SimITK: Rapid ITK Prototyping Using the Simulink Visual Programming Environment

A. W. L. Dickinson[1], P. Mousavi[1], D. G. Gobbi[2], and P. Abolmaesumi[1,3]

[1]School of Computing, Queen's University, Kingston, Ontario, Canada
[2]Atamai Inc., Calgary, Alberta, Canada
[3]Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, Canada

ABSTRACT

The Insight Segmentation and Registration Toolkit (ITK) is a long-established, software package used for image analysis, visualization, and image-guided surgery applications. This package is a collection of C++ libraries, that can pose usability problems for users without C++ programming experience. To bridge the gap between the programming complexities and the required learning curve of ITK, we present a higher-level visual programming environment that represents ITK methods and classes by wrapping them into "blocks" within MATLAB's visual programming environment, Simulink. These blocks can be connected to form workflows: visual schematics that closely represent the structure of a C++ program. Due to the heavily C++ templated nature of ITK, direct interaction between Simulink and ITK requires an intermediary to convert their respective datatypes and allow intercommunication. We have developed a "Virtual Block" that serves as an intermediate wrapper around the ITK class and is responsible for resolving the templated datatypes used by ITK to native types used by Simulink. Presently, the wrapping procedure for SimITK is semi-automatic in that it requires XML descriptions of the ITK classes as a starting point, as this data is used to create all other necessary integration files. The generation of all source code and object code from the XML is done automatically by a CMake build script that yields Simulink blocks as the final result. An example 3D segmentation workflow using cranial-CT data as well as a 3D MR-to-CT registration workflow are presented as a proof-of-concept.

Keywords: Visual Programming, ITK, MATLAB, Simulink, CMake, XML.

1. INTRODUCTION

In medical image computing and computer-assisted interventions, common image manipulation and processing techniques such as segmentation, registration, and visualization are frequently employed. Many related algorithms have been implemented and included in open-source software libraries, mainly written in C++ and available for download. The Insight Segmentation and Registration Toolkit[1] (ITK) is an example of such a library, and is primarily composed of algorithms for registering and segmenting multi-dimensional data. ITK's cross-platform nature has stimulated much interest in the medical imaging community, as it is an ideal library to solve medical image analysis problems without having to deal with the algorithms' implementation details. Although ITK is often used directly as a C++ library, higher-level interfaces can be useful where the toolkit user either lacks knowledge of C++, or requires a more rapid development schedule than C++ allows. Such interfaces in the literature have taken one of three forms: embedding/wrapping the language within a scripting language like Python or Tcl, thus removing the necessity for potentially complex C++ programming in favour of writing scripts[2]; general-purpose applications that use a GUI to provide access to a broad range of the toolkit functionalities[3,4] or give deep functionality to one specific area of the toolkit[4]; and visual programming environments that represent the various classes within the toolkit as 'blocks' that can be connected to one another graphically in the same way the code would be written[5].

Medical Imaging 2011: Visualization, Image-Guided Procedures, and Modeling,
edited by Kenneth H. Wong, David R. Holmes III, Proc. of SPIE Vol. 7964, 796438

We propose SimITK, a visual programming abstraction approach, that wraps ITK into the Simulink visual programming environment within the MATLAB^{TM}(Natick, MA) scientific computing environment. Simulink was selected as the visual programming component for this project since MATLAB is used heavily within the scientific computing community. Thus, our solution is made instantly familiar to a large user base. Furthermore, as Simulink is tightly integrated with the MATLAB programming environment, it is possible to execute MATLAB scripts in conjunction with SimITK and to move images between these two environments. In SimITK, like Simulink, workflows are represented as a diagram of a connected series of 'blocks'. Data is created, represented, modified, or augmented using these blocks; the user can connect them to establish data and object connections between the ITK classes. Parameter modifications are accomplished through double-clicking a block, which presents the user with a dialog box of familiar configuration modifiers (check boxes, radio buttons, and text-entry fields) to easily change desired parameters. When the workflow is executed, the written code representation is automatically generated from these specifications outlined by the user. The ITK images are passed through Simulink as MATLAB arrays, but memory redundancy is avoided by having the ITK memory pointers use the MATLAB arrays directly. In other words, ITK and Simulink share the memory that is used to store the images.

