SimITK: Visual Programming of the ITK Image Processing Library within Simulink

by

Andrew William Laird Dickinson

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
September 2011

Copyright © Andrew William Laird Dickinson, 2011
Abstract

The Insight Segmentation and Registration Toolkit (ITK) is a long-established image processing library used for image analysis, visualisation, and image-guided surgery applications. ITK is a collection of C++ classes that can potentially pose usability problems for users without appropriate C++ programming experience. In order to remove the programming complexities and facilitate rapid prototyping, an implementation of ITK within a higher-level visual programming environment is presented: SimITK. ITK functionalities are automatically wrapped into “blocks” within the visual programming environment of MATLAB, Simulink, where these blocks can be connected to form workflows: visual schematics that closely represent the structure of a C++ program.

The heavily C++ templated nature of ITK does not facilitate direct interaction between Simulink and ITK; an intermediary is required to convert respective datatypes and allow intercommunication. As such, a SimITK “Virtual Block” has been developed that serves as a wrapper around the ITK class responsible for resolving the datatypes used by ITK to native types used by Simulink. Part of this challenge surrounds the automated capturing and storage of the pertinent class information (name, inputs/outputs, acceptable datatypes, and allowed dimensionalities) that needs to be reflected within the final block representation. The WrapITK package, included with
ITK, serves to generate initial class representations as complex eXtended Markup Language (XML) files that are consolidated and refined to organise the information into an easily accessible structure when extracting during wrapping. Once refined, the data are transferred into custom-written SimITK-templates - one template for each required filetype - through a series of custom-keyword substitutions that replace special keywords with appropriately retrieved XML information and/or programming code.

The primary result from the SimITK wrapping procedure is the generation of multiple Simulink Block libraries. From these libraries, blocks are selected and interconnected to demonstrate case study examples: a 3D segmentation workflow using cranial-CT data and a 3D MRI-to-CT registration workflow. Suggestions for future development are included as well as several appendices containing a list of classes, code comparisons between ITK C++ and SimITK workflow, installation documentation (both user and developer), as well as example file templates used to integrate ITK within Simulink.
Statement of Co-Authorship

The work presented in this thesis was accomplished under the supervision of Dr. Parvin Mousavi and Dr. Purang Abolmaesumi, who provided feedback and corrections to the manuscript. Dr. David Gobbi provided additional supervision, development recommendations, feedback with regards to code development, and corrections to the manuscript.

Along with the previously mentioned co-authors, early work that led to the completion of this thesis was previously presented by the author at SPIE Medical Imaging 2011 [9]. Otherwise, the material presented in this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with standard referencing practices.

Preliminary plans to refine this thesis into a future journal article submission with the previously mentioned co-authors have been established.
Acknowledgments

Parvin and Purang, my supervisors, thank you for introducing me to the world where computers meet medicine. It has truly let me discover what has become one of my deepest passions. The opportunity you provided gave me the encouragement to show off my true colours and for that I will be forever grateful.

David, my mentor, I can’t express my gratitude enough for the help and guidance you’ve provided over the years and for your limitless patience and understanding through my programming “adolescence”. Thank you so very much.

Morgan for supporting me through all my crazy late nights, working weekends, and reminding me that perhaps I just need to eat something and drink less coffee. You’ve been a constant motivator in every facet of my life and a shining beacon of inspiration. We’ve come a long way, baby.

Scott for the constant challenge on the squash court and Paulette for always making me feel at home in your home. Both your care and support have meant a tremendous amount to me.
Victor, Brian, Sacha, Sean, Sahar, Laura, Layan, Mohsen, Andrew Lang, Andrew Murray, and the rest of the Med.i, MedIA, and Perk Labs - you’ve been my academic family these past years. I feel so very fortunate to have had the chance to get to know each and every one of you.

Trogdor the Dustinators and the Disc Jockeys - while most of our time is dedicated to academics, we refuse to let that be the whole story. I’ve looked forward to every intramural match in the pool or on the pitch we’ve ever played: win, lose, rain, or shine. You’ve been the absolute best to break a sweat with.

The School of Computing, my academic home, has provided me a unique environment filled with support and encouragement since my first day. When surrounded by such excellent people, you cannot help but strive for the same excellence within oneself.

My family - Ditch, Maggie, Taylor, Dave, Lauren, Zoë and Liam - thank you for everything, especially for always encouraging me to be “me”. Without your unparalleled support, this journey would have felt much more like an odyssey. I love you all.

To D.O.G., I dedicate this thesis to you in loving memory.

Andrew William Laird Dickinson
August 2011
Glossary

.cpp Source file that is compiled to generate an S-Function .mex file for a Simulink block.

.m MATLAB Callback File containing the code that enables/disables the entry of method parameters within the Mask Dialog.

.mdl Simulink Library File containing multiple SimITK blocks (formed by collecting all the .mdlpart files of a given datatype and dimensionality).

.mdlpart Simulink Mask file containing the code for the dialog box for a given block.

.mex The compiled S-Function .mex file for a Simulink block, which can be loaded by MATLAB.

.tpp Virtual Block file containing code that facilitates communication between ITK and Simulink.

2D 2 Dimensions/Dimensional.

3D 3 Dimensions/Dimensional.

API Application Programming Interface.
Canvas  The blank model file opened in Simulink that blocks can be placed upon to create workflows.

Class Template Instance  The datatype/dimensionality-specific version of a given ITK class represented in XML by WrapITK.

CT  Computed Tomography.

GUI  Graphical User Interface.

Image Registration  The process of taking multiple image datasets and transforming them such that they are all within the same coordinate frame.

Image Segmentation  The process of dividing an image into one, or many, sub-image(s) such that the altered result is easier to interpret or analyse.

ITK  The Insight Segmentation and Registration Toolkit.

Mask Dialog  The dialog box prompted when a user clicks on a Simulink block that facilitates customisation of the block parameters.

MRI  Magnetic Resonance Imaging.

Simulink  Visual Programming Control System Toolbox add-on for MATLAB.

Template Variable  An identifier in the final SimITK XML document that is a stand-in for a value that will vary depending on the datatype or dimensionality required by the specific template instance of the class that the XML document describes.
**Visual Programming** The paradigm of programming where graphical elements are manipulated and interconnected to create schematics that are functionally-equivalent to written code.

**Workflow** A Simulink canvas that has been populated with appropriately-connected SimITK blocks.

**XML** The eXtensible Markup Language used to store the ITK information.
Contents

Abstract i

Statement of Co-Authorship iii

Acknowledgments iv

Glossary vi

Contents ix

List of Figures xii

List of Code Listings xiii

List of Tables xiv

Chapter 1: Introduction 1

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.1.1 ITK: Medical Imaging Library with Pipeline Design . . . . . . 2

1.1.2 Simulink: A Modeling System for Visual Programming . . . . 4

1.2 Proposed Method: SimITK . . . . . . . . . . . . . . . . . . . . . . . 6

1.3 Thesis Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.4 Thesis Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . 8

1.5 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Chapter 2: Background 10

2.1 Dataflow Programming . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.2 Visual Programming . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.3 Image Processing Terminology . . . . . . . . . . . . . . . . . . . . . 14

2.3.1 Segmentation . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.3.2 Registration . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.4 Language Wrapping . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
Bibliography

Appendix A: Simple ITK C++/SimITK Comparison: Segmentation 81

Appendix B: Complex ITK C++/SimITK Comparison: Registration 84

Appendix C: List of Wrapped Classes
C.1 Image Filter Classes ............................................. 90
C.2 Transform Classes .............................................. 95
C.3 Optimizer Classes ............................................... 96

