SuperElastix Documentation

Release 1.0

SuperElastix Developers

Jun 15, 2018

Contents

The objective of image registration is to find the spatial relationship between two or more images. In the last decades numerous image registration methods and tools have emerged from the research community. Implementation of these methods, however, are scattered over a plethora of toolboxes each with their own interface, limitations and modus operandi.

SuperElastix is a joint effort of the Biomedical Imaging Group Rotterdam (BIGR) of Erasmus University Medical Center, The Netherlands, and the Division of Image Processing (LKEB) of Leiden University Medical Center, The Netherlands, to provide an open source, multi-platform image registration toolbox written in C++. SuperElastix aims at covering a wide range of image registration methodologies in a single experience, while considering both user-friendliness and algorithm modularity.

Contents:

CHAPTER 1

Getting Started

This page explains how to install SuperElastix.

1.1 SuperBuild On Linux

The SuperElastix SuperBuld is a script that automatically downloads and installs dependencies before compiling the main project. SuperElastix depends on ITK and elastix. The following steps assume that CMake, git and a compiler toolchain is installed. To build SuperElastix, clone the repository and invoke the SuperBuild.

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

To check that SuperElastix was build successfully, go into the `SuperElastix-build` directory and run the test suite with the following command

```
ctest -j4
```

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 SuperElastix with 4 cores (e.g. `make -j4`) we need a machine with at least 16GB of RAM.

1.2 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 SuperElastix. 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.

1.3 SuperBuild On Windows

In this guide we will use CMake to generate build files and the Visual Studio compiler to compile the project.

1. **Setup directories.**
    - Download and install CMake GUI.
    - `git clone https://github.com/SuperElastix/SuperElastix` into a source folder of your choice.
    - Point the CMake source directory to the `SuperElastix/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.
    - SuperElastix can only be compiled with Visual Studio 14 2015 or later, and is tested for 64-bit compilation only.
3. Open Visual Studio, select File -> Open Project/Solution -> Open and choose `SuperElastixSuperBuild` solution.
4. Make sure "Release" build type is selected and build the `ALL_BUILD` project.
5. Right-click on `ALL_BUILD` and click `Build`.
6. The SuperBuildSuperElastix solution only shows each library as a project. To have a more detailed view of SuperElastix open the SuperElastix solution file `<build-path>\SuperElastix-build\SuperElastix.sln` in a new Visual Studio environment.
7. Optionally, Unit Tests and example code can be run by right-clicking on `RUN_TESTS` and clicking `Project Only -> Build Only RUN_TESTS`. Alternatively, individual tests can be run by right-clicking on a specific selx<...>Test project and choosing `Set as StartUp Project`.
8. The SuperElastix commandline executable can be found in `<build-path>\SuperElastixApplications-build\Com`

Warning: A Visual Studio Debug build appears to get link errors, on both VS2015 and VS2017 (Version 15.7.1). Typical error message: `fatal error LNK1318: Unexpected PDB error; OK (0) ''`. However, a Release build should work fine. As a workaround, you may temporarily build a subset of the components, as described down here.

1.4 Building a Subset of Components

SuperElastix allows building a subset of components, by removing some of the components from "selxCompiledLibraryComponents.h". This file is placed in `<build-path>\SuperElastix-build` during the CMake generate step. Doing so may speed up compilation, and it may also work around some linker limitations (as

encountered on Windows Visual Studio Debug builds, described above here). You may place your modified copy of "selxCompiledLibraryComponents.h" elsewhere, and specify its directory path by CMake flag COMPILED_LIBRARY_CONFIG_DIR.

1.5 Manually Building the Required Libraries

Instead of letting the SuperBuild download and build the required libraries manually build libraries can be used as well. The following approach allows us to use a system version of ITK or our own version of elastix.

1. **Build ITK.**
    - Clone ITK from github.com/InsightSoftwareConsortium/ITK.
    - Configure CMake. Set the following CMake variables: BUILD_SHARED_LIBS=OFF, ITK_USE_REVIEW=ON, ITK_WRAP_*=OFF.
    - Compile ITK. Make sure to note the build settings, e.g. Release x64.
2. Build Boost
3. **Build elastix.**
    - 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.
4. **Build SuperElastix.**
    - Clone SuperElastix from github.com/SuperElastix/SuperElastix.
    - Configure CMake. Point ITK_DIR to the location of the ITK build directory and ELASTIX_DIR to the location of the elastix build directory.
    - Build SuperElastix. Make sure to configure the build settings exactly the same as ITK e.g. Release x64.

CHAPTER 2

Commandline tool

This page explains how to use the SuperElastix executable. The executable of the latest release can be downloaded from GitHub releases. Or, if SuperElastix was build from sources, it can be found at:

- Windows: `<build-path>\Applications-build\CommandLineInterface\[Release|Debug]\`
- Linux: `<build-path>/Applications-build/CommandLineInterface/`

By calling `SuperElastix --help` its options are shown.

```
Allowed options:
  --help                produce help message
  --conf arg            Configuration file: single or multiple Blueprints
                        [.xml|.json]
  --in arg              Input data: images, labels, meshes, etc. Usage arg:
                        <name>=<path> (or multiple pairs)
  --out arg             Output data: images, labels, meshes, etc. Usage arg:
                        <name>=<path> (or multiple pairs)
  --graphout arg        Output Graphviz dot file
  --logfile arg         Log output file
  --loglevel arg        Log level [off|critical|error|warning|info|debug|trace]
```

The file format of a Blueprint is described in *Configuring the Network: Blueprints*, where the `<name>`-s of the `--in` and `--out` arguments must correspond to the sinking and sourcing component identifier names as defined in the Blueprint, respectively.

2.1 Demo Experiments

To get started on how to invoke the SuperElastix executable we set up a few scripts that run demo registrations. These four scripts run the demo experiments as described in[1]:

[1] *The design of SuperElastix - a unifying framework for a wide range of image registration methodologies, F. Berendsen, K. Marstal, S. Klein and M. Staring*, found at `<source-path>\Documentation\source\paperWBIR16`.