2. METHODOLOGY

The SimITK wrapping procedure can be divided into four distinct steps: XML generation, "Virtual Block" .tpp generation, Simulink .cpp file generation, and MATLAB .mex and .mdl file generation. We first generate an XML description of each ITK class, which contains key information pertaining to each class, such as the class name, parameters, inputs, outputs, and datatypes. Once this representation is created, the rest of the process is automatic. From the XML representation of a given class, a Perl script generates the "Virtual Block" .tpp and Simulink-Function (S-Function) .cpp code, which is then compiled by the MATLAB compiler to create the .mex object file that is executed by Simulink. Another Perl script is used to generate the .mdl model files that collate the .mex files into a Simulink visual block library. CMake is used as the underlying backbone that generates, links, compiles, and builds these necessary files to create SimITK. A visualization of the wrapping methodology is presented in Figure 1.

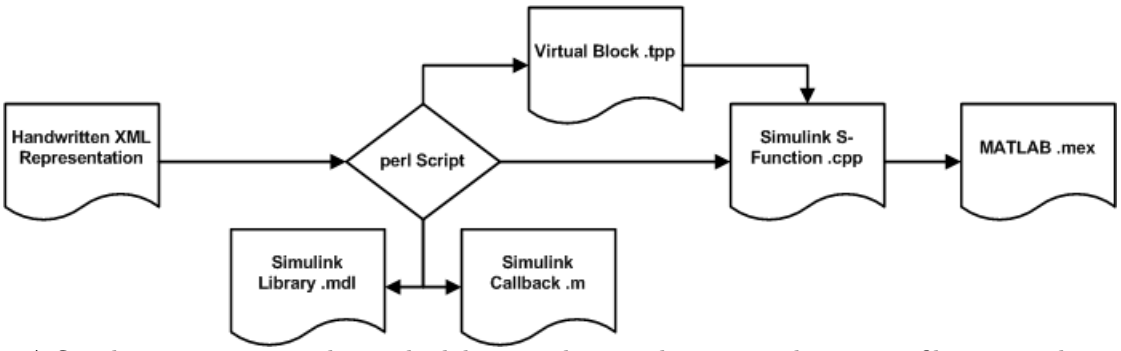


Figure 1: A flowchart representing the methodology implemented to create the various files required to integrate ITK into the Simulink visual programming environment.

The "Virtual Block" .tpp file facilitates intercommunication between the ITK and Simulink workspaces. This intermediary is required because of the heavily templated and type-definitioned nature of ITK; Simulink is unable to interpret these ITK-datatypes without assistance and vice versa. Inputs and parameters are passed into the "Virtual Block" from Simulink, and are reinterpreted as, or converted when necessary into, ITK datatypes and operated upon by the ITK filters. The result from the ITK workspace is converted back into an appropriate Simulink-datatype suitable for output to the next block in the workflow.

In order to successfully implement a given class within Simulink, an S-Function .cpp file containing Simulink-specific execution code is derived and populated with the stored XML information using the Perl script. For certain blocks, such as the Optimizer and Transform blocks, a .m file that contains Simulink-callbacks updates the block representation to change its appearance, should additional inputs or outputs be requested by the user via the block dialog box. It is worth noting that the Simulink .cpp file is responsible only for the execution, not

the graphical representation, of the block within Simulink. The MATLAB .mdl file, discussed below, describes the graphical representation and user interface.

In the final step, the Simulink .cpp file is compiled into a MATLAB .mex file that is be executed at runtime by MATLAB. This representation also needs a graphical block representation as the S-Function is purely the Simulink-executed code. This is also template-based and populated by the information contained within the XML. After all blocks have been built and compiled, all the blocks of the same datatype and dimensionality are assembled into a comprehensive .mdl library file that serves as the block repository for the user to pick and choose the blocks needed to build their workflow.

