# WrapITK: Enhanced languages support for the Insight Toolkit

*Release 0.1*

Gaëtan Lehmann[1], Zachary Pincus[2] and Benoit Regrain[3]

March 24, 2006

[1]Unité de Biologie du Développement et de la Reproduction, Institut National de la Recherche Agronomique, 78350 Jouy-en-Josas, France
[2]Program in Biomedical Informatics and Department of Biochemistry, Stanford University School of Medicine, Stanford, California
[3]CREATIS, CNRS UMR 5515, 69621 Villeurbanne, France

**Abstract**

ITK [1] is a huge image analysis library, which contains lots of state of the arts algorithms implementations. However, using it in C++ can be difficult and is definitively bad suited for prototyping. WrapITK aims to allow classes from ITK (and custom, classes that interact with ITK) to be ”wrapped” for use with languages like Python [2], Tcl [3], and Java [4].

## Contents

# 1 Introduction

WrapITK is a project designed to allow classes from ITK (and custom, classes that interact with ITK) to be ”wrapped” for use with languages like Python [2], Tcl [3], and Java [4].

Note that ITK already has a wrapping infrastructure, and that WrapITK is based on it, and use the same tools: CMake [5], GCC-XML [6] and CableSwig[1] [8]. This project aims to address the following deficits of the existing wrappers (and others):

- ITK is a huge library, but only a small number of classes are available in target languages. It become quickly frustrating for the user, especially when he has to spend lot of time to extend the current set of classes. Even if it is not yet complete, the WrapITK’s set of classes have been highly extended. Moreover, the user can choose at build time which types and which dimensions he want to wrap.
- The template arguments set is poorly chosen, making sometime impossible to create a pipeline. In WrapITK, most of the filters have the same input and output types, and only a few filters allow to change type. This make the types manipulated by filters more consistent, and the user should always be able to build his pipeline.
- Lots of types returned by ITK object’s methods are not usable in target languages. For example, the `GetPixel()` method of the class `itk::Image` returns a string describing a pointer, but don’t return the pixel value. In WrapITK, most types used in classes are available in target languages.

[1] CableSwig is based on a now quite old version of SWIG [7].

- Names in target languages are inconsistent. WrapITK use a strict naming convention which should make easier to identify the template arguments.
- The ITK wrapping system is difficult to understand and maintain. WrapITK was written - and thoroughly documented - to be as easy as possible to understand, maintain, and extend.
- It is non-trivial to add wrappers for different ITK classes to the system. In WrapITK, adding a wrapper can be as simple as adding a single file containing a few well-documented cmake macros.
- It is difficult if not impossible to add original-style ITK wrappers for external C++ classes that interact with ITK. WrapITK provides explicit hooks for external C++ classes to be wrapped and even installed in the WrapITK tree so that they interact seamlessly with the other wrapped classes.
- The python's `InsightToolkit` module is only structured as a big list of names. It make it nearly unusable in the python interpreter. WrapITK comes with a new well designed python module easy to use in interpreter, and providing run-time lookup of templated types - thing which can't be easily done in C++. Additionally, WrapITK ensures that `SmartPointer`s are always returned and acceptable as input, so no bare pointers are ever exposed to Python. This is not the case in the standard ITK wrappers.
- ITK was broken on MacOS X [9] with python.

Currently, WrapITK have been tested on Mandriva Linux [10], and MacOS X, and only supports Python properly; Java and Tcl libraries build correctly but the Java and Tcl support classes for loading these libraries is entirely out of date and is not working.

The article you're reading is not as nice and complete than what we would have done, but it seems important to us to release our work and to get feedbacks as soon as possible. The article will continue to evolve with WrapITK.

# 2 User guide

## 2.1 Installation

### Get the software

A tarball archive is submitted with the article.

To get the last version from the darcs repository can be obtained with the command `darcs get http://voxel.jouy.inra.fr/darcs/contrib-itk/WrapITK/`.

For the user who don't want to use darcs [11] but want the last development version, a nightly updated archive is available at `http://voxel.jouy.inra.fr/darcs/contrib-itk/WrapITK/WrapITK.tar.gz`.

RPM packages for Mandriva Linux 2006 are available at `http://voxel.jouy.inra.fr/mdk/mima2`.

### CableSwig

WrapITK requires ITK and CableSwig to have been previously downloaded and built. To get CableSwig, simply run: `cvs -d:pserver:anonymous@public.kitware.com:/cvsroot/CableSwig co CableSwig` (Note that no cvs login is needed here.)

If you check out CableSwig into the `Insight/Utilities` directory, then it will be built as a part of ITK, and will be automatically detected by WrapITK when ITK is found.

### Required and optional patches

WrapITK will work properly with the ITK 2.6.0 release.

There are some optional patches to the ITK source in `WrapITK/patches/optional` which can be applied to version 2.6.0. These optional patches provide better support for python by providing some methods like `__str__`, or methods for standard python sequence interface (see below).

### Build options

After CableSwig and ITK have been (possibly patched) and built, building WrapITK with cmake is simple. Run `ccmake` in a new directory with the path to the WrapITK source tree as the first argument, and provide the locations of the ITK and CableSwig build trees if ccmake so requests. Build options are relatively self-explanatory.