Appendix D: User Installation Documentation
D.1 Installation Prerequisites ....................................... 97
D.2 Installation Instructions ......................................... 97

Appendix E: Developer Installation Documentation
E.1 Installation Prerequisites ....................................... 99
E.2 Installing and Configuring of Components ................. 100
  E.2.1 Build, Compile, Install ITK 3.20 (or later) ............ 100
  E.2.2 WrapITK (if not ITK v3.20) .......................... 101
  E.2.3 Compiling, Building, Installing SimITK .............. 101

Appendix F: SimITK File Examples and Templates
F.1 SimITK XML Examples .......................................... 102
F.2 Filter Virtual Block .tpp Template .......................... 106
F.3 Filter S-Function .cpp Template ............................. 108
F.4 Simulink Filter Mask .mdlpart Template .................... 113
F.5 Simulink Library .mdl Template ............................. 114
F.6 MATLAB Filter Callback .m Template ....................... 118
## List of Figures

1.1 A comparison of pipeline-based ITK programming and graphical programming within Simulink. ........................................... 5

3.1 Graphical overview of the SimITK wrapping procedure ............. 37

4.1 An Example SimITK Library. .............................................. 54
4.2 Placing a block on a blank canvas ..................................... 55
4.3 The steps to establishing an inter-block connection. .................. 56
4.4 Connecting blocks on a canvas ......................................... 57
4.5 Example of a Simulink Mask Dialog .................................... 58
4.6 Block modifications. ....................................................... 59
4.7 3D Cranial Volume Skull Segmentation Workflow ................... 61
4.8 3D Cranial Volume Registration Workflow ........................... 62
4.9 Registration Reference Images ........................................ 63
4.10 Pre- and post-registration execution image comparison .............. 64

A.1 Equivalent SimITK Pixel-Intensity Threshold Workflow. .......... 83

B.1 Equivalent SimITK Registration Workflow ......................... 89
List of Code Listings

3.1 Sample WrapITK XML ........................................... 40
3.2 Final SimWrapITK XML ........................................... 40

A.1 C++ Code for Creation of Pixel-Intensity Threshold Program .... 81

B.1 C++ Code for Creation of an MRI-to-CT Registration Program .... 84

F.1 Sample SimWrapITK Dumpfile Outlining Class Data Hierarchy ... 102
F.2 Sample SimWrapITK Dumpfile Outlining Method Data Hierarchy .. 103
F.3 SimWrapITK itkCannyEdgeDetectionImageFilter XML ............ 104
F.4 SimITK Virtual Block Template .................................. 106
F.5 SimITK Simulink S-Function Template ............................ 108
F.6 SimITK Simulink Block Filter Mask Template .................... 113
F.7 SimITK Simulink Library Template .............................. 114
F.8 SimITK MATLAB Callback Template ............................. 118
List of Tables

3.1 Execution Times of S-Function Methods . . . . . . . . . . . . . . . . 48
3.2 Perl Scripts/Modules and Associated Filetypes . . . . . . . . . . 50

4.1 Comparison of MRI to CT Registration code runtimes (100 executions, each consisting of 100 iterations, timed until completion) . . . . . . . . 66
Chapter 1

Introduction

1.1 Motivation

Medical images are ubiquitously used to aid in diagnosing and treating patients within the healthcare system. Common examples like confirming a fractured bone through capturing a radiograph or determining the gender of a developing foetus through performing an ultrasound scan are performed regularly. The diagnostic value of these images can be further improved by appropriate image processing techniques. In addition to their diagnostic value, medical imaging is also used to develop surgical plans and interventions. For example, by determining minimally-invasive areas for incisions or locations and angles for optimal needle trajectories, from real-time and pre-operative medical images, a higher calibre of care can ideally be delivered to the patient.

Typically, image-guided diagnosis and interventions are performed with the aid of software designed to manipulate and augment the medical images such that regions of interest can be made more clearly visible, surgical plans can be derived, or relationships can be visualised between different sets of data. The Insight Segmentation
1.1. MOTIVATION

and Registration ToolKit (ITK) is an example of such a software commonly used in research and clinical applications and represents one key component of this thesis; the other, discussed in this chapter, is the visual programming environment Simulink.

1.1.1 ITK: Medical Imaging Library with Pipeline Design

ITK [20] is a free, open-source, and cross-platform collection of C++ image processing classes and libraries. Started in 1999, funded by the US National Library of Medicine of the National Institutes of Health (NLM/NIH), the goal of ITK was to develop an open-source registration and segmentation toolkit. Using ITK also grants access to another NLM-funded project, the Visible Human Project: complete, anatomically detailed, three-dimensional representations of the normal male and female human bodies.

Classes in ITK serve as individual functionalities within ITK and are largely broken down into many different types, such as file readers/writers for loading/saving of image data and image filters that perform image processing techniques on loaded data. As of November 2010\(^1\), ITK consisted of 9,693 files composed of 1,152,146 lines of code. Of these files, approximately 2,700 are ITK-functionality files representing a third of all of ITK; the remaining files aid in the building and installing of ITK or provide documentation.

The classes are divided into a hierarchy\(^2\) that provides a taxonomy based on functionality. The largest branch is the “Process Object” branch where most classes used to perform image processing can be found. Examples of these classes are those

---

\(^1\)Facts and figures taken from the presentation “Status of the Toolkit”, delivered by Hans Johnson at the Fall 2010 Meeting for ITKv4 in Iowa City, IA. Meeting details, as well as the presentation, can be found at [http://www.itk.org/Wiki/Fall_ITKv4_2010_Meeting](http://www.itk.org/Wiki/Fall_ITKv4_2010_Meeting)

\(^2\)The hierarchy for ITK v3.20 can be found at [http://www.itk.org/Doxygen320/html/modules.html](http://www.itk.org/Doxygen320/html/modules.html)
used to read/write data as well perform tasks involved in image segmentation and registration frameworks.

From this class collection, one selects and connects desired classes to one another to form a program where, upon execution, the data will flow from class to class being manipulated by each specified subsequent class until the program reaches completion. This path the data takes is commonly referred to as a “pipeline”. To describe this further, a graphical representation of a typical ITK pipeline program can be seen in Figure 1.1(a).

A further complication when constructing a program in ITK is to ensure that all data are appropriately represented within native ITK structures, of which there are numerous (on the order of tens, if not a couple hundred). Most information stored, used, and generated by ITK has an associated ITK-specific type that must be explicitly declared when writing ITK-C++ code. Certain classes require input data to be of a specific format, such as Image Filters requiring the data having been previously read and stored as an `ImageType` before processing can take place. Others generate data within particular constructs that may be unexpected, such as rotation and/or translation calculations stored as a `ParametersType`, which will likely require conversion before being used further.

These heavy programming requirements are a major setback keeping libraries like ITK from being commonplace within a biomedical research or surgical team; the in-depth knowledge of C++ programming language is essential. Even for computer science graduates, learning the thorough inner-workings of ITK and building proficiency comes with an extensive uphill learning curve. For comparisons of required codes, a simpler example of a pixel-intensity threshold program is shown in both ITK-C++
1.1. MOTIVATION

along with its equivalent workflow using the proposed SimITK method are presented in Appendix A, while a more complex example of a registration of MRI-to-CT data is shown in equivalent code and workflow forms in Appendix B.