- 1A_SuperElastix_elastix_NC
- 1B_SuperElastix_elastix_MSD
- 2A_SuperElastix_itkv4_NC
- 2B_SuperElastix_itkv4_MSD

The bash (linux) or batch (windows) scripts are bundled in the release download. Or, if SuperElastix was build from sources, follow these instructions to setup the scripts:

- Windows:
  - open <build-path>\Applications-build\SuperElastixApplications.sln
  - in project solution explorer right-click on Demo -> Project Only -> Build Only Demo
- Linux:
  - change dir to <build-path>/Applications-build/
  - make demo

By building the demo target the SuperElastix executable, image data, configuration files and commandline scripts will be copied to the <DEMO_PREFIX> directory. The <DEMO_PREFIX> can be set by CMake and defaults to <build-path>/SuperElastixApplications-build/Demo.

CHAPTER 3

Design

With SuperElastix we aim to capture a wide range of registration methods, accessible via a single high-level user interface. At the core of our design is a single collection of components with heterogeneous levels of functionality and granularity. This means that a component can implement a single concept (metric, transform, etc.) to be reused in many methods, can implement multiple (tightly coupled) concepts in one, or even be a full registration algorithm. Breaking up algorithms in small components allows a user to mix-and-match component, whereas treating a larger part of algorithms as monolithic blocks lowers the barrier for integration of new methods and paradigms and reuse of existing codebases. By a high level user configuration, components are selected at run-time and are connected via a user-defined network layout. A generic handshake mechanism verifies whether connected components are compatible both mathematically as on a software level. The user is notified if specific combinations are incompatible or not supported (yet).

3.1 Network of components

A classical approach for a registration toolbox is to use an object-oriented design that decomposes the registration problem into classes like metrics, transforms, optimizers, etc. Extended types of behavior (mutual information, affine transform, etc) can be implemented via subtyping. Such a decomposition, however, might not be adequate to handle all of the diversity in existing registration paradigms. And, even within the same paradigm existing toolboxes have slighly different decomposions. Unification of registration methods by refactoring existing toolboxes into one object-oriented design is a tremendous amount of work, if possible at all. Additionally, an object-oriented design typically poses a rather rigid layout how e.g. an optimizer interacts with a metric and a tranforms. Within the large variety of paradigms, there seems to be little consensus to a standard organization. The design of SuperElastix, therefore, takes a different approach. A registration algorithm is considered to be a network of functional components. The network layout (i.e. amount of nodes and connections) is user configurable and the components (i.e. the nodes) are treated uniformely. In our framework each component formalizes its interfaces that determines whether components can connect. Components can have diverse functionalities such as mutual information, affine transform, or a full registration pipeline, but all have a generic mechanism to perform a handshake.

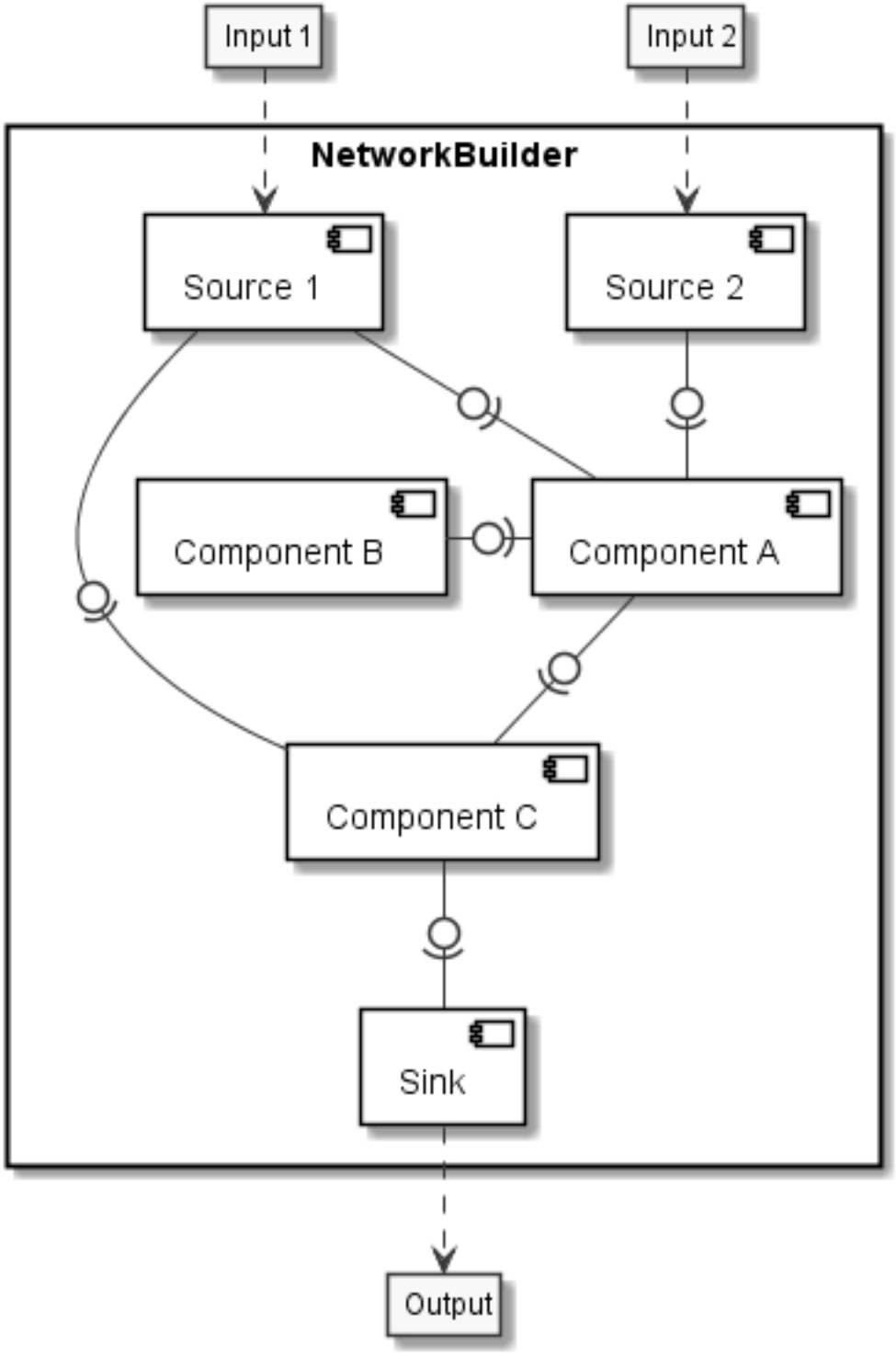

Fig. 1: A user configurable network consisting of many connected Components.