The project is provided with defaults build option which should OK for most of users. However, for specific needs, you might want to change those options:

- `WRAP_TEMPLATE_IF_DIMS` is the list of dimensions which will be available in the target languages. The dimensions must be separated by a semicolon (`;`). By default dimensions 2 and 3 are available.
- `WRAP_covariant_vector_double`, `OFF` by default.
- `WRAP_covariant_vector_float`, `ON` by default.
- `WRAP_double` `OFF`, by default.
- `WRAP_float` `ON`, by default. Note that it is the only signed type selected by default, so you will have to use floats to manipulate signed values.
- `WRAP_rgb_unsigned_char`, `ON` by default.
- `WRAP_rgb_unsigned_short`, `OFF` by default.
- `WRAP_signed_char`, `OFF` by default.
- `WRAP_signed_long`, `OFF` by default.
- `WRAP_signed_short`, `OFF` by default.
- `WRAP_unsigned_char`, `OFF` by default.
- `WRAP_unsigned_long`, `OFF` by default. Some filters, like `WatershedImageFilter` require this type. Some filters to return to a wrapped type from `unsigned long` are provided, even if this option is set to `OFF`.
- `WRAP_unsigned_short`, `ON` by default. `unsigned short` is the only integer type available by default. This type have been choose rather than `unsigned char` to be able to manipulate 8-bits as well as 16-bits images, and to be able to manipulate labeled images more than 255 labels. It is still possible to save images with the `unsigned char` type, even if `WRAP_unsigned_char` is set to `OFF`.

- WRAP_vector_double, OFF by default.
- WRAP_vector_float, ON by default.

The user should modify those options carefully: activate all the types, and/or add lots of dimensions will produce very large binary files which will take lots of memory once loaded.

Note that each individual filter that is wrapped can declare which dimensions it should be wrapped for, and what image types it can accept. For example, a filter could declare that it should only be wrapped for 3D images with floating-point typed pixels. In this case, then wrappers will only be created if the user has selected to build 3-dimensional image wrappers and has selected one or more floating point types (e.g. double or float) in ccmake. Thus, the ccmake configuration specifies the maximum possible range of image and filter types to be created, and each filter is wrapped for some subset of that range.

Project should always be built outside the source directory, in a build directory for example.

#### Install WrapITK or use it in the build tree

Once built, WrapITK can be installed or used in place.

## 2.2 Python usage

#### Configuring python and importing the libraries

If WrapITK has been installed, then using it from within python is trivial: simply issue the command import itk, and you are ready to go. This is because WrapITK installs a .pth file in the python site-packages directory so that python knows where to find the itk scripts.

On linux boxes however, the user have to set the LD_LIBRARY_PATH to point to libSwigRuntime.so. For example export LD_LIBRARY_PATH=/usr/lib/InsightToolkit/WrapITK/Python-SWIG.

If WrapITK has not been installed, then you will either need to set the PYTHONPATH environment variable to contain the directory /path-to-WrapITK-build/Python, add this path to sys.paths within python, or start python from that directory. After this, import itk will work properly.

#### Template usage

Most class in the itk python module are ”template proxy classes” that encapsulate all of the template instantiations that were created at build time. If three-dimensional unsigned char and unsigned short image types were created, they can be accessed as follows:

- itk.Image[itk.UC, 3]
- itk.Image[itk.US, 3]

Note that the C type unsigned char is given with itk.UC, and unsigned short with itk.US.

The template parameters can also be put in a variable, and declared once in a script:

```
dim = 3
pixelType = itk.UC
imageType = itk.Image[pixelType, dim]

image = imageType.New()
```

This construction is similar to what is done in C++, and make it easy to change the dimension used for example - it can even be changed at run-time.

A more convenient syntax for usage in interpreter is also available:

- `itk.Image.UC3`
- `itk.Image.US3`

`itk.Image.UC3` refere to the same class than `itk.Image[itk.UC, 3]` but have the advantage to allow to use the tab-completion in the interpreter, and so let the user easily know which template arguments he can use. However, this notation is more rigid than the one above and won't let the user specify the type and the dimension used in a single place, and thus, should be use only in interpreter.

Filters templated on images can be similarly accessed:

- `itk.ImageFileReader[itk.Image[itk.UC,3]]`
- or `itk.ImageFileReader[itk.Image.UC3]`
- or `itk.ImageFileReader.IUC3`
- or `itk.ImageFileReader[imageType]`
- or even with an instance of the class used as template parameter: `itk.ImageFileReader[image]`.

This makes it easy to write generic routines which can deal with any input image type. For example, a function which take an image as parameter and write it to a file without having to give the image type can be:

```
def write( image, fileName ) :
    writer = itk.ImageFileWriter[ image ].New()
    writer.SetFileName( fileName )
    writer.SetInput( image )
    writer.Update()
```

#### The *New()* method

Lots of classes have a `New()` method which returns a smart pointer to that class. The `New()` method in python has some additional features:

- Arguments to the new method are assumed to be filter inputs. So you could write:

```
adder = itk.AddImageFilter[...].New()
adder.SetInput1( readerA.GetOutput() )
adder.SetInput2( readerB.GetOutput() )
```

or you could write

```
adder = itk.AddImageFilter[...].New( readerA.GetOutput(), readerB.GetOutput() )
```

or even

```
adder = itk.AddImageFilter[...].New( readerA, readerB )
```

In that case, `New` will use the `GetOutput()` method of the object, if it exist, to get the image and set the inputs of the new filter.