This programming requirement is an overall hindrance to the adoption of tools like ITK within medical research. As such, a way to abstract the complexities of the programming away so the learning curve is greatly reduced, thus increasing accessibility, would be a potentially welcomed solution. Simultaneously, a means to rapidly prototype ITK programs that can showcase and take advantage of the powers of these libraries would be of great benefit to both present and future researchers.

Fortunately, the pipeline-based nature of ITK and the visual programming approach employed by Simulink, an environment within the ubiquitous MATLAB (Mathworks, Natick, MA) scientific programming suite, are both based on dataflow programming making the two ideal for combined use.

1.1.2 Simulink: A Modeling System for Visual Programming

Simulink is an interactive graphical environment within MATLAB. As described on its product page\(^3\), Simulink is “an environment for multidomain simulation and model-based design for dynamic and embedded systems. It provides an interactive graphical environment and a customisable set of block libraries that let you design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing”. From a given block library (or combination thereof), desired blocks are selected and added to a blank Simulink canvas before being interconnected to create a model file. Subsequently, the model file is executed to perform a given task. Figure 1.1(b) is an

\(^3\)http://www.mathworks.com/products/simulink/
1.1. MOTIVATION

(a) A flowchart representing the pipeline-based nature of ITK programming where classes are connected to one another to create a virtual information “processing chain”. In most ITK cases, image data are read in using a file reader class and connected to one (or many) image processing classes before being, in this example, written to disk.

(b) An example Simulink model that computes the product and subsequent integral (highlighted in yellow) of two different inputs (in cyan and green) before outputting them (in red) for future use.

Figure 1.1: A comparison of pipeline-based ITK programming and graphical programming within Simulink.

element of a Simulink signal-processing model that takes two different inputs, calculates their product and performs an integration before outputting the result for later use.

A benefit of using graphical environments like Simulink, particularly in cases where the user does not have a developed programming skill-set, is that the necessity to learn traditional written programming code is abstracted away and replaced by an equivalent visual representation. As such, this allows the user to focus on the details of solving the problem at hand with less of a requirement on the details of coding.
1.2 Proposed Method: SimITK

As shown in Figure 1.1, similarities can be observed between the thought-process involved in pipeline-based image processing programming and creating visual programming models: blocks represent data or performed actions and arrows represent the transfer of information between blocks. From this observation, it was hypothesised that these two ideas could be joined such that the power of image processing could be clearly delivered to the user using the simplicity of visual programming. Simultaneously, the complication of conversions to/from specific ITK types could be automatically handled by the software without requiring user-intervention, increasing the focus placed on solving the problem and reducing that on programming details.

The remainder of this thesis will present and outline SimITK, an implementation of ITK pipeline-based programming within the Simulink visual programming environment. Simulink was the targeted visual programming environment because of the ubiquity of MATLAB, its parent application, within research and development environments. This ubiquity would allow for an already large and established user base to take advantage of the image processing capabilities of ITK within a familiar environment. Furthermore, since MATLAB can execute Simulink codes (and vice versa), any Simulink model files with SimITK blocks could interact with any previously-developed MATLAB code, increasing the variety and power of the toolset available to the user. Utilising a previously developed, well-maintained graphical environment also eliminated the need for creating such an environment specifically for this project; an unnecessary task when a mature, well-known environment like Simulink is available.
1.3 Thesis Objectives

The objective of this work is to develop a procedure that will take the functionalities of ITK and create Simulink block equivalents that can be graphically connected in the same manner one would construct the corresponding ITK code. For users with little or no programming experience, such as medical professionals, this solution would allow the image processing capabilities within ITK to be presented in a non-invasive and intuitive manner. In contrast, programmers unfamiliar with the details of ITK could also benefit from SimITK as it offers access to ITK without the necessity to learn the required “language” of ITK nor high-level C++.

This objective can be divided into four subsequent goals:

1. The creation of an organisational schema, terms of constraints on the structure and content of the document, capable of storing a representation of the ITK information required for each class (name, input(s), output(s), permitted datatype(s), acceptable dimensionality/-ies, class method(s), method argument(s), etc) for future retrieval and use in generating Simulink blocks.

2. The automated generation of a schema file for each desired ITK functionality.

3. The creation of Simulink Block Libraries, as well as any additional required integration files, that would then be used to create SimITK workflows, from these previously generated schema files.

4. The evaluation of the SimITK workflow against a pure C++ program: demonstrating that using SimITK only differs from standard ITK code in terms of appearance; functionally the two are equivalent.
1.4 Thesis Contributions

In this thesis, I have:

- Successfully automated the generation of files containing ITK information using a schema designed for simple information retrieval to aid in generating a Simulink block from an ITK class.

- Successfully generated Simulink Block Libraries of a potent subset of over 130 classes typically used in the creation of ITK segmentation and registration workflows.

- Demonstrated that SimITK is capable of creating full workflows that yield the same results as their pure-C++ code counterparts.

- Demonstrated in test case studies that SimITK runtime is on par (in some cases faster) than their pure-C++ code counterparts.

- Created a website, http://www.simitkvtk.com/, that promotes and supports SimITK and its sister-project, SimVTK. Information on the website includes past and present releases, extensive user and developer documentation, as well as multiple examples and tutorials complete with sample data.

1.5 Thesis Outline

This thesis is divided into five chapters that present the schema representation itself, the process of generating the schema files, how the information within those files is integrated into Simulink blocks, and the results from creating SimITK workflows. The thesis is organised as follows:
Chapter 2, Background: provides a brief description of image processing, dataflow/pipeline-style programming, visual programming, and language wrapping. It also presents prior work in this area.

Chapter 3, Methods: details both the process of creating the schema files and how the stored information is used to generate the final Simulink blocks.

Chapter 4, Results: outlines the usability of SimITK within Simulink including examples of the block libraries, workflow construction, and runtime performance details.

Chapter 5, Conclusions and Future Work: presents the key conclusions of the thesis and possible areas of future work that can aid in preparing this work for future development and enhancements.
Chapter 2

Background

In this chapter, the aspects fundamental to the SimITK method will be presented that cover the programming style, user interaction methods, relevant terminology, integration techniques, and related works.

This background knowledge has been divided into five distinct sections as they are of particular interest to SimITK:

- **Dataflow Programming** as this is the employed method of information transfer and underlying programming paradigm used in ITK, and by association, SimITK.

- **Visual Programming** will be used to abstract away the programming nuances of writing code in favour of a more intuitive and easy-to-understand graphical representation. The core concepts will be explained here and parallels will be drawn between an early implementation and the current Simulink environment.

- **Image Processing Terminology** will also be discussed for the case studies built by SimITK to ensure the reader is familiar with these tasks.

- **Language Wrapping** is presented as SimITK is the integration of the ITK image processing library within Simulink through a sort of language wrapping, or encapsulation process.
2.1 DATAFLOW PROGRAMMING

ITK-Based Image Processing Implementations are also presented as related works that highlight the shortcomings and strengths of currently available implementations and how SimITK differs and provides its own unique opportunities.

2.1 Dataflow Programming

An implementation of the Dataflow Programming paradigm was initially envisioned in the late 1960s (Adams [1]) and early 1970s (Dennis [8]) out of an increased desire to design a method of programming based on the connections, the flow, between program elements as opposed to focusing on the data changes occurring as program execution progressed. Typically, this programming style is modeled similar to a graph diagram representation, illustrated in Figure 1 of [8], to which clear parallels can be made to Figure 1.1.