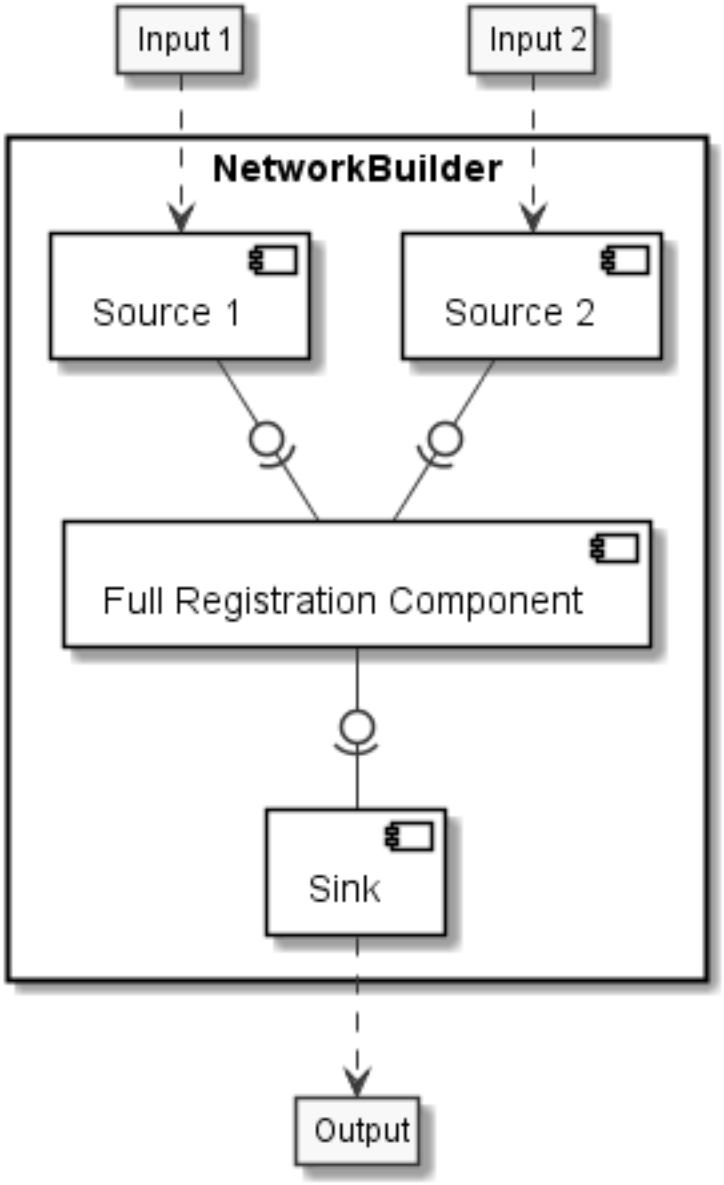

Fig. 2: Components can fulfill many tasks upto a full registration pipeline.

3.2 Configuring the Network: Blueprints

To configure an algorithmic network a Blueprint is passed to SuperElastix. A blueprint contains the description of a full network or is a partial configuration. Hence, multiple Blueprints can be passed to SuperElastix which together form a full configuration. A full configuration defines a mathematical graph in terms of nodes: the `Component`-s, and edges: the `Connection`-s. Properties can be put at both the `Component`-s and the `Connection`-s. These properties are in the form of `Key: Value` pairs, where Keys and Values are strings or Values are lists of strings. The minimal required property for a `Component` is `"Name": <Identifier>`, with <Identifier> being any name in the form of a string. The minimal required properties for a `Connection` are `"Out": <IdentifierA>` and `"In": <IdentifierB>`, with the Identifiers refering to the `Components` it connects.

Listing 1: Layout of a Blueprint json-file

```
{
"Component": {
    "Name": <Identifier>,
    <PropertyKey1> : <PropertyValue1>,
    ...
    <PropertyKeyN> : <PropertyValueN>
}
"Component": {
    ...
}
"Connection": {
    "Out": <IdentifierA>,
    "In": <IdentifierB>,
    <PropertyKeyM> : <PropertyValueM>,
    ...
}
"Connection": {
    ...
}
}
```

Additional properties for a `Component` can be:

- Classname (e.g. `"NameOfClass" : "itkGradientDescentOptimizerv4Component"`)
- A template parameter, (e.g. `"PixelType" : "double"`)
- Settings (e.g. `"NumberOfLevels" : "3"` or `"SmoothingSigmasPerLevel" : ["8", "4", "2"]`)

Additional properties for a `Connection` can be:

- Interface name (e.g. `"NameOfInterface": "itkMetricv4Interface"`)
- template parameter (e.g. `"Dimensionality" : "3"`)
- A tag (e.g. `"Role" : "Fixed"`)

Given that the json file is valid and the blueprint meets the minimal required properties, SuperElastix has a flexible mechanism to parse all additional properties and to realize the algorithmic network without unnescarilly requiring a verbose configuration file and aimed at having a user-extendible database of components. Since, initially, each node in the graph can be any component that is in the component database of SuperElastix, each property is considered an exclusion criterion. Additionally, the properties defined at a connection form exclusion criteria for both components involved. When all exclusion criteria are considered and for any node in the graph it cannnot be uniquely defined which component needs to be realized, SuperElastix will stop and report which component requires more or less criteria.

The advantages of this mechanism are:

- A user can select a component based on its connections (i.e. its role; what it can do) and doen't have to know the particular name of the class.
- By extending SuperElastix with new components that have identical connections to existing components it will automatically ask the user to provide additional (discriminating) properties.
- A user, for instance, does not need to specify "Dimensionality" : "3" for all components in the Blueprint if this can be deduced from one component in the network via its connections.
- A partially defined Blueprint can leave certain properties open, such that these can be defined by an other partially configuring blueprint that is passed to SuperElastix additionally.