- Additionally, keyword arguments are allowed as well. Keyword arguments cause the corresponding `Set...` method to be called, so you could write the following:

```
itk.ImageFileWriter[image].New(image, FileName="foo.tif")
```

or

```
itk.ImageFileWriter[image].New(Input=image, FileName="foo.tif")
```

With that notation, the `write` function becomes more simple:

```
def write( image, fileName ) :
    writer = itk.ImageFileWriter[ image ].New( image, FileName=fileName )
    writer.Update()
```

and, more important, most of classes can be instantiated and parametered in one line, which make ITK less verbose, and a lots more easy to use in the interpreter.

### Python sequences and ITK

To set the radius of a `MedianImageFilter` object, for example, the user have to create a `Size` object and use it as argument of the `SetRadius()` method.

```
12> radius = itk.Size[2]()

13> radius.SetElement(0, 3)

14> radius.SetElement(1, 5)

15> median.SetRadius(radius)
```

Note that the `SetElement()` method doesn't check the bound of the object, and thus is unsafe. The following code is executed, and can lead to a segmentation fault.

```
16> radius.SetElement(1000, 5)
```

A more safe and convenient way to do that, if you have installed the optional patches, is to use the standard python list interface.

```
17> radius[0] = 3

18> radius[1] = 5
```

This time, a bound check is performed, and the user is not able to use an invalid index.

```
20> radius[2] = 1
---------------------------------------------------------------------------
exceptions.IndexError                         Traceback (most recent call last)

/home/glehmann/src/contrib-itk/regionalExtrema/<ipython console>

/home/glehmann/src/contrib-itk/regionalExtrema/itkSize.py in __setitem__(*args)

IndexError: /usr/include/InsightToolkit/Common/itkSize.h:202:
itk::ERROR: Size: index out of range
```

Even if it’s a safe method, it is still not really convenient.

Instead of using `Size` object, it is possible to use python sequences, like lists and tuples.

```
21> median.SetRadius([3, 5])

22> median.SetRadius((3, 5))
```

Also, with the optional patches, some itk object can be converted to python sequences.

```
22> median.GetRadius()
22> <C itk::Size<(2)> instance at _58b40f09_p_itk__SizeT2_t>

23> list(median.GetRadius())
23> [3, 5]

24> tuple(median.GetRadius())
24> (3, 5)
```

To set the same radius for all dimensions, it is possible to use the * python sequence operator - that way, it is possible to write code independent of dimension.

```
25> median.SetRadius( [3]*2 )
```

Or a simple number can also be used.

```
26> median.SetRadius( 3 )
```

Here is the list of itk classes which can currently be substituted by python sequences:

- `CovariantVector`
- `FixedArray`
- `Index`
- `Offset`
- `Size`
- `Vector`
- `ContinuousIndex`

### Python specific functions in the *itk* module

Some convenient functions are provided with the itk module. They all begin with a lower case character to clearly show they are not part of ITK.

- `itk.image(object)` try to return an image from the object given in parameter. If the object is an image, it is returned without changes. If the object is a filter, the object returned by `GetOutput()` method is returned. This function is used in most of the next functions to allow the user to pass an image or a filter, and is available here for the same usage in some custom fuctions.
- `itk.range(object)` return the range of values of an image in a tuple. `object` can be an image or a filter. In case of a filter, `itk.image()` is used to get the output image of the filter. The function update the pipeline by calling `UpdateOutputInformation()` and `Update()`.

  This function is only a convenient function for a common task while prototyping.

  Example:

  ```
  1> import itk

  2> reader = itk.ImageFileReader.IUC2.New(FileName="cthead1.png")

  3> itk.range(reader)
  3> (0, 255)
  ```

- `itk.size(object)` return the size of an image. `object` can be an image or a filter. In case of a filter, `itk.image()` is used to get the output image of the filter. The function update only the information of the pipeline, by calling `UpdateOutputInformation()`, but don't trigger a full update of the pipeline.

  This function is only a convenient function for a common task while prototyping.

  Example:

```
4> itk.size(reader)
4> <C itk::Size<(2)> instance at _d40b8a09_p_itk__SizeT2_t>

5> print itk.size(reader)
<Size [256, 256]>

6> list(itk.size(reader))
6> [256, 256]
```

Note that commands 5 and 6 can be used only with the optional patches.

- `itk.template(object)` returns the template class and parameters of a class or instance of this class.

  Example:

  ```
  7> itk.template(reader)
  7> (<itkTemplate itk::ImageFileReader>, (<class 'itkImage.itkImageUC2'>,))
  ```

- `itk.write(object, fileName)` write an image in a file, without having to pass the image type. `object` can be an image or a filter. In case of a filter, `itk.image()` is used to get the output image of the filter. The function update the pipeline by calling `UpdateOutputInformation()` and `Update()`.

  This function is only a convenient function for a common task while prototyping.

