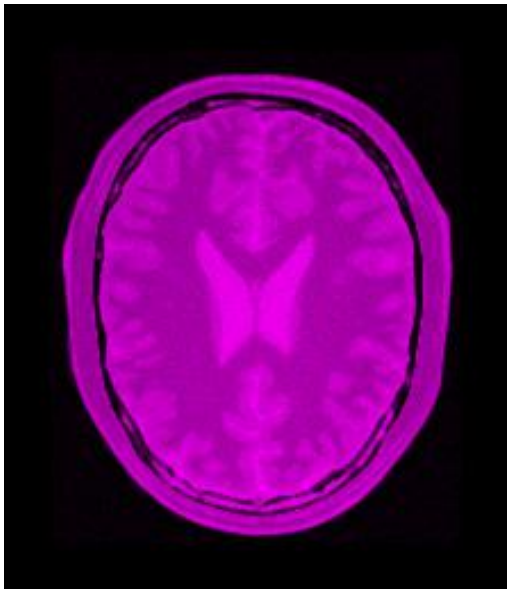


Figure 11. Mean image before registration (left) and mean image after registration (right).

Notice that the only difference from the previous example is the requested parameter map. In fact, only the *Transform* parameter seperates the two parameter maps. The following parameter map is equivalent to the one used above:

```
import SimpleITK as sitk

parameterMap = sitk.GetDefaultParameterMap("rigid")
parameterMap["Transform"] = ["AffineTransform"]
```

You can inspect the default parameter maps in the **'elastix repository <https://github.com/mstaring/elastix/blob/617b0729fb6200fce279f7e6388967c6315ddc90/src/Core/Main/elxParameterObject.cx L362'_** to convince yourself.

This demonstrates how easy it is to try out different registration components with SimpleElastix. In the next example we will introduce non-rigid registration and initialize the moving image with an affine transform.

CHAPTER 7

Non-rigid Registration

Non-rigid registration methods are capable of aligning images where correspondence cannot be achieved without localized deformations and can therefore better accomodate anatomical, physiological and pathological variability between patients.

B-splines are often used to parameterize a free-form deformation (FFD) field. This is a much harder registration problem than any of the previous examples due to a much higher-dimensional parameter space and we are therefore best off using a multi-resolution approach with affine initialization. This is very easy to do in SimpleElastix.

Consider the following mean image of two different subjects.

The following code runs multi-resolution affine initialization and starts a non-rigid method multi-resolution non-rigid method using the affine transform as initialization :

```
import SimpleITK as sitk

elastixImageFilter = sitk.ElastixImageFilter()
elastixImageFilter.SetFixedImage(sitk.ReadImage("fixedImage.nii"))
elastixImageFilter.SetMovingImage(sitk.ReadImage("movingImage.nii"))

parameterMapVector = sitk.VectorOfParameterMap()
parameterMapVector.append(sitk.GetDefaultParameterMap("affine"))
parameterMapVector.append(sitk.GetDefaultParameterMap("bspline"))
elastixImageFilter.SetParameterMap(parameterMapVector)

elastixImageFilter.Execute()
sitk.WriteImage(elastixImageFilter.GetResultImage())
```

The result image is seen below.

In this case, we are able to compensate for many non-rigid differences between the two images. Note, however, that brain image registration is a difficult to task because of complex anatomical variations. Entire registration packages are dedicated to brain image processing. You might want to consider a more refined approach in critical applications.

In the next section we introduce groupwise registration, where many images are registered simultaneously a mean frame of reference.