An example of software that uses dataflow programming paradigm, and is of particular interest to several of the image processing implementations outlined in this chapter, is the Visualization Toolkit (VTK) [22] produced by Kitware Inc., the creators of ITK. VTK is a free, open-source, and cross-platform library used for visualising data commonly applied to areas such as 3D computer graphics and image processing. VTK consists of C++ class libraries at its core as well as several wrapped interface layers such as Tcl/Tk1, Java2, and Python3, thus enabling a user to write VTK programs in a language other than C++. These classes must be interconnected to form a pipeline that will accomplish the desired task. Once established, these connections are what will facilitate the dataflow in the same manner outlined by

---

1http://www.tcl.tk/
2http://www.java.com/
3http://www.python.org/
2.2. VISUAL PROGRAMMING

Adams and Dennis.

ITK uses the same pipeline-based programming as VTK; a paradigm that is well-reflected by visual programming, an interactive style of creating programs through interconnecting visual representations of code elements. This concept is described in greater detail in the following section.

2.2 Visual Programming

At its core, the paradigm of visual programming aims to substitute the written aspect of programming with functionally-equivalent visual representations. All programming elements, such as input data and executed commands, are represented by graphics that can be interconnected to create a program. Further interaction with the representation facilitates customisation in order to modify the element appropriately for the given program being designed. Visual programming implementations are often closely coupled with dataflow programming [17] as the two concepts are easily meshed.

As an example of an early visual programming implementation, Pictorial Transformations (PT) [19] is where the user performs “algorithm animation”: a programmer specifies an algorithm in PT by altering and interacting with visual representation objects. By the nature of the programming, as the “animation” is visually constructed, the algorithm is largely developed at the same time.

A “graphical object editor” is included within the PT suite that allows for the construction of new object types. The editor allows for the parameterisation of the iconic object display depending on its set attributes. PT objects consist of tuples of attribute-value pairs as well as a function that creates the iconic display of the pairs corresponding to a given object. Should further structure be required, another object
2.2. VISUAL PROGRAMMING

can be the value of an attribute.

Programming PT requires the creation of graphical objects and then the connection of a series of objects to create an overall algorithm. The authors consider a *picture* to be a collection of graphical objects while a *film* is a sequence of manipulations performed on a picture. A starting picture must first be created so that pictorial transformations can be applied. The process of applying the transformations is recorded as one, or many, films. On a broader scope than films exist *selections*: a set of target objects could be used in other selections or could control object modifications. *Selection criteria* are used to then determine which objects from the set should be considered the domain of objects to be potentially manipulated while a *selection condition set* selects from that domain the objects that meet the attributes specified within the condition set.

Together, these elements contribute to filming an animation and is referred to as being in the context of a *situation*. Initial filming takes place in the context of the initial situation. New situations are created through invoking other films on the objects or by evaluating *predicates*: a condition that is tested through filming a picture to return a value. Predicates are used primarily to facilitate conditional filming.

Many of the concepts introduced in PT parallel the Simulink environment. The described *visual objects* are akin to the Simulink blocks that will be interconnected to build workflows. The parallel is furthered as the blocks are complete with attribute-value pairs that are used to define method arguments, much like the visual objects. On a larger scale, *pictures* are akin to complete SimITK workflows composed of many blocks that perform the desired image processing task. In terms of the action
performed by the workflow, this is equivalent to a film in PT. Furthermore, a variant on selections can be seen in Registration example (Section 4.2.3) in the form of an optimizer block as they make use of criteria previously calculated at execution time to direct future iterations.

2.3 Image Processing Terminology

Two tasks commonly performed using image processing techniques are segmentation and registration. Workflows were built to highlight these tasks and were used to test and evaluate SimITK. In this section, the pertinent terminology will be briefly explained as well as descriptions of the approaches taken.

2.3.1 Segmentation

Segmentation is the process of dividing an image into one, or many, sub-image(s) such that the altered result is easier to interpret or analyse [13]. The process requires detecting commonality, such as colour or intensity, to create sub-regions within the image as a whole. Within medical imaging for instance, segmentation is frequently performed on data to highlight bony areas from regions of soft tissue as to simplify the image when, for example, diagnosing a patient, planning surgery, or studying anatomy.

There are several common techniques used in segmentation including, but not limited to: thresholding, based on common traits; clustering, where the image is divided into a defined number of clusters, based on similar characteristics; and edge detection. All of these techniques aid in segmenting the image such that the result is more focused on the area of interest.
An example of a simple segmentation application is presented as a case study for SimITK in Section 4.2.2 that uses a common technique, pixel-intensity thresholding, to segment the bright bony areas away from the background and surrounding tissues captured by the original data.

### 2.3.2 Registration

*Registration* is the process of taking multiple datasets and transforming them such that all are within the same coordinate frame. Information commonly registered include multiple photographs, for the purpose of producing a larger image, or data collected from multiple sensors that could be combined to give a more complete understanding of the scene under investigation. In the case of medical imaging, registrations are frequently employed when relating data of the same patient acquired through different scanning modalities (e.g. magnetic resonance imaging (MRI), computed tomography (CT), or ultrasound) and/or at different time points. Typically, one of the datasets is established as the source while leaving the other data as target(s) [14]. When the target data are aligned to the source, the registration has converged and is complete.

When discussing a registration framework, there are key aspects that require clear definition before the registration can commence. These aspects, once defined, outline the iterative process that will be employed when attempting to align the datasets. From the initial states of the datasets, the target is transformed in subsequently smaller steps until an optimal alignment between the source and target is achieved, as calculated by the registration.

As outlined in [39], registrations can be divided into two general categories: rigid
and non-rigid. The former is commonly based on either geometry or pixel/voxel intensity, with its associated transformation being the simplest with only six degrees of freedom, is invertible, and has closed-form solutions [2, 18]. A closed-form solution is a mathematical solution to a problem; in the case of image registration, this would represent a correspondence between points in the data. To contrast, non-rigid registration is commonly either feature or intensity-based and accompanied by transformations with several more degrees of freedom [16].

Including the source and target data, another four factors must be considered that will be outlined next. Since an example rigid registration of MRI-to-CT data will be presented as a case study for SimITK in Section 4.2.3, the four discussed factors described below pertain to rigid registrations exclusively.

**Transform**

The coordinate transformation, often referred to as the “transform”, is the means by which the two datasets are to be aligned. In the case of 2D registration, this is accomplished through either translating and/or rotating the target to align it with the source. In the case of rotation, a centre of rotation must be chosen. The transform values computed by a registration represent the parameters (degrees of rotation and translation distance) performed on the target that optimally aligned it to the source.

In the presented case study, a 3D Centered Euler transform is employed that manipulates the target MRI volume, compared to the source CT volume, by performing rotations about and translations along the three coordinates of the volume using the volumetric centrepoint as the origin.
2.3. IMAGE PROCESSING TERMINOLOGY

Metric

The image similarity measure, often shortened to “metric”, is used to measure the similarity of appearance of the two data. Some metrics need to be minimised (if they express the difference between images, as the lowest difference would yield the optimal result) while others need to be maximised (if they express similarity between images, the most similar combination is optimal).

In the presented case study, the metric used is *Mutual Information* that, as implied by its name, measures the shared information between the CT and MRI data. It seeks to measure how much knowing about one dataset reduces uncertainty about the other. If the CT and MRI data were completely independent (for example, scans of different regions of the same patient), then the CT-information does not aid in determining any MRI-information; their mutual information is zero. However, if the two datasets were identical (for example, registering the CT data to itself) then all information is shared and their mutual information would be 1.