  Example:

  ```
  8> itk.write(reader, 'out.png')
  ```

- `itk.show()`, `itk.show2D()` and `itk.show3D()` are used to display images. `itk.show2D()` requires to have imview [12] installed. `itk.show3D()` requires to have Vtk for python, ItkVtkGlue[2] for python, PyQt [13], and iPython [14] installed, and to use iPython with the `-qthread` option.

  `itk.show()` call the best viewer according to the image type.

  `itk.show2D()` can be called with a 3D image as parameter to show the image slice by slice.

  `itk.show3D()` display a volumic rendering of the image. See Figure 1.

- `itk.strel(d, s)` is used to create a binary ball structuring of dimension `d` and size `s`. Structuring element support is quite bad currently in WrapITK and should change in the future. Using `itk.strel` rather than creating a `BinaryBallStructuringElement` directly is recommended to have backward compatibility when the structuring element type will be changed.

- `itk.auto_progress(b)` is used to automatically add a progress report to all the newly created filters. `b` must be `True` or `False`. If `b` is true, something like

  ```
  9> median.Update()
  itkMedianImageFilterIF2IF2: 0.109990
  ```

  is displayed on the standard output. While prototyping, it is a convenient way for the user to know if the execution time will be short or if he can do something more useful [3] than waiting for the filter to complete.

[2] ItkVtkGlue can be found in the *ExternalProject* directory of WrapITK

[3] like having a cup of tea

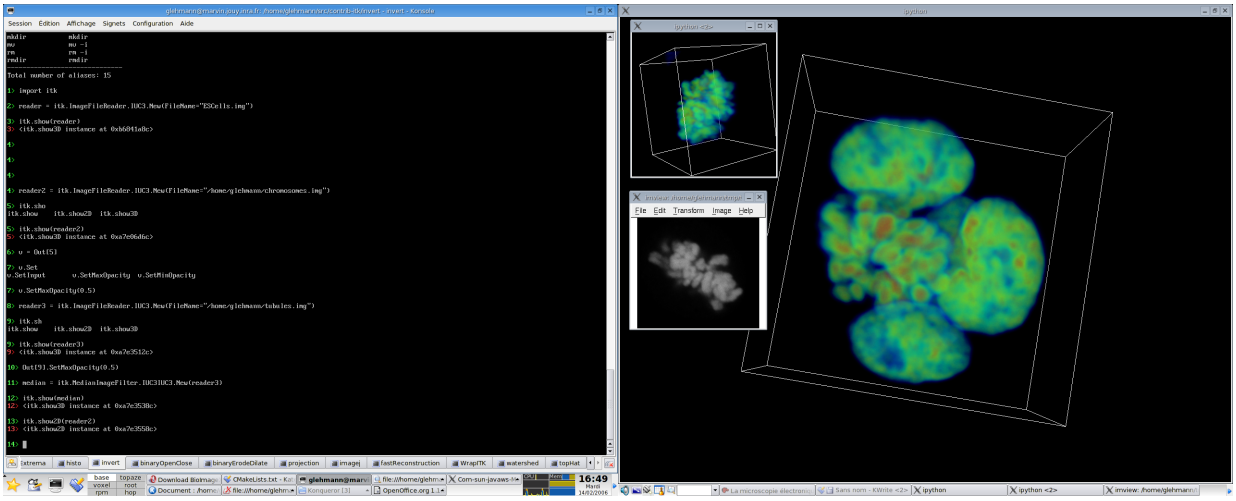

Figure 1: A screenshot of WrapITK in action with python.

- `itk.class_(object)` return the class of an object. The `__class__` attribute is often not what the user want with ITK. `itk.class_` is a convenient function to get the class of an ITK object.

  Note that it is called `class_` and not `class`, because `class` is a reserved word in python.

  Example:

  ```
  10> median.__class__
  10> <class 'itkMedianImageFilter.itkMedianImageFilterIF2IF2_PointerPtr'>

  11> itk.class_(median)
  11> <class 'itkMedianImageFilter.itkMedianImageFilterIF2IF2'>
  ```

### Advanced Features

As an extra bonus, it is possible to view the doxygen documentation for each class as the python docstring. This string is available as:

```
print itk.Image.__doc__
```

or even better (if you use iPython)

```
itk.Image?
```

Several steps are necessary to obtain this nirvana, however. First, when configuring the build in ccmake, you must set `DOXYGEN_MAN_PATH` to some directory where man pages for the ITK classes will be created. Then, after the build, you must run `make_doxygen_config.py` from within the `Python` directory in the build directory, to collect information about the wrapped classes and create a doxygen configuration file to make these man pages. Finally, run doxygen with that configuration file. After these three simple steps, class docstrings will contain the man page information. Note that this is limited to systems which support the python `commands` module, and which have `groff` in the path. This basically means anything but windows [15] will work. (Cygwin should work too.)