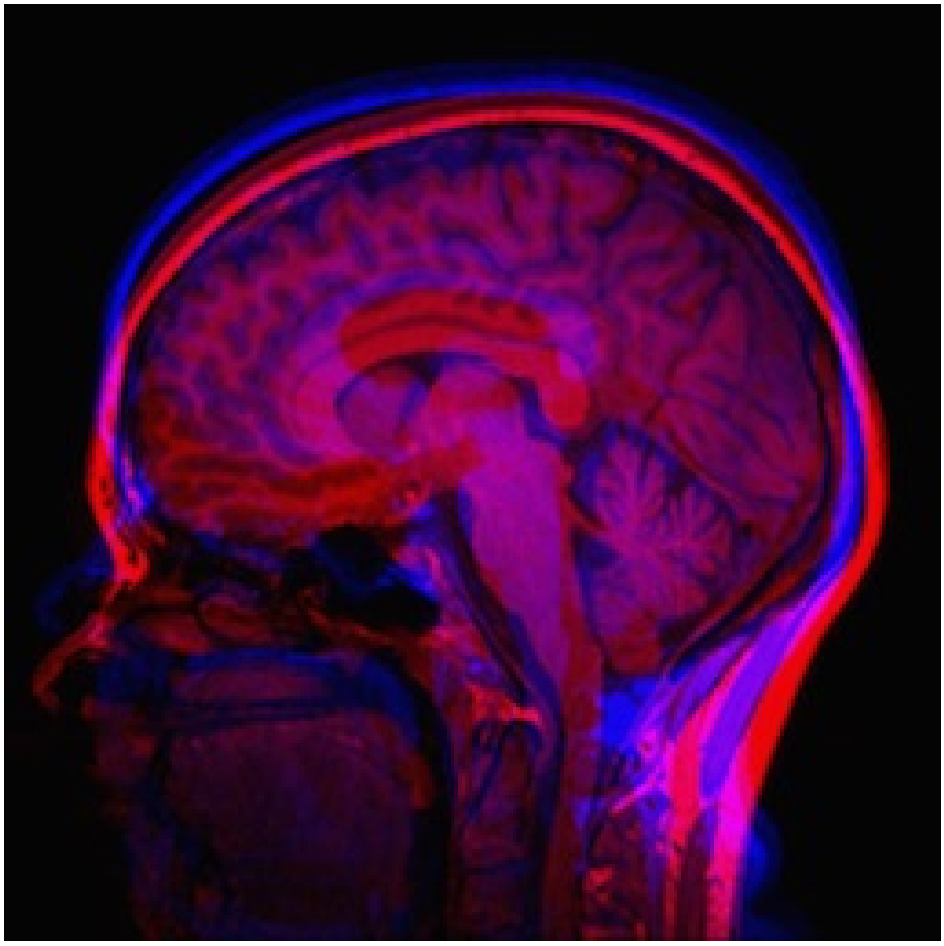


Fig. 7.1: Figure 13: Mean original image.