3.3 Generic handshake mechanism

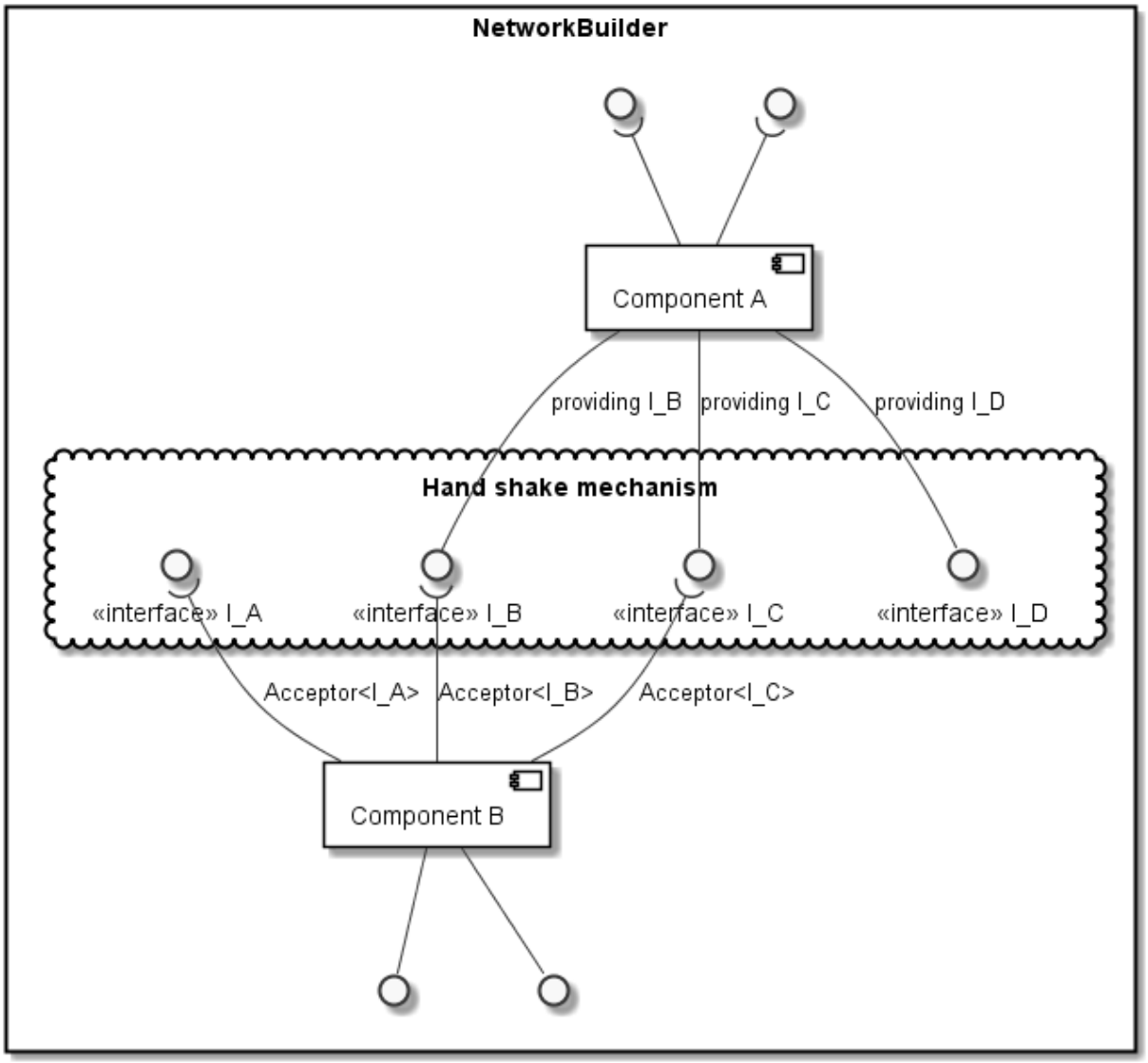

Fig. 3: The handshake mechanism validates whether components can be connected or not.

Prior to running the registration algorithm contructed from a user-defined network, SuperElastix validates whether components can be connected or not. This hand mechanism performs the necessary checks on what a component can do (as defined by its interfaces), which is required to establish a connection. The advantage of explicitly handling this generically and on a higher level, is that components themselves do not need to perform these checks on neighboring components, which would require a component to embed specific knowledge about other components.

To manage all possible types of collaboration, SuperElastix maintains an extensible collection of component interfaces. Any component in the toolbox must be defined in terms of one or more interfaces, which are either accepting or providing. After succesfull handshakes all components check if sufficient accepting interfaces have been connected. The underlying implementation details can be found in the development section.

CHAPTER 4

Library Usage

4.1 SuperElastixFilter input and output datatypes

The SuperElastixFilter is designed to be part of an itk pipeline such that it can connect to other itk filters and supports the itk update mechanism. All input and output data required by the (configured) algorithm are exposed by the SuperElastixFilter. In this philosophy the SuperElastixFilter does/should not read or write data (images, meshes, etc) from disk directly. Therefore, in the commandline tool, which uses the SuperElastixFilter, readers and writers reside outside the SuperElastixFilter. However, unlike common itk filters, the inputs and outputs of the SuperElastixFilter are typically unknown at compile time, because they depend on the Blueprint configuration describing the actual algorithm to execute. This complicates the setup of a pipeline, since up and downstream itk filters are typically templated over their datatypes. To stay as close as possible to the itk philosophy, the SuperElastixFilter supports 2 modes of operation:

- *Known input and output types at compilation time*: E.g. to connect the SuperElastixFilter to conventional ITK filters, which are templated on data types, or an application that embeds a dedicated registration task. That is, the application developer makes sure that any Blueprints to be used, will correspond to the (compile-time defined) number and types of inputs and outputs by known identifier names (defined by the Sink and Source Components). In this mode, the order in which the inputs and outputs are connected to other filters, and the Blueprint (Object) is set, is arbitrary. However, to connect the output of the SuperElastixFilter a templated version of GetOutput must be used: `ImageFileWriter<KnownImageType>::Pointer my_writer; ... my_writer->SetInput(superElastixFilter->GetOutput<KnownImageType>(identifier))`. An example snippet:

Listing 1: Example usage if input and output types are known at compilation time

```
// Set up the ITK reader
using InputImageType = itk::Image<float,3>;
using ImageReaderType = itk::ImageFileReader<InputImageType>;
ImageReaderType::Pointer reader = ImageReaderType::New();
reader->SetFileName( path );

// Set up the ITK writer
using OutputImageType = itk::Image<double,3>;
```

(continues on next page)