In addition (as mentioned above), WrapITK by default ensures that no bare pointers are ever returned to python: instead reference-counting `SmartPointer`s are used. However, there may be times when extracting a bare pointer or

creating a new `SmartPointer` is necessary. To get a bare pointer from a smart pointer, use the `GetPointer()` method, as in ITK proper. To create a new smart pointer, the `SmartPointer` template proxy class can be used just as above:

```
smartPtr = itk.SmartPointer[itk.Image[itk.US, 2]](image.GetPointer())
```

or just

```
smartPtr = itk.SmartPointer[image](image.GetPointer())
```

`itk` module can be very long to import. The `itkConfig` module define a `ImportCallback` method which will be called when each sub module is imported in the import process. `ImportCallback` can be customized to report the progress status of the import process. It must be a function that can take the name of the library being imported as a parameter. Here is an example of a very basic callback function which display the name of the submodule being imported on the standard error output.

```
import sys, itkConfig
def stderr_callback(name):
  print >> sys.stderr, name
itkConfig.ImportCallback = stderr_callback
import itk
```

### 2.3 TCL usage

Write me.

### 2.4 Java usage

Write me.

## 3 Developer guide

What follows is a brief description of how the WrapITK build system works, how it can be extended, and how to write external projects.

### 3.1 WrapITK description

#### Creating a CMakeLists.txt file for a wrapper library

Each WrapITK sub-library (e.g. `ITKCommonA`, or `ITKAlgorithms`) lives in a sub-directory of the WrapITK project (e.g. `CommonA` or `Algorithms`) with a `CMakeLists.txt` file that describes how that library and language support files (e.g. python template definitions) is to be created. Moreover, any external project will need a similar file to describe how to create that library.

See `SampleCMakeLists.txt` in this directory for a description of each macro and option that can appear in such a file. What follows is the usual set of commands that will appear:

```
BEGIN_WRAPPER_LIBRARY("MySpatialObjectExtensions")
SET(WRAPPER_LIBRARY_DEPENDS ITKSpatialObject ITKCommonA)
SET(WRAPPER_LIBRARY_LINK_LIBRARIES ITKCommon)
WRAPPER_LIBRARY_CREATE_WRAP_FILES()
WRAPPER_LIBRARY_CREATE_LIBRARY()
```

- BEGIN_WRAPPER_LIBRARY() sets up the environment to wrap a set of classes into a library with a given name. This macro is defined in ConfigureWrapping.cmake WRAPPER_LIBRARY_DEPENDS stores the list of WrapITK libraries on which the current library depends (e.g. which libraries wrap classes like Image or SpatialObject, that are going to be used in the current library). Every project should at least depend on ITKCommonA.
- WRAPPER_LIBRARY_LINK_LIBRARIES stores a set of other libraries to add at link time. This can be 3rd party libraries that you will use (be sure to properly set LINK_DIRECTORIES in this case), or more commonly, the ITK libraries that need to be linked in, like ITKCommon, ITKIO, or other.
- WRAPPER_LIBRARY_CREATE_WRAP_FILES() scans all of the wrap_XXX.cmake files in the current directory and uses the directives within to create CableSwig input files for these classes. Information about template instantiations is also recorded for the language support files that are created next. This macro is defined in CreateCableSwigInputs.cmake, and calls language support macros from CreateLanguageSupport.cmake.
- Finally, WRAPPER_LIBRARY_CREATE_LIBRARY() creates rules to parse the CableSwig inputs and compile a wrapper library. This macro also causes various language support files to be created (python only currently) which make it easy to load that library in python, and which know about the template instances defined. This macro is defined in CreateWrapperLibrary.cmake, and calls language support macros from CreateLanguageSupport.cmake.

### Creating wrapXXX.cmake files to wrap classes

A wrap_XXX.cmake file defines a group of classes and/or template instantiations to be wrapped. Often one such file is defined for each class wrapped, but this is not strictly necessary.

Within such a file, directives are issued to wrap classes and particular template instances.

WrapITK define several macros and variable designed to:

- make creation of wrappers easy. The syntax is simple enough to get in quickly.
- make choice of template arguments explicit. It should be easy to understand the idea of the author of a wrapper by reading the file.
- support mostly transparently the dimensions and types chosen by the user.

The most common case should be to create a new wrapper for a simple image filter, like MedianImageFilter. Let see that example in details.

Here is the BasicFiltersB/wrap_itkMedianImageFilter.cmake file:

```
WRAP_CLASS("itk::MedianImageFilter" POINTER)
  WRAP_IMAGE_FILTER_INT(2)
  WRAP_IMAGE_FILTER_SIGN_INT(2)
  WRAP_IMAGE_FILTER_REAL(2)
END_WRAP_CLASS()
```

The file contains a WRAP_CLASS - END_WRAP_CLASS block, which itself contains some WRAP_IMAGE_FILTER_* macros. WRAP_CLASS("itk::MedianImageFilter" POINTER) begin the wrapping of the itk::MedianImageFilter templated class. The name of the class must be fully qualified. The option POINTER indicate that the object of the class can be manipulated with a SmartPointer, and that the SmartPointer specialization for the class itk::MedianImageFilter must be created.

Then, several WRAP_IMAGE_FILTER_* macros are called. They are convenient macro to create wrapper for classes which take only image types as template arguments. The parameter, here 2, give the number of required template arguments. The two image types used as template parameter are the same.