Optimizer

The optimal transformation for the registration is found through an optimisation process, which varies the parameters of the transformation in order to maximise or minimise depending on the utilised image similarity metric. When the variance of the transformation parameters exceeds (or falls beneath; metric-depending) a user-defined threshold, the optimisation process terminates.

In the presented case study, a *Gradient Descent* optimizer is used that calculates the gradient, the direction of progression, from the previous registration attempts in order to extrapolate the next set of transformation parameters to attempt when
2.4. LANGUAGE WRAPPING

registering.

**Interpolator**

Digital images are described as a finite set of data samples, but the image actually represents a continuum, e.g. $I(x, y)$ where $x$ and $y$ are continuous variables. Therefore, an image has to be expressed in terms of a mathematical function $I(x, y)$, i.e. an interpolator, that can be applied to the discrete samples.

In the presented case study, *Nearest Neighbour* interpolation is used which takes a given point requiring interpolation, locates its closest point, and uses the value of that closest point as the interpolated value for the original point.

Once appropriate source and target data, transform, metric, optimizer, and interpolator have all been defined, the registration has been established and can be performed.

2.4 Language Wrapping

Wrapping is a computer language translation technique that aims to translate just the application programming interface (API) into a new language, while leaving the implementation in the original programming language. This is commonly done to facilitate the interaction of a software library written in the former language within the environment of the latter. Reasons for this could include the latter language having a simpler syntax, leading to more timely development; being a more appropriate choice for a given implementation, to perhaps be integrated with previous developments; or being already familiar to the user, so only the details of the wrapped library need be learned.
2.4. LANGUAGE WRAPPING

The Simplified Wrapper and Interface Generator\(^4\) (SWIG) \([5]\) is an example of a tool that can be used to take libraries and/or programs written and implemented in C or C++-like programming languages and integrate them within higher-level programming environments like perl, PHP, Python, Tcl or Ruby\(^5\). Such integration facilitates rapid testing and prototyping of the original library within the new environment through the automatic generation of functionally equivalent code. Through using environments like those previously listed, a programmer needs only download and install the appropriate runtime environment to execute written codes; no further complications involving code compilation, code building, or library linking is required.

CableSWIG\(^6\) is an extension of SWIG designed specifically for the ITK image processing library. Through the use of CableSWIG when compiling and installing ITK, one can choose to integrate ITK into any or all of the Tcl, Java, and Python development environments, if desired. ITK can also, as one would expect, be utilised through its original C++ programming environment.

WrapITK\(^7\) \([26]\) depends on GCCXML\(^8\), an application included as a part of CableSWIG, to create representations of ITK classes that can aid in the integration, or wrapping, of ITK into an environment different than those specified by CableSWIG.

In the case of SimITK, the proposed method outlined in this thesis, WrapITK represents a crucial component necessary to encapsulate ITK within Simulink. The implemented method detailed in the following chapter is a complex wrapping process that integrates the C++ ITK library into the Simulink visual programming environment to facilitate the rapid development of ITK programs in the form of SimITK.

\(^4\)http://www.swig.org/
\(^5\)http://www.ruby-lang.org/
\(^6\)http://www.itk.org/ITK/resources/CableSwig.html
\(^7\)http://code.google.com/p/wrapitk/
\(^8\)http://www.gccxml.org/HTML/Index.html
2.5. ITK-BASED IMAGE PROCESSING IMPLEMENTATIONS

In this section, current ITK-based implementations from the literature will be investigated to provide an overview of the advantages, shortcomings, and power each suite has to offer.

2.5.1 Environments

Here, the implementation details of several different image processing environments will be discussed including SCIRun, ANALYZE, MATITK, 3D Slicer, and XIP. These environments are entire software solutions that provide image processing abilities to the user.

SCIRun

SCIRun\(^9\) \([31]\) is a scientific programming environment that allows the interactive construction, debugging, and steering of large-scale scientific computations. Using an iterative process of manipulating a pipeline to generate desired result(s), the user is capable of “steering” the pipeline, interacting with user-controllable scientific simulations occurring during computation, to accomplish the desired computation(s).

SCIRun acts like a small operating system in terms of memory allocation for the various tasks being performed as this ensures resources are used effectively. Since the inception of SCIRun in the early-1990s, it could be considered a precursor to modern integrated development environments, visual programming environments, and

\(^9\)http://software.sci.utah.edu/scirun.html
development environments as a whole [29, 30]. To that effect, SCIRun [21] is very early work on what we now have as suites like ANALYZE and MITK. Furthermore, SCIRun has been considered preferable to other suites because of its focus on dataflow programming and application prototyping abilities [6].

**ANALYZE**

ANALYZE\(^{10}\) [35] is a software package that permits detailed investigation and evaluation of 3D and 4D biomedical images. It is usable with multiple modalities such as Computed Tomography (CT), Magnetic Resonance Imaging (MRI), and Ultrasound (US), while also supporting a wide range of image formats, and is equipped with a fully interactive Graphical User Interface (GUI). ANALYZE also features one of the first implementations of a volume rendering algorithm used for visualisation. As of its initial 1990 publication [35], the suite was advanced for its time supporting reading and writing to magnetic tape.

At its core, referred to as the ANALYZE Visualization Workshop, ANALYZE is a standalone UNIX program capable of operating on standard computers and requires no additional specialised hardware. The core was designed to be modular and is divided into six distinct tool-groups: image I/O, processing, segmentation, registration, quantitative analysis, and 3D visualisation and rendering. ANALYZE supports a variety of functions that were cutting-edge at the time of initial release (such as select, pan, zoom, colour map, *etc*) that would now be an expected toolset within any modern application.

Following initial publication, further development has allowed ANALYZE to mature such that “ITK functionality” has been implemented [3]. This integrates ITK

\(^{10}\)http://www.mayo.edu/bir/Software/analyze/analyze.html
into the ANALYZE environment, where ITK functionality is defined as all the steps necessary to achieve a desired output, such as a segmentation or registration. All necessary data are mapped from ANALYZE data structures to ITK classes. These mapped data are passed to a templated function which contains all the ITK process classes required for algorithm execution. This includes accounting for ANALYZE being written in C and ITK in C++. Following data processing, the data are then remapped into native ANALYZE-code. To increase efficiency, any data will be directly manipulated instead of being copied to another region of memory, if at all possible.

MATITK

MATITK\textsuperscript{11} \cite{MATITK} is an implementation of ITK that integrates a selection of ITK filters into the MATLAB scientific computing suite. The core of MATITK is MATITK.cpp, a C++ file the user places in a MATLAB-accessible location such that it can be invoked from the MATLAB command prompt to execute and run image filters on desired datasets. Proper usage requires invoking MATITK.cpp along with a flag for the desired filter to-be-executed along with appropriate input and output file names. A table of the supported filters is supplied in \cite{MATITK} along with the specific MATITK command-line acronym required to call the corresponding image filter. Results are output through visualisation using a MATLAB Figure window.

The ITK classes and algorithms implemented in MATITK were accomplished using filtered code derived from examples supplied by Kitware Inc. in ITK. Perl scripts were specifically created to facilitate automatic generation of ITK filtering code.