(continued from previous page)

```
using ImageWriterType = itk::ImageFileWriter<OutputImageType>;
ImageWriterType::Pointer writer = ImageWriterType::New();
writer->SetFileName( path );

// Connect the ITK pipeline (in arbitary order)
// Assume superElastixFilter was instantiated.
superElastixFilter->SetInput( "FixedImage", reader->GetOutput());
// The output of superElastixFilter needs to be made of OutputImageType explicitly.
writer->SetInput(superElastixFilter->GetOutput<OutputImageType>( "ResultImage" ));
// Assume blueprint was instantiated. It is requered that the blueprint defines the␣
↪a source
// component the named "FixedImage" that corresponds to the InputImageType. This␣
↪holds
// for the sink component "ResultImage" and OutputImageType.
superElastixFilter->SetBlueprint(blueprint);

// Updating the writer makes the superElastixFilter first parse the blueprint and the
// connection before it executes.
writer->Update();
```

- *Unknown input and output types at compilation time*: E.g. the class implementing the commandline interface is not aware of the datatypes used by all components. (In this way, adding custom components with new types does not affect the source code of the commandline interface). The commandline interface is invoked by pairs of filenames and identifier names. The identifiers refer to Sink or Source Components as defined via the Blueprint that, in turn, define the data types. In this mode, the commandline interface typically cannot instantiate readers or writers because they are templated over the data types. Instead, the SuperElastixFilter is requested to return appropriate readers and writers corresponding to the identifier names. SuperElastix will return respectively an AnyReader or AnyWriter, which are non-templated Base Classes that, if updated, use the appropriate reader of writer internally (by use of polymorphism): `AnyWriter::Pointer my_writer; ... my_writer->SetInput(superElastixFilter->GetOutput(identifier))`. In this mode, it is required to set the Blueprint prior to request and connect readers or writers. An example snippet:

Listing 2: Example usage if input and output types are unknown at compilation time

```
// Assume superElastixFilter and blueprint were instantiated.
// Set Blueprint first, which defines a source component called "FixedImage" and a␣
↪sink
// component called "ResultImage".
superElastixFilter->SetBlueprint(blueprint);

// Get AnyReader for "FixedImage", this triggers the parsing of the Blueprint.
selx::AnyFileReader::Pointer reader = superElastixFilter->GetInputFileReader(
↪"FixedImage" );
reader->SetFileName( path );

// Get AnyWriter for "ResultImage"
selx::AnyFileWriter::Pointer writer = superElastixFilter->GetOutputFileWriter(
↪"ResultImage" );
writer->SetFileName( path );

// Connect the ITK pipeline
superElastixFilter->SetInput( "FixedImage", reader->GetOutput() );
writer->SetInput( superElastixFilter->GetOutput( "ResultImage" ) );

// Updating the writer makes the superElastixFilter to execute.
```

(continues on next page)

(continued from previous page)

```
writer->Update();
```

Mixing these to modes of operation is allowed too.

4.2 SuperElastixFilter component database manipulation

We provide two library interfaces, each supporting a different use case:

- *"Precompiled" SuperElastix ITK filter*, designed to be used in external applications, such as the commandline interface or company applications.
- *"Templated" SuperElastix ITK filter*, offering the most flexibility, useful for external third-party components and extreme use cases.

In both cases SuperElastixFilter has an internal database of components that can be used to dynamically construct the registration algorithm of choice. In the "Precompiled" library this database is populated with a predefined list of components (each with predefined template arguments, such as dimensionality and pixel type, etc). Predefinition of the components allows for hiding the implementation details of the components and speeds up the compilation process of the application (done via the Pimpl idiom). The "Precompiled" library is still and ITK filter and depends on the (templated) header files of the itk library. The superElastixFilter is instantiated like this:

Listing 3: Example usage of "Precompiled" SuperElastix ITK filter

```
#include "selxSuperElastixFilter.h"
selx::SuperElastixFilter::Pointer superElastixFilter =␣
↪selx::SuperElastixFilter::New();
```

In the "Templated" library the database of components can be populated by the user at compilation time by passing the component classes as template arguments. Applications using this library need access to all of SuperElastix internal source and header files at compilation time. This approach provides the flexibility to compile an instance of the SuperElastix ITK filter with, for instance, a sub- or superset of the default components, a set of components with exotic dimensionality or pixel types or even with third party components. Compiling the SuperElastix ITK filter with a small set of components is typically done in our Unit tests when testing a specific component or combination of components. Adding a third-party component to SuperElastix via template arguments does not require any modification of the source code files of the SuperElastixFilter. A third-party component can adhere to the existing already defined interfaces classes, but op top of that it can also define new interface classes. For example, the templated superElastixFilter is instantiated like this:

Listing 4: Example usage of "Templated" SuperElastix ITK filter

```
#include "selxSuperElastixFilterCustomComponents.h"
// ... and #include all headers of the components used

/** Construct a list with user required components */
using RegisterComponents =  TypeList<
  ItkImageSourceComponent< 2, float >,
  DisplacementFieldItkImageFilterSinkComponent< 2, float >,
  ItkImageRegistrationMethodv4Component< 3, double, double >,
  ItkImageRegistrationMethodv4Component< 2, float, double >,
  ItkANTSNeighborhoodCorrelationImageToImageMetricv4Component< 2, float >,
  ItkMeanSquaresImageToImageMetricv4Component< 2, float, double  >,
  ItkGradientDescentOptimizerv4Component< double >,
  ItkAffineTransformComponent< double, 2 >,
  ItkTransformDisplacementFilterComponent< 2, float, double >,
```

(continues on next page)

(continued from previous page)

```
  RegistrationControllerComponent< >
  >;

SuperElastixFilterBase::Pointer superElastixFilter =
  SuperElastixFilterCustomComponents< RegisterComponents >::New();
```

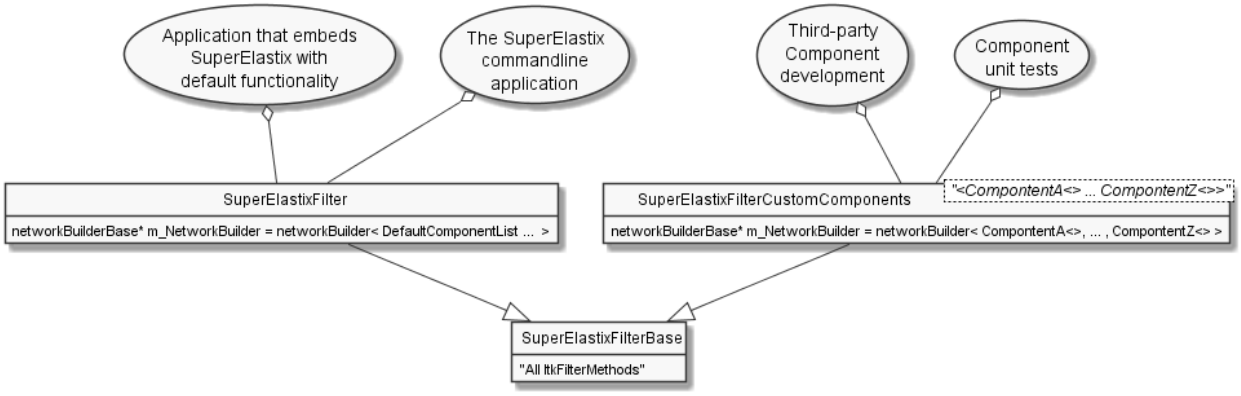

Fig. 1: UML use case diagram for “Templated” and “Precompiled” library

CHAPTER 5

User Component Creation

This is a quick overview how to add you own registration components to SuperElastix.