All of the available directives are defined and documented in CreateCableSwigInputs.cmake. The basics are presented here:

- WRAP_CLASS("fully_qualified::ClassName" [POINTER|POINTER_WITH_SUPERCLASS]) causes a templated class to be wrapped. All namespaces must be included in the class name, and note that no template instantiation is given. Template instantiations are created with various WRAP directives, described below, between invocations of WRAP_CLASS() and END_WRAP_CLASS().

  WRAP_CLASS("itk::ImageFilter") issues an implicit call to WRAP_INCLUDE("itkImageFilter.h"), so the header for the wrapped class itself does not need to be manually included. To disable this behavior, set WRAPPER_AUTO_INCLUDE_HEADERS to OFF.

  The final optional parameter to WRAP_CLASS is POINTER or POINTER_WITH_SUPERCLASS. If no options are passed, then the class is wrapped as-is. If POINTER is passed, then the class and the typedef'd class::Pointer type is wrapped. (Class::Pointer had better be a SmartPointer instantiation, or things won't work. This is always the case for ITK-style code.) If POINTER_WITH_SUPERCLASS is provided, then class::Pointer, class::Superclass and class::Superclass::Pointer are all wrapped. (Again, this only works for ITK-style code where the class has a typedef'd Superclass, and the superclass has Self and Pointer typedefs). POINTER_WITH_SUPERCLASS is especially useful for wrapping classes whose superclasses depend on the template definitions of the given filter. E.g. any of the functor image filters, which define totally different superclass template parameters depending on which functor is used.

- END_WRAP_CLASS() – end a block of template instantiations for a particular class.

- WRAP_INCLUDE("header.h"). By default, itkMedianImageFilter.h is included when the causes the itk::MedianImageFilter is wrapped, and this behavior is most of the time enough. If it not enough, this macro can be used to include some specific files.

- WRAPPER_AUTO_INCLUDE_HEADERS. This variable is set to ON by default, but can be set to OFF to disable the auto include feature. This feature should be used when several classes to wrap come from the same header file. WRAPPER_AUTO_INCLUDE_HEADERS is re-set to ON for each new wrap_xxx.cmake file.

- WRAP_TEMPLATE("mangled_suffix" "template parameters"). When issued between WRAP_CLASS and END_WRAP_CLASS, this command causes a particular template instantiation of the current class to be wrapped. The parameter mangled_suffix is a suffix to append to the class's name that uniquely identifies this particular template instantiation, and "template parameters" are whatever should go between the < > template instantiation brackets. (Do not include the brackets.) If you are wrapping a filter, there are simpler macros to use, which are defined at the bottom of CreateCableSwigInputs and described below.

- WRAP_NON_TEMPLATE_CLASS("fully_qualified::ClassName" [POINTER|POINTER_WITH_SUPERCLASS]). Same as WRAP_CLASS, but creates a wrapper for a non-templated class. No END_WRAP_CLASS() is necessary after this macro because there is no block of template instantiating commands to close.

WrapITK define some lists which group the types and dimensions. Those list can be used by the developer to create a wrappers but must *never* be modified.

- WRAP_ITK_DIMS contains all the dimensions selected by the user.
- WRAP_ITK_INT contains all unsigned integer types selected by the user.
- WRAP_ITK_SIGN_INT contains all signed integer types selected by the user.
- WRAP_ITK_INTEGRAL contains all signed and unsigned integral types selected by the user.
- WRAP_ITK_REAL contains all the real types selected by the user.
- WRAP_ITK_SCALAR contains all the scalar types selected by the user.
- WRAP_ITK_RGB contains all the RGB types selected by the user.
- WRAP_ITK_VECTOR_REAL contains all the Vector types selected by the user.
- WRAP_ITK_COV_VECTOR_REAL contains all the CovariantVector types selected by the user.
- WRAP_ITK_VECTOR contains all the Vector and CovariantVector types selected by the user.

- WRAP_ITK_ALL_TYPES contains all the types selected by the user.
- SMALLER_THAN_D contains all the types ”smaller” than double selected by the user. This variable is useful when a filter decrease the range of pixel value, like BinaryThresholdImageFilter.
- SMALLER_THAN_UL contains all the types ”smaller” than unsigned long selected by the user.
- SMALLER_THAN_US contains all the types ”smaller” than unsigned short selected by the user.
- SMALLER_THAN_SL contains all the types ”smaller” than signed long selected by the user.
- SMALLER_THAN_SS contains all the types ”smaller” than signed short selected by the user.

WrapITK provides some macros to manipulate those list and use them to create the wrappers. Most of those macros are there to fill a lack of feature to manipulate lists in CMake, and should be replaced by some CMake native commands in the future.

- UNIQUE()
- SORT()
- INTERSECTION()
- FILTER()
- FILTER_DIMS()
- INCREMENT()
- DECREMENT()

Some convenient macros are available to wrap image filters.

These macros often take an optional second parameter which is a ”dimensionality condition” to restrict the dimensions that the filter will be instantiated for. The condition can either be a single number indicating the one dimension allowed, a list of dimensions that are allowed (either as a single -delimited string or just a set of separate parameters), or something of the form n+ (where n is a number) indicating that instantiations are allowed for dimension n and above.