\textsuperscript{11}\url{http://matitk.cs.sfu.ca/}
3D Slicer

3D Slicer\textsuperscript{12} (Slicer) [11, 12] is a modular suite created to address five frequent issues within the operating room: image quality, imaging time, multi-modal fusion, localisation time, and 3D visualisation. Slicer can be divided into three distinct modules: a VTK module for image processing and visualisation, an OpenGL module for graphics acceleration, and Tcl/Tk for the GUI. This modularity allows for Slicer to capitalise on being extensible: only a new Tcl module specific to a new implementation as well as the corresponding VTK objects for rendering and visualisation are required when implementing a custom module.

The Slicer UI reformats multi-volume data into three distinct slice views for visualisation purposes. Each of these three views are orthogonal, or independently oblique to one another, with each slice able to be oriented orthogonally relative either to the scanner being used (in millimetres) or in terms of the data (in voxels). The region of interest can also be changed through tracking a user’s pointing device. In terms of scanners, Slicer is compatible with data from multiple modalities including, but not limited to, MRI and CT.

Slicer also features: a wide variety of segmentation tools; 3D surface models; multi-modal registration; the Medical Reality Modeling Language (MRML), a language used for describing scenes akin to Computer-Assisted Design/Computer-Assisted Modeling; and trajectory assistance by aiding in providing an overlay of the path to be taken by a given tool or device.

Slicer has also been demonstrated and validated through integrations in MR-guided therapies and continues to be used widely, as outlined in [15].

\textsuperscript{12}http://www.slicer.org/
eXtensible Imaging Platform (XIP)

XIP [28,33] is an open-source environment, produced by the Cancer Biomedical Informatics Grid\textsuperscript{13} (caBIG), launched by the National Cancer Institute\textsuperscript{14} at the National Institutes for Health\textsuperscript{15}, that can be used for rapidly developing medical imaging applications from an extensible set of modular elements. Its initial development phase reached completion in 2007, and can be described as an amalgam of three components: the XIP Application Builder, an Integrated Development Environment that aids in building applications by graphically linking the desired modules together; XIP Libraries, sets of automatically generated objects from custom or existing class libraries (like ITK) that can be used as modules to create XIP applications; and the XIP Reference Implementation that provides infrastructure (such as data, services, and security) where XIP applications can run, operate, and implement processing logic to analyse and visualise medical images and information.

XIP supports the rapid development of plug-in applications for image analysis and visualisation. XIP is equipped with a Digital Imaging and Communications in Medicine (DICOM) interface, a mechanism by which any host supporting the interface may control (as in start, stop, pause, obtain status) and exchange data with any application that supports the same interface profile/version. Abstract models for the data being exchanged are included that allow an application to interact with existing data and produce new data without concern as to how or where the data are stored (including data format). The DICOM interface can also facilitate data access, either through a native model or direct access, should knowing the exact location and

\textsuperscript{13}https://cabig.nci.nih.gov/
\textsuperscript{14}http://www.cancer.gov/
\textsuperscript{15}http://www.nih.gov/
format of the data be required.

Summary

In the aforementioned implementations (SCIRun, ANALYZE, MATITK, Slicer, and XIP), the convenience of ITK-integration carries a potentially heavy setback: the deep knowledge of ITK is substituted for deep knowledge of the environment. The danger of this is that should the user become proficient in the environment and not the library, moving to a different environment may require much relearning leading to prolonged development. Furthermore, when collaborating or presenting results, having the environment installed becomes a necessity. As such, with ITK becoming increasingly ubiquitous, it may be of greater interest to take advantage of the functionalities ITK offers by being familiar with the inner-workings of ITK rather than a given implementation environment.

MATITK differs slightly in that it requires the user be familiar with MATLAB programming conventions. No visual programming elements or a GUI are provided, so the user must have a previous understanding of the subset of integrated filters as to be able to invoke the appropriate MATITK code. Complications arise when attempting to perform multiple filters in succession; the output from one filter must be saved to disk and reloaded when executing the second filter.

2.5.2 Tools and Toolkits

In this section, tools and toolkits that provide a deep functionality of a limited set of image processing techniques are discussed. Most commonly, these are powerful segmentation tools, but only serve this specific functionality.
2.5. ITK-BASED IMAGE PROCESSING IMPLEMENTATIONS

The Image-Guided Surgery Toolkit (IGSTK)

IGSTK [10] is a programming library composed of a combination of software that contains the basic components commonly required to develop an image-guided surgery system. Included components consist of tracker control and display components for overlaying images of patient anatomy and surgical instruments, and common procedure requirements for surgical interventions like radio-frequency ablation. The components can be divided into four major categories: the tracker component, an object-oriented representation of tracking devices, tools, and all relevant static and dynamic communication of associated tracking information (such as position, orientation, etc); the “Spatial Objects” component, an implementation of ITK used for accomplishing image processing tasks; the “Spatial Object Representations” component, for 3D or 2D tomographic/projective viewing; and the “Viewer” component, an implementation of VTK classes that can aid in creating more specialised visualisations. Typical uses for IGSTK include: 2D and 3D imaging display requirements (for modalities such as CT, MRI, and US), as well as organ movement and tracking for use in segmentation and registration programs.

IGSTK exploits the use of compartmentalisation and encapsulation in many ways. The components have been designed such that they have rigid, well-understood, boundaries with easily visualised interaction patterns. This allows for the integration of libraries like ITK to be done in a safe (in terms of memory use and robustness) and predictable manner. This strategy, however, does come with issues in that the encapsulation is accompanied by increased time and processing requirements. Furthermore, since these components are automatically integrated into the libraries, it is very difficult to debug inter-component interactions.
ITK-SNAP

ITK-SNAP [40, 41] is “an open-source application intended to bring level set active contour segmentation to the fingertips of clinical user” [40] (a comprehensive discussion of the level set algorithm is provided in [40]). ITK-SNAP was created with the intent of providing users with minimal or no mathematical background the power to make segmentation and parameter selection as easy as possible. This goal stemmed from the authors observing that “many biomedical research labs continue to rely on manual delineation” [40], so ITK-SNAP aids in decreasing this dependency by implementing the level set algorithm and providing additional tools used for manual outlining and quality control to the user. The self-described goals of ITK-SNAP are: to allow the user to specifically focus on the problem of segmenting anatomical structures instead of introducing a prohibitively steep learning curve; to construct a friendly and well-documented user interface that would break up the task of initialisation and parameter selection into a series of intuitive steps; to provide an integrated toolbox for manual post-processing of segmentation results; and to make the tool freely accessible and readily available through being an open-source project.

While other software suites like ANALYZE and Slicer, provide a broad range of tools and algorithms, ITK-SNAP is strictly intended for segmentation. As such, the functionalities made available to the user are a small, robust segmentation toolset. The ITK-SNAP environment is not visual programming, but an interactive GUI that emphasises the 3D nature of images: three orthogonal views are displayed that intersect at a given point identified by the “3D cursor”, a tool to determine the exact location of an area of interest within a volume. Each view can be zoomed and allows for performing manual segmentation. It is possible to assign a numerical label to each
voxeL that corresponds to an anatomical structure for ease of viewing. The benefit to such a strategy is that a segmentation is represented as an image, not a set of contours, which greatly improves visualisation and accurate, uncomplicated segmentation.

For ease of use, a wizard is provided that is capable of defining segmentation parameters (complete with live preview and feedback) because of the tight integration of ITK within the GUI. Based on the input given to the wizard, methods of segmentation can be interactively applied that give the user abilities to choose from a selection of algorithms including those based on pixel intensity and image edges.