In addition to the wrapped ITK classes, we have created and included a custom ITK class, `StepByStepRegistrationMethod`, that allows Simulink to introspectively collect information like the metric value, transform parameters, and any other intermediary results. Functionally, the class operates identically to the `itkImageRegistrationMethod` except that the "ticking" of the ITK clock is controlled by the Simulink clock. This was implemented so that the user could retrieve parameters in Simulink, such as the metric value and transform parameters, at the current iteration.

3. APPLICATION

An example workflow demonstrating a segmenation of the skull from cranial CT data is shown in Figure 2. A file reader is used to load the data and pass it to the core of the workflow: a threshold filter that replaces pixels below the pixel-intensity threshold with black, while highlighting the bony areas (intensities above threshold) in white. The modified volume is then written to disk.

A second example performing a registration of MR and CT data of different cranial data is shown in Figure 3. File readers are used to load in the CT and MR data as the inputs to a Centered 3D Euler registration method using Nearest Neighbour interpolation, Mutual Information as the metric, and Gradient Descent to optimize the registration. Following registration, the MR data is resampled using the optimal registration transform parameters before being written to disk as a new file.

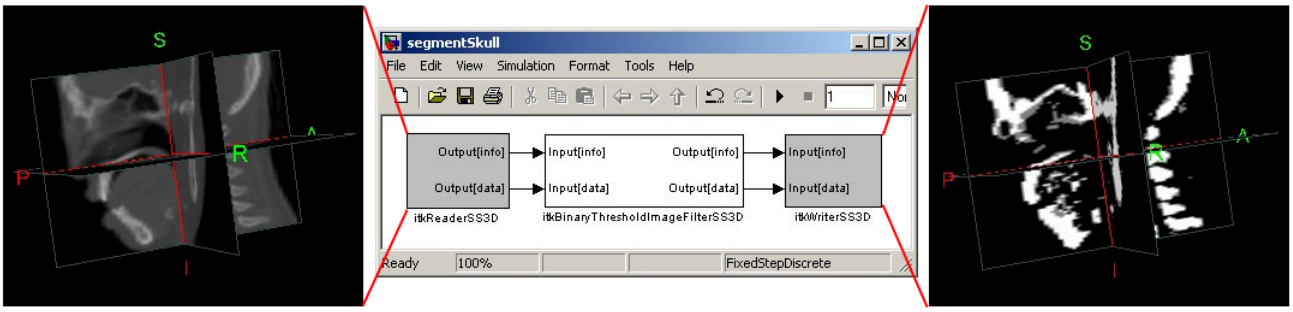


Figure 2: Example of a 3-dimensional segmentation workflow using cranial CT data to highlight areas of bone in SimITK.

4. RESULTS

For reference, all results were computed using an Intel Core 2 Quad CPU Q9400 @ 2.66 GHz with 4 GB of RAM, Windows XP Service Pack 3, ITK 3.18, MATLAB/Simulink R2008b, CMake 2.8 and Visual Studio 2008. All benchmarking times were generated using the same C++ code injected at equivalent points within each respective code to ensure an even footprint for comparison.

With respect to the data, segmentation cranial data can be found within the Examples directory of a base ITK installation and the registration cranial CT (Figure 4a) and MRI data (Figure 5a) can be acquired from the Visible Human Project at http://www.nlm.nih.gov/research/visible/visible_human.html. Figure 5d is the pre-registered MRI volume coloured red while Figure 5e is the registered MRI data coloured green. Figure 5c is an overlay of both Figures 5d and 5e.