Test if you can build SuperElastix locally:

- git clone github.com/SuperElastix/SuperElastix (to contribute consider forking first)
- use cmake with our SuperBuild to get and build all external dependencies automatically, see *Getting Started*.

To create a new component:

- The build process of SuperElastix *automatically scans* its `Modules/Components/` directory for the source files of components.
- Create a Component group directory `Modules/Components/<MyComponentGroup>`. A registration method can be composed of multiple components that can be grouped.
- Tip: copy one of the other Component Groups and change the names.

The minimal required directory structures should be as following:

```
Modules/Components/<MyComponentGroup>/Module<MyComponentGroup>.cmake
Modules/Components/<MyComponentGroup>/include/selxModule<MyComponentGroup>.h
```

- The `Module<MyComponentGroup>.cmake` file lists all components source files and `selxModule<MyComponentGroup>.h` file lists which components (with various template arguments) are compiled into SuperElastix.

5.1 SuperElastixComponent class

A SuperElastix Component consists of accepting and providing interfaces. To let the handshake mechanism handle a component correctly the component (class) must adhere to the following structure. The component class must derive from the `SuperElastixComponent` class (solely). The `SuperElastixComponent` is a templated class with signature `< <Providing<I_A, I_B, ... >, Accepting<I_C, I_D, ... > > >`, with classes `Providing` and `Accepting` acting as placeholders to indicate the role of the interfaces `I`. By inheriting from the

`SuperElastixComponent` class the component developer needs to provide the implementation for a number of methods. These are:

- All methods that have been defined in the providing interface classes that component developer selected.
- A `virtual void Accept(I_x*)` for each interface class `I_x` that has been selected as accepting interface. (This example uses raw pointes, but in the reality we use code:*std::shared_ptr* for this).
- The `virtual bool MeetsCriterion( const CriterionType & criterion )`, which returns true if and only if the component has an implementation for which the criterion (read from the Blueprint) holds or can be fulfilled.

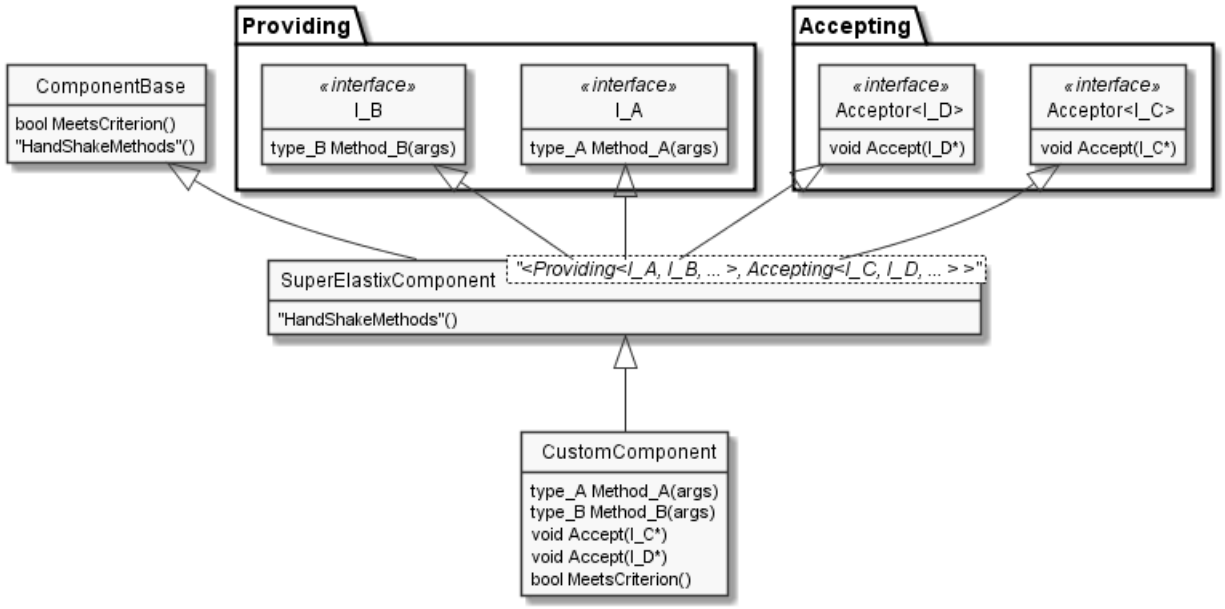

Fig. 1: UML class inheritance diagram of a Component in SuperElastix

Listing 1: Layout of an example component of SuperElastix

```
// Required include guards
#ifndef selxExampleComponent_h
#define selxExampleComponent_h

// Required include of selxSuperElastixComponent
#include "selxSuperElastixComponent.h"

// Optionally include other interface definitions
// #include "selxSinksAndSourcesInterfaces.h"

// Optionally include your code base specific headers.
// ...

namespace selx
{
// Choose your own template arguments for the component
template< int Dimensionality, class PixelType, class TInternalComputationValue >
class ExampleComponent :
  public SuperElastixComponent<
```

(continues on next page)