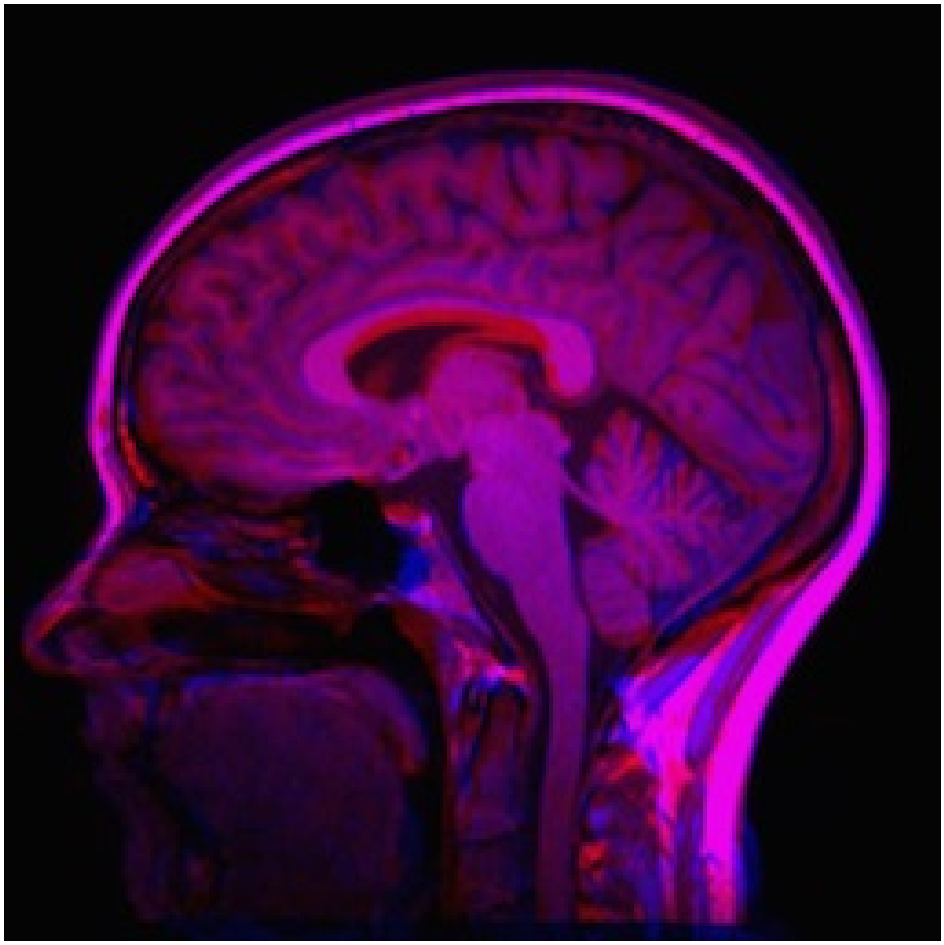


Fig. 7.2: Figure 14: Mean result image.

CHAPTER 8

Groupwise Registration

Groupwise registration methods try to mitigate uncertainties associated with any one image by simultaneously registering all images in a population. This incorporates all image information in registration process and eliminates bias towards a chosen reference frame. The method described here uses a 3D (2D+time) and 4D (3D+time) free-form B-spline deformation model and a similarity metric that minimizes variance of intensities under the constraint that the average deformation over images is zero. This constraint defines a true mean frame of reference that lie in the center of the population without having to calculate it explicitly.

The method can take into account temporal smoothness of the deformations and a cyclic transform in the time dimension. This may be appropriate if it is known a priori that the anatomical motion has a cyclic nature e.g. in cases of cardiac or respiratory motion.

Note that brain registration is a difficult to task because of complex anatomical variations and almost a scientific topic in itself. Entire registration packages are dedicated to just brain image processing. In this section we are less strict with the end result and focus on illustrating the groupwise registration method in SimpleElastix.

Consider the following mean image:

Elastix takes a single N+1 dimensional image for groupwise registration. Therefore we need to first concatenate the images along the higher dimension. SimpleITK makes this very easy with the `JoinSeries` image filter. The registration step is business as usual:

```
import SimpleITK as sitk

# Concatenate the ND images into one (N+1)D image
population = ['image1.hdr', ..., 'imageN.hdr']
vectorOfImages = sitk.VectorOfImage()

for filename in population
  vectorOfImages.push_back(sitk.ReadImage(filename))

image = sitk.JoinSeries(vectorOfImages)

# Register
elastixImageFilter = sitk.ElastixImageFilter()
elastixImageFilter.SetFixedImage(image)
```