- WRAP_IMAGE_FILTER_type(size) . type can be one of:
  - INT
  - SIGN_INT
  - REAL
  - VECTOR_REAL
  - COV_VECTOR_REAL
  - RGB
  - ALL
  - SCALAR
  - VECTOR

  This macro create a template instantiation with size itk::Image parameters of the given pixel type. So if you are wrapping a filter which should take two images with integral pixel types, write WRAP_IMAGE_FILTER_INT(2). The specific integral data type(s) (char, long, or short in the WRAP_IMAGE_FILTER_INT case) will be determined by the user-selected build parameters (e.g. WRAP_long, and WRAP_short).

- `WRAP_IMAGE_FILTER(param_types param_count)` is a more general macro for wrapping image filters that need one or more image parameters of the same type. The first parameter to this macro is a list of image pixel types for which filter instantiations should be created. The second is a `param_count` parameter which controls how many image template parameters are created. The optional third parameter is a dimensionality condition.

  E.g. `WRAP_IMAGE_FILTER("${WRAP_ITK_ALL}" 2)` will create template instantiations of the filter for every pixel type that the user has selected.

- `WRAP_IMAGE_FILTER_TYPES()`. Creates template instantiations of the current image filter, for all the dimensions selected by the user (or dimensions selected by the user that meet the optional dimensionality condition). This macro takes a variable number of arguments, which should correspond to the image pixel types of the images in the filter's template parameter list. The optional dimensionality condition should be placed in the last parameter.

- `WRAP_IMAGE_FILTER_COMBINATIONS()` takes a variable number of parameters. Each parameter is a list of image pixel types. Filter instantiations are created for every combination of different pixel types in different parameters. A dimensionality condition may be optionally specified as the first parameter. E.g. `WRAP_IMAGE_FILTER_COMBINATIONS("UC;US" "UC;US")` will create: `filter<itk::Image<unsigned char, d>, itk::Image<unsigned char, d> >`, `filter<itk::Image<unsigned char, d>, itk::Image<unsigned short, d> >`, `filter<itk::Image<unsigned short, d>, itk::Image<unsigned char, d> >`, and `filter<itk::Image<unsigned short, d>, itk::Image<unsigned short, d> >` where `d` is the image dimension, for each selected image dimension.

## 3.2 Extending or customizing WrapITK

To minimize build times and library size, it is possible to manually prevent various classes from being wrapped. WrapITK is divided into several sub-libraries, each with a sub-directory: `Algorithms`, `BasicFilters[ABC]`, `Common[AB]`, `IO`, `Numerics`, `SpatialObject`, and `VXLNumerics`. Within these directories are sets or `wrap_XXX.cmake` files, where `XXX` is the name of the class (or set of classes) to be wrapped. To prevent one of these classes from being wrapped, simply rename the file to anything that does *not* start with `wrap_` and end with cmake. (E.g. append `.notwrapped` to the name.) (This is probably unsafe to do in the `Common`, `Numerics`, or `IO` directories.)

To add classes to be wrapped, it is recommended that you create a simple *External Project* described below. If this is out of the question, you could create additional `wrap_XXX.cmake` files in the appropriate directory. (Read on for instructions as to what to put in these files.)

## 3.3 External projects

### Why external projects?

External projects let the developer access some custom class with the target languages and is a powerful way to extend WrapITK, test new wrapper, wrap more types, etc. A nice side effect of wrappers, for contributions[4] for example, is to build *all* the methods of the wrapped classes, and so to be sure everything builds as it should [5].

[4] A nice template for contributions to the Insight Journal [16] which include the template code to build wrappers is available at http://voxel.jouy.inra.fr/darcs/contrib-itk/template/. Just use the command *darcs get http://voxel.jouy.inra.fr/darcs/contrib-itk/template/ contribName* and edit the project name in the *CMakeLists.txt* file to begin your new contribution.

[5] We have found and fixed numbers of bug in ITK while adding more classes to WrapITK

### Building

To build an external project, first ensure that WrapITK has been properly built. Then use `ccmake` to configure a build directory for the external project. If WrapITK has not been installed, you will have to manually enter the path to the WrapITK build directory.

By default, the build options are the same than the one used for building WrapITK, but can be modified in the advanced options.

### Usage

Once an external project has been built, it can be tested directly from the build tree. Start python in the external project build directory's Python subdirectory, and run the command `import ProjectConfig` (or `import ProjectConfig-[Debug|Release|...]` if you were using an IDE, depending on which build configuration was set from the IDE). This command sets up the search paths properly so that WrapITK and the newly-created library files can be found. Then type `import ...` (where `...` is replaced with the name of the external project; e.g. `import BufferConversion`), and use the project.

### Installation

Simply type `make install` (or run your IDE's install step) to install the external project into the WrapITK tree (provided WrapITK has already been installed). Now the external project can be used just like any of the other WrapITK libraries, and it will be imported into the `itk` namespace when the `import itk` command is issued from Python. (This can be disabled by setting `WRAPPER_LIBRARY_AUTO_LOAD` to `OFF` in the external project's `CMakeLists.txt`.)

### Top-level CMakeLists for external projects

In addition to having a set of `wrap_XXX.cmake` files and the proper commands to read in these files and create a library (all described above), an external project's CMakeLists file needs at least one additional command to start it out: `FIND_PACKAGE(WrapITK REQUIRED)`.