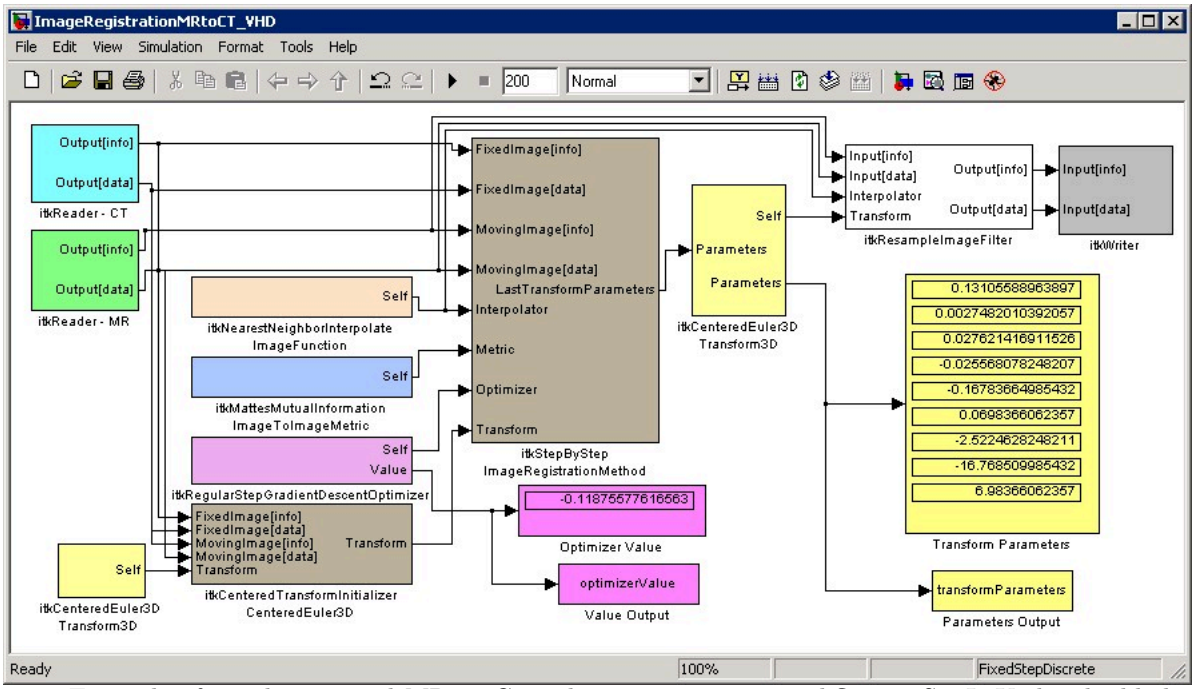
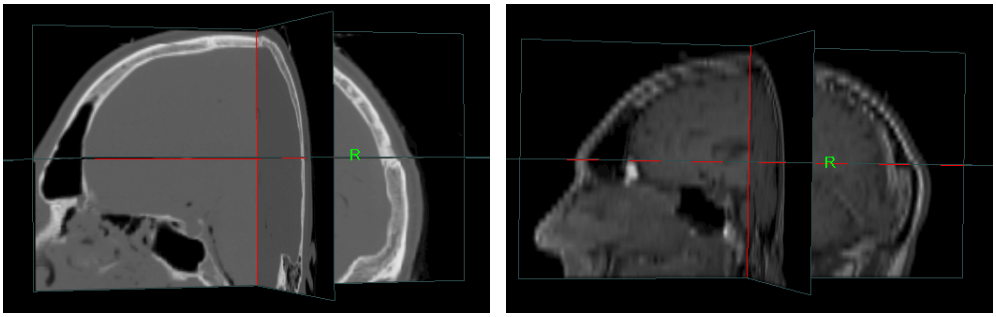


Figure 3: Example of a 3-dimensional MR to CT volume registration workflow in SimITK that highlights the output of various block parameters.



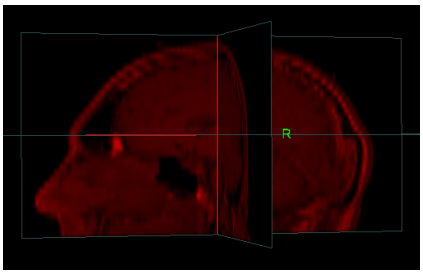
(a) The CT data were processed into a 129 slice (1 mm thickness) .VTK volume file of 245-by-255 pixel images with one pixel equivalent to 0.898438 mm.