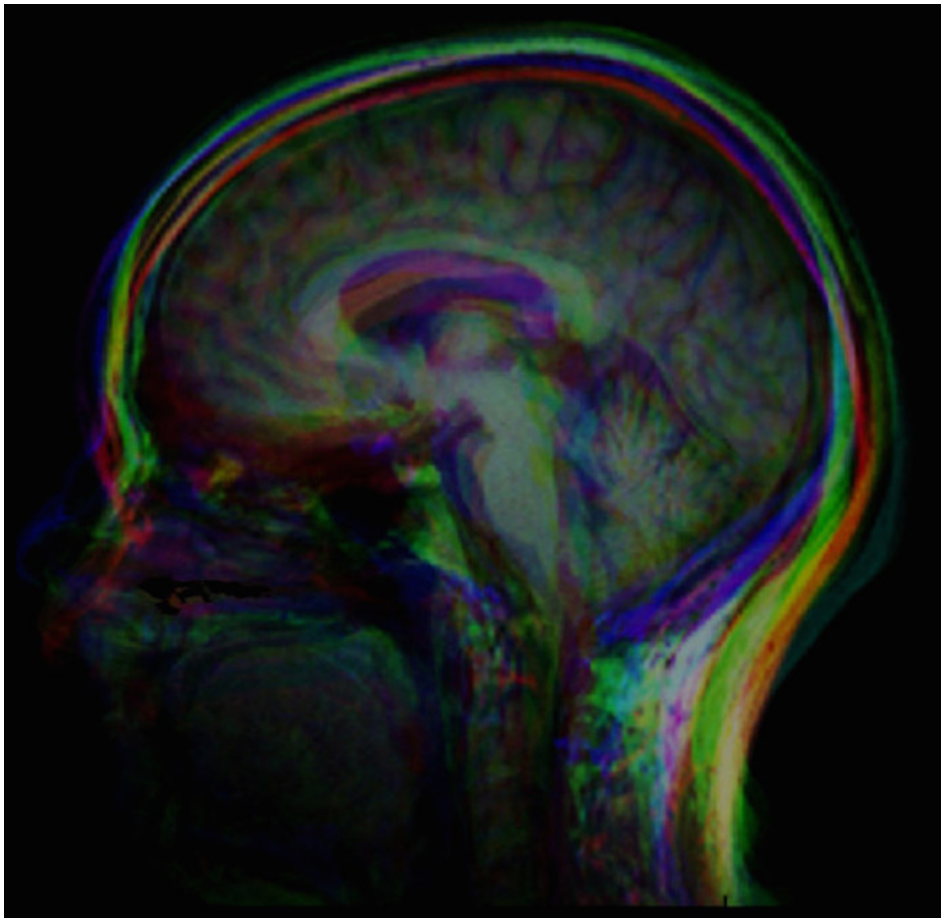


Fig. 8.1: Figure 12: Mean original image.

```
elastixImageFilter.SetMovingImage(image)
elastixImageFilter.SetParameterMap(sitk.GetDefaultParameterMap('groupwise'))
elastixImageFilter.Execute()
```

While the groupwise transform works only on the moving image we need to pass a dummy fixed image is to prevent elastix from throwing errors. This does not consume extra memory as only pointers are passed internally.

The result image is shown in Figure 13. It is clear that anatomical correpondence is obtained in many regions of the brain. However, there are a some anatomical regions that have not been registered correctly, particularly near Corpus Collosum. Generally these kinds of difficult registration problems require a lot of parameter tuning. No way around that. In a later chapter we introduce methods for assessment of registration quality.

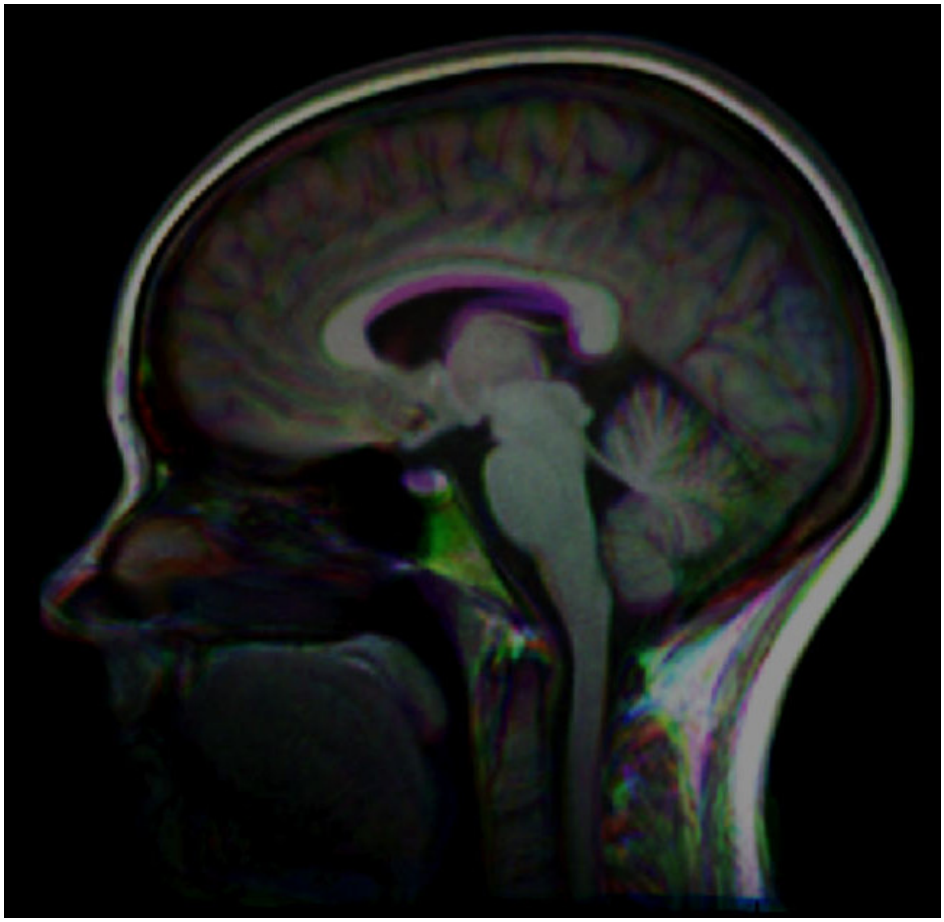


Fig. 8.2: Figure 13: Mean result image.