This command will cause cmake to try to find the WrapITK build/install directory. If WrapITK has been installed, this will work on the first try. Otherwise, you will have to set (within ccmake, or in the CMakeLists if you prefer) the variable `WrapITK_DIR` to contain the path to the WrapITK build directory.

### Examples

In `WrapITK/ExternalProjects` there are several sample "External Projects" that can be built to provide additional functionality to WrapITK and to serve as a demonstration for how to create your own such projects. One project is an ITK-VTK [17] bridge, and the other is a Python class to allow conversion from Numeric/Numarray/numpy [18, 19, 20] matrices to ITK images (and vice-versa).

More examples can be found in the contributions to the Insight Journal [16], or directly at `http://voxel.jouy.inra.fr/darcsweb/`.

### BufferConversion: an example of extension for one language

Write me.

#### ItkVtkGlue: an example of extension for all languages, including C++

ItkVtkGlue wrap the classes used to convert data from ITK to VTK [17] and from VTK to ITK. Those classes comes from the InsightApplications [1], and make the conversion as simple in python as in C++. It has been tested with VTK 5.0.0.

By default, those classes are not loaded in the `itk` module to avoid loading all the `vtk` module while importing `itk`. The command `import itkvtk` will load the module, and add the classes both in `itkvtk` and `itk` module. The classes are then usable as the other itk classes.

Example:

```
1> import itk

2> import itkvtk

3> reader = itk.ImageFileReader.IUC3.New()

4> converter = itk.ImageToVTKImageFilter.IUC3.New(reader)

5> converter.GetOutput()
5> <libvtkFilteringPython.vtkImageData vtkobject at 0xb7675b60>
```

The `itk.show3D` class use the class `ImageToVTKImageFilter` to create the volume rendering shown in Figure 1.

### 3.4 Extending language support and adding more languages

Write me.

#### Generating target language code

Write me.

#### typemaps

Write me.

### 3.5 Contributing to WrapITK

WrapITK is an opensource project, and so all contributions are welcome. Here are some points which requires special attention:

- Test it and report problem. That's the more important thing to do: we need feedbacks to enhance WrapITK quality !
- Work on tcl, java, and others. We are not tcl or java developers, and so are not able to complete the work for those languages. Any help from tcl and java expert would be highly appreciated. Also, there is no reason to be limited to python, tcl and java, and WrapITK can be extended to other languages supported by swig like perl [21], ruby [22], ocaml [23] and others.
- Add more classes. WrapITK add lots of new classes compared to the current wrapping system, but there is still lots of work to do, especially to support more filter dedicated to `Vector` pixels.

`darcs` [11] allow to easily contribute to WrapITK, by sending patch by email with the command `darcs send`, while keeping credits for the work done. Feel free to send patches; they will be tested and integrated in the project.

A web interface [24] for the WrapITK's darcs repository is available at `http://voxel.jouy.inra.fr/darcsweb/`.

## 4 Acknowledgments

I thank Dr Pierre Adenot and all the *Embryon et Biotechnologie* team for their patience during the long development time before getting a tool really usable.

We would like to thank Charl P. Botha for is help to debug WrapITK on windows plateform, and for the patches he has contributed, as well as Richard Beare for his early interest in using itk with python, for his useful feedbacks, and for his work on buffer conversion in python.

We thank Brad King for his assistance during the development process.

Finally, we thank the ITK developers for the great tool which is ITK, and for the previous work done on wrapping system - without it WrapITK would not be there.

## 5 Conclusion

ITK is a great library, with the drawback to be nearly unusable for prototyping, and to have a poor support for other languages than C++. WrapITK address those issue and finally give to ITK a quite good support for python. We hope that support will soon be extended to tcl and java.

## References

[1] `http://www.itk.org`. (document), 3.3

[2] `http://www.python.org`. (document), 1

[3] `http://www.tcl.tk`. (document), 1

[4] `http://java.sun.com`. (document), 1

[5] `http://www.cmake.org`. 1

[6] `http://www.gccxml.org`. 1

[7] `http://www.swig.org/`. 1

[8] `http://www.itk.org/HTML/CableSwig.html`. 1

[9] `http://www.apple.com/macosx`. 1

[10] `http://www.mandriva.com`. 1

[11] `http://www.darcs.net`. 2.1, 3.5

[12] `http://www.cmis.csiro.au/Hugues.Talbot/imview/`. 2.2

[13] `http://www.riverbankcomputing.co.uk/pyqt`. 2.2

[14] `http://ipython.scipy.org`. 2.2

[15] `http://www.microsoft.com/windows`. 2.2

[16] `http://www.insight-journal.org`. 4, 3.3

[17] `http://www.vtk.org`. 3.3, 3.3

[18] http://numeric.scipy.org. 3.3

[19] http://www.stsci.edu/resources/software_hardware/numarray. 3.3

[20] http://www.numpy.org. 3.3

[21] http://www.perl.org. 3.5

[22] http://www.ruby-lang.org. 3.5

[23] http://caml.inria.fr. 3.5

[24] http://auriga.wearlab.de/~alb/darcsweb/. 3.5

[25] L. Ibanez and W. Schroeder. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-10-6, http://www.itk.org/ItkSoftwareGuide.pdf, 2003.