(b) The fresh T1 MRI data were processed into a 34 slice (4 mm thickness) .VTK volume file of 128-by-128 pixel images with one pixel equivalent to 1.01562 mm.

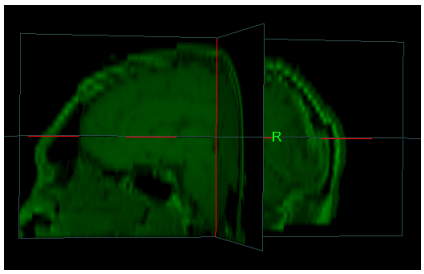
Figure 4: Reference images of the initial CT and MRI data used in the Registration example (Figure 3).

Table 1: Comparison of MRI to CT Registration code runtimes (100 executions timed until completion)

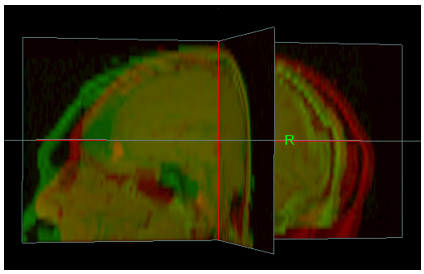
	`itkImageRegistrationMethod`	`StepByStepImageRegistrationMethod`
Pure C++ ITK Program	170 ± 1 seconds	600 ± 1 second
SimITK Workflow	158 ± 1 seconds	450 ± 1 second
Time Difference	12 seconds (8.8%)	150 seconds (25%)



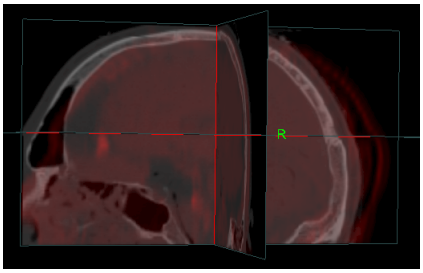
(a) The unregistered MRI data visualized in red.



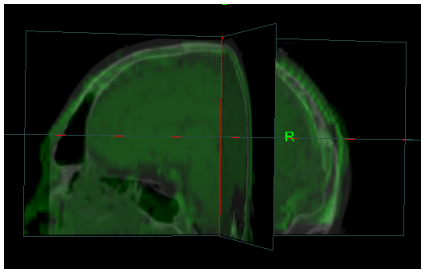
(b) The registered MRI data visualized in green.



(c) An overlay of the MRI data pre- and post-registration.



(d) The CT data with unregistered MRI data overlay in red.



(e) The CT data with final registered MRI data overlay in green.

Figure 5: A comparison of the CT and MRI datasets pre- and post- execution of the registration implementations.

5. DISCUSSION

For the segmentation workflow (Figure 2), the run times were deemed equivalent in both pure, written C++ ITK-code and our rapidly prototyped SimITK flowchart as the difference in runtime was on the order of milliseconds and calculating the time difference would have introduced more overhead than the code execution itself.

The results from our registration example (Figure 3) can be seen in Table 1 which compares pure ITK C++ to SimITK, for both the standard `itkImageRegistrationMethod` and our own custom `StepByStepImageRegistrationMethod`. All the methods ran for the same number of iterations and provided the same final registration parameters. Note that we did not anticipate that the registration time in SimITK would be shorter than in pure ITK, and since we added our timing code for the start and stop of the registration identically in each case, we can only hypothesize that this discrepancy is due to the fact that SimITK uses pre-allocated MATLAB array memory for all image storage, while pure ITK uses system calls to allocate and initialize image memory.

The longer registration times for our `StepByStepRegistrationMethod` compared to the standard `itkImageRegistrationMethod`, on the other hand, were expected. The former runs the registration for a few iterations before returning control to the application (e.g. Simulink) so that introspection can be performed before the registration is resumed. The latter is designed to run the registration to completion in one go – it provides command/observer methods that can be used to do introspection in pure C++ programs but which cannot be used for this purpose within Simulink. On a technical note, `StepByStepImageRegistrationMethod` is currently only compatible with `itkRegularStepGradientDescentBaseOptimizer`-derived optimizer classes as they are capable of starting, stopping, and resuming ITK registrations.