**VolView**

VolView is an interactive, cross-platform system produced by Kitware, the creators of ITK, for volume visualisation; specifically of medical and scientific data. Within VolView, loaded 2D/3D data can be interactively analysed and investigated using tools like volume rendering, maximum intensity projections, and oblique reformatting. VolView is extensible through using their built-in plug-in API.

VolView has been extended to integrate ITK such that pipelines generated using ITK classes could be run on a given dataset prior to visualisation. VolView has also been found to be potentially advantageous when developing non-automatic end-user applications based on existing ITK functionalities.

**Summary**

These tools were designed to excel at their specific task. Provided this is all the user requires, tools like these will serve their purpose and would be ideal. However, extending them beyond their scope is not feasible. In such cases, complimenting the

---

tool with another image processing library or software may be a more appropriate solution.

With ITK-SNAP, for example, the user is given access to level set segmentation tools in a manner aimed to be familiar to non-mathematicians. So long as these are the only tools the user needs, the user requires nothing further of their tool. However, ITK-SNAP can not be easily expanded nor can it be easily integrated into an automated system should repeating a given method be required.

2.5.3 Integrated Visual Programming

In this section, several other visual programming environments that integrate image processing abilities will be discussed.

ITKBoard

ITKBoard [25] is an extension of ITK that offers intuitive, efficient experimentation and prototyping of image processing pipelines through interaction with its GUI. Among many roles, it can be used as a visual debugger for algorithm development. As in the cases of other suites mentioned in this chapter, ITK is at the core of ITKBoard. Since ITK is a library and not an end-user application, difficulties arise for non-programmers when they attempt to interact with the library. The goal of ITKBoard is to provide a visual tool to help in the deployment, development, and refinement of segmentation and registration algorithms that benefits the image processing community using a GUI.

Features and functionality included with ITKBoard include: the design and construction of multiple image processing pipelines through GUI interaction, complete
2.5. ITK-BASED IMAGE PROCESSING IMPLEMENTATIONS

with fine-tuning filter parameters; integrating ITK filters into modules for image processing; on-the-fly visualisation of the result from each pipeline filter in succession, allowing the user to see the progress made by the pipeline at each step; code-generation abilities that build stand-alone applications for each pipeline design for export and future reuse outside of ITKBoard; multi-threading support; and the automatic generation of wrappers for third-party ITK-style filters. ITKBoard also supports the design of custom-written plug-ins that can facilitate further ITKBoard GUI functionality without requiring a full recompilation.

In regards to the automatic wrapper, the associated header files with the to-be-imported code provide built-in functions to prototype well-formed ITK code. Properties can be detected by parsing the header for function names with matching pairs of functions beginning with Get and Set. As with ITK, ensuring the use of appropriate datatypes is crucial to the proper execution of a pipeline. Thus, an iterative depth first search is used to traverse the hierarchy of classes and superclasses until an appropriate level has been reached that allows the pipeline to appropriately define its datatype. This is a method that differs from implementations like SCIRun [31] that use a custom XML document to store such information. The formatting of ITK header files facilitates quick recognition of filter information, and the ITK class hierarchy can be reused for pre-compiled functions.

MeVisLab

MeVisLab\textsuperscript{17} [24,34] is a fusion of several libraries and interfaces into one: ITK; VTK; Open Inventor\textsuperscript{18}, an interactive environment that combines both dataflow and visual

\textsuperscript{17}http://www.mevislab.de/
\textsuperscript{18}http://oss.sgi.com/projects/inventor/
programming; the core custom “MeVis” image library, a general framework similar to ITK in that it seeks to solve similar segmentation and registration problems, but differs in that its algorithms are encapsulated as self-descriptive functional units which can be combined to form image processing pipelines; all within a visual programming framework. MeVisLab is implemented in C++ and uses the Qt library to incorporate a GUI for visual programming.

Integration of the various libraries is performed automatically. The source code is parsed into an XML structure and given an appropriate prefix (such as itk for ITK, vtk for VTK) to identify its contributing library. From these XML representations, modules for each class and function are created using a custom, pre-built template specific to the library being integrated (i.e. importing an ITK class uses the appropriate ITK template); special cases are treated and handled independently, as required. The same method is used for also integrating the supplied custom MeVis image processing library. For all integrated libraries, appropriate documentation is integrated to aid the user.

Image processing tasks are primarily done with the MeVis image processing library while visualisation tasks are handled by the full integration and extension of Open Inventor [36].

Extensibility within MeVisLab is facilitated by creating modules for either Open Inventor, or the MeVis Image Library; both of which require C++ knowledge. Also, MeVisLab has been found to be a beneficial solution because of its implementation of the dataflow programming paradigm and abilities when creating application prototypes [6].
MITK

MITK\(^{19}\) \([37, 42]\) is based on ITK and VTK, but extends them with features required for interactive systems. MITK provides a toolkit on which applications can be built and designed for general solutions, specific problems, and development environments. The key features to MITK include the coordination of multiple 2D and 3D visualisations of arbitrary data; a general interaction GUI concept that includes helpful shortcuts and keystrokes common to most other environments, such as undo and redo; extendability and flexibility to create tailored applications from a fusion of the included multiple libraries and components; while also abstracting away different layers of hidden complexity. While based on ITK and VTK, the two libraries are tightly integrated such that they are not implemented into the MITK environment, they combine to make MITK itself.

The core of MITK uses a self-described “data tree” to define the contents of the output visualisation window. When including objects to be visualised, associated information is stored within a database to allow for easy manipulation (when animating an object or objects, for example). This makes MITK have a model-view-controller-like design: interactions are performed directly on the data, not on the object-representation, before being rendered. On a technical note, MITK handles all interactions are in terms of physical millimeter distances and not pixels/voxels.

MITK has been found to be preferable over other suites when developing automated end-user applications that might include new ITK classes specifically designed for the application \([6]\).

\(^{19}\)http://www.mitk.org/
2.5. ITK-BASED IMAGE PROCESSING IMPLEMENTATIONS

In 2010 [38], MITK was integrated into the XIP 2.5.1 visual programming environment. This development allowed proficient users of either environment to benefit by taking advantage of the abilities of the other.

Also in 2010 [4], the MITK Image Guided Therapy framework was created with the goal of integrating software tools, algorithms and tracking device interfaces into MITK. It provides a comprehensive software framework for computer-aided diagnosis support, therapy planning, treatment support, and radiological follow-up within the MITK environment.

Summary

These implementations of ITK similarly accomplish several of the same objectives as SimITK including the ease-of-use through employing the visual programming paradigm. In the case of ITKBoard, this includes the convenience of automatically wrapping ITK. However, they all differ sharply as a proprietary GUI has been developed exclusively for use within the implementation. SimITK is different in that it uses WrapITK, which, unlike the other toolkits, provides a comprehensive, automatic translation of the ITK interface instead of adopting a “pick-and-choose” approach where a hand-picked subset is translated. The use of Simulink, which is a general-purpose visual programming environment and is integrated with MATLAB, is the other difference which couples the power of ITK with the scientific computing abilities of MATLAB, allowing for the creation of more complex workflows.
2.5.4 Culmination

All of these previously detailed fields, areas, and concepts will be fused together to create the SimITK visual programming implementation of the ITK image processing library.

As outlined, the ITK image processing library uses dataflow programming concepts to build programs composed of connected classes. These ITK class interconnections can be naturally represented by the visual programming paradigm found at the core of Simulink. Therefore, to successfully integrate ITK within Simulink, a sort of language wrapping procedure is used to combine the image processing power of ITK with the visual ease of Simulink to create SimITK.