(continued from previous page)

```
    // define any number of Accepting interfaces
    Accepting<
      ExampleAInterface< Dimensionality >,
      ExampleBInterface< TInternalComputationValue, Dimensionality >
    >,
    // define any number of Providing interfaces
    Providing<
      ExampleCInterface< Dimensionality, PixelType >
    >
  >
{

public:

  // Important: the definition of Superclass must match the definition above.
  using Superclass = SuperElastixComponent<
    Accepting< ExampleAInterface< Dimensionality >,
    ExampleBInterface< TInternalComputationValue, Dimensionality >
    >,
    Providing< ExampleCInterface< Dimensionality, PixelType >
    >
  >;

  // A constructor with arguments for name and logger is required.
  ExampleComponent( const std::string & name, LoggerImpl & logger );

  virtual ~ExampleComponent();

  //For each Accepting Interface a Accept method must be implemented:
      // Accepting ExampleAInterface
  virtual int Accept( typename ExampleAInterface< Dimensionality >::Pointer )␣
↪override;

      // Accepting ExampleBInterface
  virtual int Accept( typename ExampleBInterface< TInternalComputationValue,␣
↪Dimensionality >::Pointer ) override;

  // All methods in all Providing Interfaces must be implemented:
      // Providing ExampleCInterface
  virtual SomeImageType<PixelType, Dimensionality>* GetImage() override;

  //BaseClass methods
  virtual bool MeetsCriterion( const ComponentBase::CriterionType & criterion )␣
↪override;

      // Optional: The default implementation, which requires all Accepting␣
↪interfaces to be connected, can be overridden
      // virtual bool ConnectionsSatisfied() override;

private:

  // Typically a component stores the pointer to the Interfaces it accepts by␣
↪Accept(), however
  // this is not required.
  typename ExampleAInterface< Dimensionality >::Pointer m_ExampleAInterface;

  // Optionally include your own methods and members
```

(continues on next page)

(continued from previous page)

```
  // ...

protected:

  // Optional, but recommended: TemplateProperties() is typically used in␣
↪MeetsCriterion()
  // return the class name and the template arguments to uniquely identify this␣
↪component.
  static inline const std::map< std::string, std::string > TemplateProperties()
  {
  return { { keys::NameOfClass, "ExampleComponent" },
           { keys::PixelType, PodString< PixelType >::Get() },
           { keys::InternalComputationValueType, PodString< TInternalComputationValue␣
↪>::Get() },
           { keys::Dimensionality, std::to_string( Dimensionality ) }
         };
  }
};
} //end namespace selx
#ifndef ITK_MANUAL_INSTANTIATION
#include "selxExampleComponent.hxx"
#endif
#endif // #define ExampleComponent_h
```

Listing 2: Interface definitions of an example component of SuperElastix

```
// And interface class is pure virtual, thus no methods have an implementation at␣
↪this stage
template< int Dimensionality >
class ExampleAInterface
{
public:
  // Some convenience typedefs
  using Type = ExampleAInterface< Dimensionality>;
  using Pointer = std::shared_ptr< Type >;

  // Define 1 or more methods, with any type of input and output arguments.
  virtual int MethodA1() = 0;
  // virtual bool MethodA2( TInternalComputationValueType value) = 0;
};

template< class TInternalComputationValueType, int Dimensionality >
class ExampleBInterface
{
  // ...
};

template< class PixelType, int Dimensionality >
class ExampleCInterface
{
  using Type = ExampleCInterface< PixelType, Dimensionality>;
  using Pointer = std::shared_ptr< Type >;
  virtual SomeImageType<PixelType, Dimensionality>* GetImage( ) = 0;
};

// ...
```

(continues on next page)

(continued from previous page)

```
template< class PixelType, int Dimensionality >
struct Properties< ExampleCInterface< PixelType, Dimensionality >>
{
  static const std::map< std::string, std::string > Get()
  {
    // return all the properties how to identify this interface as strings
    return { { keys::NameOfInterface, "ExampleCInterface" }, // required: class name
      { keys::PixelType, PodString< PixelType >::Get() }, // required: all template
↪arguments
      { keys::Dimensionality, std::to_string( Dimensionality ) },
      { "Role", "Fixed" } // optional: more descriptive properties to select this
↪interface
    };
  }
};
```

5.1.1 MeetsCriterion

Each Component needs to implement its `MeetsCriterion` method. The primary task of this method is to let the network builder select the right component based on the properties the user defined in the blueprint, as described in the section *Configuring the Network: Blueprints*. The network builder does this by passing one property key-value pair at the time to the component, which replies if it accepts the property key and if so whether it accepts the property value. To handle the properties that are template parameters, e.g. `"PixelType" : "double"`, SuperElastix has the convience function `CheckTemplateProperties`. The handling of all other properties needs to be implemented explicitly. In this lies also the secondary task of `MeetsCriterion`, that is, storing or using each property key-value pair that is a parameter setting of the component, such as `"NumberOfLevels" : "3"` or `"SmoothingSigmasPerLevel" : ["8", "4", "2"]`. In most cases (e.g. when wrapping an ITK filter) this means setting the properties at the wrapped member class directly.

5.2 Cmake module selection system

The modules can specify on which of the other modules they depend, and the build system make sure dependencies are enabled, and that they are enabled in the correct order. This means that users are always building the smallest possible binary, reducing binary size and compilation time. The following output shows the result of the default build, which builds the library interface along with elastix, nifityreg and the ITKv4 registration methods.

```
-- Found the following SuperElastix modules:
--   ModuleBlueprints
--   ModuleCommon
--   ModuleComponentInterface
--   ModuleController
--   ModuleElastix
--   ModuleExamples
--   ModuleItkSmoothingRecursiveGaussianImageFilter
--   ModuleNiftyreg
--   ModuleSinksAndSources
--   ModuleItkImageRegistrationMethodv4
--   ModuleItkSyNImageRegistrationMethod
--   ModuleConfigurationReader
--   ModuleFileIO
--   ModuleFilter
```

(continues on next page)

(continued from previous page)

```
--   ModuleLogger
-- Enabling ModuleFilter requested by SuperElastix.
-- Enabling ModuleBlueprints requested by ModuleFilter.
-- ModuleBlueprints enabled.
-- Enabling ModuleController requested by ModuleFilter.
-- ModuleController enabled.
-- Enabling ModuleElastix requested by ModuleFilter.
-- ModuleElastix enabled.
-- Enabling ModuleExamples requested by ModuleFilter.
-- Enabling ModuleComponentInterface requested by ModuleExamples.
-- Enabling ModuleCommon requested by ModuleComponentInterface.
-- ModuleCommon enabled.
-- Enabling ModuleFileIO requested by ModuleComponentInterface.
-- ModuleFileIO enabled.
-- ModuleComponentInterface enabled.
-- ModuleExamples enabled.
-- Enabling ModuleItkImageRegistrationMethodv4 requested by ModuleFilter.
-- ModuleItkImageRegistrationMethodv4 enabled.
-- Enabling ModuleItkSmoothingRecursiveGaussianImageFilter requested by ModuleFilter.
-- ModuleItkSmoothingRecursiveGaussianImageFilter enabled.
-- Enabling ModuleSinksAndSources requested by ModuleFilter.
-- Enabling ModuleController requested by ModuleSinksAndSources.
-- ModuleController already enabled.
-- ModuleSinksAndSources enabled.
-- Enabling ModuleNiftyreg requested by ModuleFilter.
```

Modules are enabled once, even when requested multiple times, and can be turned off and on via CMake.