Tip: We can use the `JoinSeries()` SimpleITK method to construct a 4D image from multiple 3D images and the `Extract()` SimpleITK method to pick out a 3D image from a result 4D image.

Note that the `JoinSeries` method may throw an error "Inputs do not occupy the same physical space!" if image information is not perfectly aligned. These can be caused by slight differences between the image origins, spacing, or axes. The tolerance that SimpleITK uses for these settings can be adjusted using :code: *sitk.ProcessObject.SetGlobalDefaultDirectionTolerance(x)* and :code: *sitk.ProcessObject.SetGlobalDefaultCoordinateTolerance(x)*. We may need to change the image origins to make sure they are the same. This can be done by copying the origin of one of the images `origin = firstImage`.

`GetOrigin()` and setting it to the others `otherImages.SetOrigin(origin)`

CHAPTER 9

Point-based Registation

Point-based registration allows us to help the registration via pre-defined sets of corresponding points. The `CorrespondingPointsEuclideanDistanceMetric` minimises the distance of between a points on the fixed image and corresponding points on the moving image. The metric can be used to help in a difficult registration task by taking into account positions are known to correspond. Think of it as a way of embedding expert knowledge in the registration procedure. We can manually select points or automatically them via centroids of segmentations for example. Anything is possible.

To use `CorrespondingPointsEuclideanDistanceMetric` we append it to the list of metrics in the parameter map.

```
import SimpleITK as sitk

parameterMap = sitk.GetDefaultParameterMap("bspline")
parameterMap["Metric"].append("CorrespondingPointsEuclideanDistanceMetric")
```

Note: The `CorrespondingPointsEuclideanDistanceMetric` metric must be specified as the last metric due to technical constraints in elastix.

The metric can also be added to all metrics in the ElastixImageFilter object with a single call.

```
import SimpleITK as sitk

elastixImageFilter = sitk.ElastixImageFilter()
elastixImageFilter.AddParameter( "Metric", "CorrespondingPointsEuclideanDistanceMetric
↪" )
```

Or to a single parameter map (here we assume that SimpleElastix contains at least two parameter maps)

```
elastixImageFilter.AddParameter( 1, "Metric",
↪"CorrespondingPointsEuclideanDistanceMetric" )
```

The point set are specified as text files. They can either be in VTK pointdata legacy format or elastix' own format that usually has the `pts` extension.

```
<index, point>
<number of points>
point1 x point1 y [point1 z]
point2 x point2 y [point2 z]
```

`index` is used when points are specified in pixel coordinates. `point` is used when the points are specified in world coordinates. For example, this is a valid point file:

```
point
3
102.8 -33.4 57.0
178.1 -10.9 14.5
180.4 -18.1 78.9
```

Transforming Point Sets

We can apply a transformation computed with SimpleElastix to a point set with SimpleTransformix.

```
import SimpleITK as sitk

# Compute the transformation from moving image to the fixed image
elastixImageFilter = sitk.ElastixImageFilter()
elastixImageFilter.SetFixedImage(sitk.ReadImage("fixedImage.nii")
elastixImageFilter.SetMovingImage(sitk.ReadImage("movingImage.nii")
elastixImageFilter.Execute()

# Warp point set. The transformed points will be written to a file named
# outputpoints.txt in the output directory determined by SetOutputDirectory()
# (defaults to working directory)
transformixImageFilter = sitk.TransformixImageFilter()
transformixImageFilter.SetTransformParameterMap(elastixImageFilter.
↪GetTransformParameterMap())
transformixImageFilter.SetFixedPointSet("fixedPointSet.pts")
transformixImageFilter.Execute()
```

Warning: The input points are specified in the fixed image domain (!) and warped from the fixed image to moving image since the transformation direction is from fixed to moving image. If we want to warp points from the moving image to fixed image, we need the inverse transform. This can be computed manually (see section 6.1.6 in the elastix manual) or via `elastixImageFilter.ExecuteInverse()`.

CHAPTER 10

Acknowledgements

SimpleElastix is developed and maintained by Kasper Marstal. Elastix is developed and maintained by Stefan Klein, Erasmus Medical Center, Rotterdam, Netherlands, and Marius Staring, Leiden University Medical Center, Leiden, Netherlands. SimpleITK is developed and maintained by Bradley Lowekamp and fellow Kitware developers.

The reader is refered to the elastix manual for a technical discussion of individual algorithms.