Our goal for SimITK is to create a Simulink wrapper for ITK, where all ITK classes are represented by blocks. Visual workflows could then be created that provide the same power and flexibility as C++ ITK programs but with a fraction of the amount of time required to learn to program ITK in C++. At present, we consider SimITK to be a prototype since only a small (but useful) subset of ITK has been wrapped via a semi-automatic method. We have already achieved fully automatic wrapping for our SimVTK[6] solution, the Visualization Toolkit[7] (VTK) counterpart of this project; however, providing the same degree of automation for wrapping ITK is much more

challenging. In comparison to VTK, which primarily uses vanilla C-language types in its interfaces, ITK defines its own interface datatypes, most of which are templated. Future work will involve automatic generation of XML representations that can handle these specific datatypes by integrating WrapITK[2], an optional ITK build option that generates a complete XML representation of each class (including dependencies and the various templated type-definitions used) that could then be refined to fully automate the SimITK build process.

6. NOVELTY AND CONTRIBUTIONS

The build system of SimITK has also been modularized to take full advantage of the CMake build environment. For our earlier work, the dependency chain in our build scripts was very crude and partial rebuilds were only possible in certain situations. Our current build system is much cleaner, allowing for a much faster debug/rebuild cycle (which is of extreme importance given the size of ITK). The SimITK software is available for free download at http://media.cs.queensu.ca/SimITKVTK/.

Compared to our previous preliminary report[6], we have demonstrated that SimITK is capable of 3D image analysis. Furthermore, our implementation has been expanded to provide direct introspection of the transform parameters during any optimization process such as ITK registration (see Figure 3). Parameters can be graphed in real time as the registration progresses, or they can be saved to a MATLAB array for future analysis.

7. CONCLUSIONS

This release of SimITK demonstrates thorough proof-of-concept that this wrapping methodology can successfully create workflows that parallel the functionality of a pure C++ ITK program while incurring no additional time-overhead. SimITK is still in the prototype stage, as XML descriptions of a small subset of ITK classes are created. Our current efforts are directed towards leveraging WrapITK to automatically generate the XML descriptions, allowing the full automated generation of Simulink blocks for most of ITK.

REFERENCES

[1] L. Ibanez, W. Schroeder, L. N. J. C., [*ITK Software Guide: The Insight Segmentation and Registration Toolkit (Version 2.4)*], Kitware Inc., Clifton Park, New York (2005).

[2] Lehmann G., Pincus Z., R. B., "WrapITK: Enhanced languages support for the Insight Toolkit," *The Insight Journal - 2006 January - June* (2006).

[3] Gering, D. T., Nabavi, A., Kikinis, R., Hata, N., O'Donnell, L. J., Grimson, W. E. L., Jolesz, F. A., Black, P. M., and III, W. M. W., "An integrated visualization system for surgical planning and guidance using image fusion and an open mr," *Journal of Magnetic Resonance Imaging* **13**(6), 967–975 (2001).

[4] Yushkevich, P., Piven, J., Cody, H., Sean Ho, J. C. G., and Gerig, G., "User-guided level set segmentation of anatomical structures with itk-snap," *The Insight Journal - 2005 MICCAI Open-Source Workshop* (2005).

[5] Paladini, G. and Azar, F. S., "An extensible imaging platform for optical imaging applications," *Multimodal Biomedical Imaging IV* **7171**(1), 717108, SPIE (2009).

[6] Gobbi, D., Mousavi, P., Li, K., Xiang, J., Campigotto, A., Fichtinger, G., and Abolmaesumi, P., "Simulink Libraries for Visual Programming of VTK and ITK," *The MIDAS Journal - Systems and Architectures for Computer Assisted Interventions (MICCAI 2008 Workshop)* (2008).

[7] Kitware Inc., [*VTK Users Guide Version 5*], Kitware Inc., 5th ed. (September 2006).