To add a module to SuperElastix, the developer creates a new directory and a CMake file that honor some naming conventions. The name of CMake file should Module[Name].cmake where [Name] is the name of the module. The CMake file contains a collection of CMake variables that the build system will use to integrate the module as component in the SuperElastixFilter. Users will never have to touch code outside module directory.

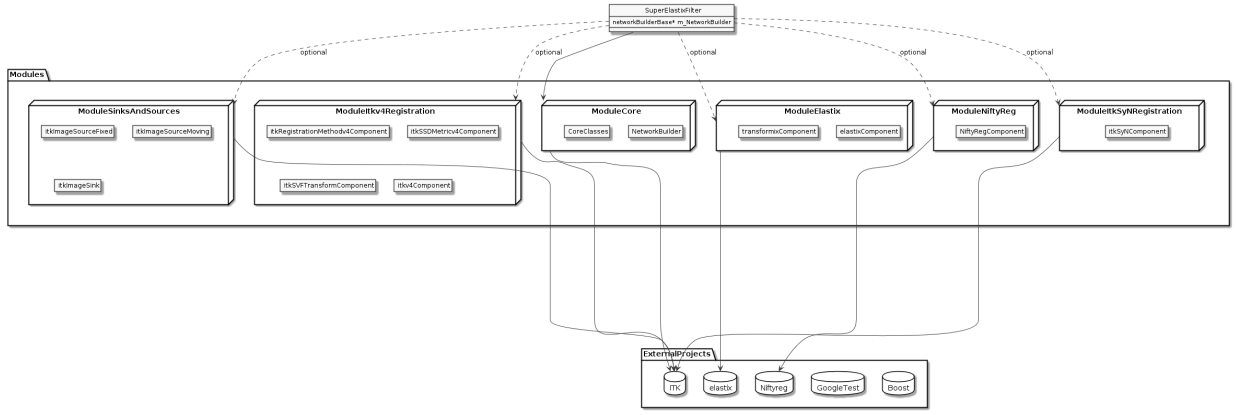

Fig. 2: Modules, Components and external projects

CHAPTER 6

ToolBoxes

This page lists the tools that are embedded in SuperElastix or are considered to be.

Table 1: Embedded Toolboxes.

Name	Full Name	Url
elastix		http://elastix.isi.uu.nl/
NiftyReg		https://cmiclab.cs.ucl.ac.uk/mmodat/niftyreg
ANTS (itkv4)	Advanced Normalization Tools	https://github.com/ANTsX/ANTs
	SYN: Symmetric Normalization	https://www.ncbi.nlm.nih.gov/pubmed/17659998
	DMFFD: itkBSplineSyN	https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3870320/
	DMFFD: itkTimeVaryingBSplineVelocityField	
	itkTimeVaryingVelocityField	

Table 2: Considered Toolboxes.

Name	Full Name	Url
MIRTK	The Medical Image Registration ToolKit	https://github.com/BioMedIA/MIRTK
plastimatch		http://plastimatch.org/
Variational	ITK Variational Registration Framework	http://www.insight-journal.org/browse/publication/917
Demons (itk)		https://itk.org/Doxygen/html/classitk_1_1DemonsRegistrationFilter.html
	DiffeomorphicDemons	http://www.insight-journal.org/browse/publication/154
	PDEDeformableRegistration	https://itk.org/Doxygen/html/classitk_1_1PDEDeformableRegistrationFilter.html
	Symmetric Log-Domain Diffeomorphic Demons	https://hal.inria.fr/hal-00813744/document
DRAMMS	Deformable Registration via Attribute Matching and Mutual-Saliency Weighting	http://www.med.upenn.edu/sbia/dramms.html
MIRORR	Multimodal Image Registration using blOck-matching and Robust Regression	http://aehrc.github.io/Mirorr/
BRAINSTools/BRAINSFit	A suite of tools for medical image processing focused on brain analysis	https://github.com/BRAINSia/BRAINSTools/tree/master/BRAINSFit

Table 3: Incompatible/Unclassified Toolboxes.

Name	Full Name	Url
ABSORB	ABSORB for groupwise registration	
DROP	Deformable Image Registration using Discrete Optimization	
DTI-DROID	Deformable Registration using Orientation and Intensity Descriptors	
MIND	Modality Independent Shape Descriptor for Multimodal Deformable Registration	http://www.ibme.ox.ac.uk/research/biomedia/julia-schnabel/Software
DEEDS	Dense Displacement Sampling for Deformable Registration	http://www.ibme.ox.ac.uk/research/biomedia/julia-schnabel/Software
uTIlzReg	Diffeomorphic registration of 2D/3D nifti images	http://www.ibme.ox.ac.uk/research/biomedia/julia-schnabel/Software
FLIRT	FMRIB's Linear Image Registration Tool	
FNIRT	FSL non-linear registration tool	
FAIR	flexible algorithms for image registration	
LDDMM	The Large Deformation Diffeomorphic Metric Mapping tool	
DARTEL	Diffeomorphic Anatomical Registration using Exponenial Lie algebra	
GS	Geodesic shooting	
IRTK	Image Registration Toolkit	
AIR	Automatic Image Registration	
ART	The Automatic Registration Toolbox	
SAFIR-FLIRT	Fast and flexible image registration toolbox	
MIRT	Medical Image Registration Toolbox for Matlab	
MIPAV	Medical Image Processing, Analysis, and Visualization	
HAMMER	Hierarchical Attribute Matching Mechanism for Elastic Registration	
Animal		
CVS FreeSurfer	combined volumetric and surface-based (CVS) registration	
SICLE	small deformation inverse consistent linear elastic	
SLE		
JRD-fluid		
NAMIC sandbox		

CHAPTER 7

Dashboard

All branches and pull requests are build and unit tested by Continuous Integration: see our CDash for the latest status: https://my.cdash.org/index.php?project=SuperElastix