The next chapter extensively outlines the SimITK wrapping process including the steps taken, challenges faced, and solutions implemented.
Chapter 3

Methods

3.1 Overview

The implemented method that creates the SimITK Simulink block libraries is a complex wrapping procedure. At its core, a simple and legible representation of each ITK class is required. This transparent representation is critical to the whole method because the information stored therein will be retrieved frequently throughout the wrapping process and substituted into several different, custom-written templates — one for each filetype needed to successfully integrate ITK within Simulink. Within the representation, pertinent class information needs to be stored such as the name, type (Image Filter, Optimizer, Transform, \textit{etc}), parameters, as well as all inputs, outputs, methods, method arguments, acceptable dimensionalities and permitted datatypes.

The \texttt{eXtensible Markup Language} (XML) was selected as the language best suited for creating a schema, an organisational structuring of the contained data, that would allow for storage and retrieval of the pertinent ITK class information. Another reason for selecting XML stemmed from the desire to take advantage of WrapITK, an
optional package within ITK that generates a complete, yet complex, XML repre-
sentation of an ITK class. However, this complex XML document requires significant
consolidation to make it useful within the SimITK wrapping procedure.

After an XML representation of each ITK class is generated using WrapITK and
refined into a useable state, the ITK information therein is used to generate the
code and the user interface data files for SimITK by expansion of keywords within
specialised SimITK-template files: special keywords left in the templates serve as
anchor points to be replaced by ITK information and/or integration code. One series
of substitutions exists for each template with every filetype requiring its own template.
Several perl scripts and modules were written to perform the substitution series for
each template in order to generate class-specific files.

A graphical overview of the wrapping method that highlights when and from
where the XML are generated, the different required files, and the order in which
they are created can be seen in Figure 3.1. As a convention, the top-most files are
lowest on the dependency tree, meaning they must be generated before files lower in
the Figure.

As illustrated, after the XML class information has been retrieved, it is substituted
into a template for the Virtual Block .tpp file, a communication block that exists
between the Simulink and ITK workspaces, that the subsequent files require prior to
being generated.

Following .tpp generation, the several required Simulink/MATLAB source code
files need to be generated. One is a Simulink “S-Function” .cpp file that contains
the code that will be executed by Simulink at runtime. Two accompanying files
also require generation that will provide the block representation within Simulink:
3.1. OVERVIEW

Figure 3.1: A graphical overview of the SimITK wrapping procedure. The dependency tree is constructed such that a parent file is a required dependency of a child file (e.g. the .cpp depends on both the .tpp and SimWrapITK .xml files being generated before it can be generated itself).
3.2. XML CREATION

representing the ITK classes in a simple, standardised, and legible format was absolutely critical for ensuring an effective wrapping procedure. As previously stated, XML was selected as the language of choice because of its previous use in the WrapITK package within ITK and because of its flexibility as a data language—a required consideration as the XML information needed to be later substituted into the various file templates. Section 3.2.1 details the optional WrapITK package and the challenges therein while Section 3.2.2 describes the solutions to these problems as well as the final structure of the SimITK XML schema.
3.2. XML CREATION

3.2.1 WrapITK

WrapITK [26] is an optional package that can be included with an ITK installation. It makes use of a group of applications that aid in wrapping C and C++-like languages into other non-C/C++ languages and/or environments. GCCXML, a program included within the WrapITK package, was critical in the XML generation step of the wrapping procedure. As ITK is being compiled, GCCXML works alongside the compiler to output an XML representation of each ITK class according to user-specified configuration options prior to compilation. The complexity of the generated XML is due to GCCXML directly outputting the resulting internal gcc data structures found when parsing the ITK C++ code.

The output representation contains all class information including name, parameters, datatype, dimensionality, methods, method arguments, inputs, and outputs as well as any required ITK-types and associated superclasses with their pertinent class information (the same aforementioned information list). An example of this XML can be seen in Listing 3.1. To give the reader a more concrete understanding for the desired output from the XML conversion, the final representation for this same method can be seen in Listing 3.2.

It can be observed from Listing 3.1 that while some of the information can be easily extracted by reading the XML (the method name from the “name” element, for example), there is much information to be desired from this code that is not directly apparent: class-specific information like dimensionality, datatype, and how a given entry may be related or connected to another. WrapITK expresses these relations as unique id values assigned to each entry where these values are the sole identifier one can use to reverse-engineer any relations. The complementary identifier
is the context tag that can be thought of as the reverse relation; it provides the “parent” id value connecting that entry to another.

As previously mentioned, all acceptable datatypes and dimensionalities are included within the initial XML document generated by WrapITK. Since an ITK class is capable of handling image data stored in different dimensionalities and datatypes, the XML representation contains a “class template instance” for each combination. For example, if a given class can handle data in both two and three dimensions of either a float, or a short datatype, four class instances will exist: 2D-float, 3D-float, 2D-short, and 3D-short. These multiple instances of the same class cause problems when attempting to resolve if/when variables corresponding to the datatype or dimensionality were used within the class instance. These “template variables” act
as placeholders - the exact value is not important, but knowing that this variable corresponds to information that depends on the class instance is critical. Discovering when these variables have been used is extremely difficult as GCCXML resolves these variables to the instance information within its output XML.

As such, template variables must be reverse-engineered from the class instances in order to re-establish when such variables were used so this requirement, as well as all other pertinent class information, can be reflected within the final SimWrapITK XML to be used throughout the wrapping procedure. The techniques used to recover the class information and its final representation are detailed next.

### 3.2.2 SimWrapITK

The solution used to integrate the ITK class information into the SimITK wrapping procedure is to process and refine the original WrapITK XML such that the final output is a simple and transparent class representation. This was accomplished through the creation of `convertWrapITKtoSimITK.pl`, a perl script that uses the `XML::DOM` perl module, a package capable of parsing XML data, to analyse the WrapITK XML file for a given class, construct a hierarchy of XML nodes based on the relations between the entries, and produce a final XML document containing the extracted class information in an easily-retrievable format.

The process of converting the WrapITK XML to the final SimWrapITK XML can be broken into two steps. The first being the creation of an intermediary “dump” file containing the reconstructed, raw XML structure (examples provided in Appendix Listings F.1 and F.2) that was used to create and discover the relationships between

---

1The perl module can be found within CPAN at [http://search.cpan.org/~tjmather/XML-DOM-1.44/lib/XML/DOM.pm](http://search.cpan.org/~tjmather/XML-DOM-1.44/lib/XML/DOM.pm).
the class instances, parameters, methods, and method arguments. The second step requires generating the final SimWrapITK XML. When creating the dumpfiles, it was observed that much of the critical class and method argument information was being kept as separate entries in non-obvious, buried locations within the WrapITK XML. By resolving the associated $id$ and $context$ values, relations are established between the class instances, their methods, and corresponding arguments and stored in a $Class\_Data$ or $Argument\_Data$ variable associated with the class instance or method, respectively.

With these new variables declared and associated information stored therein, the next task is to resolve the inputs, outputs, methods, and superclasses (along with their respective inputs, outputs, methods, and superclasses traversing up the entire ITK class hierarchy) associated with each class instance. This is accomplished through extracting and parsing the $members$ node of each class instance (see Appendix F, Listing F.1, lines 22-29). Each $members$ entry represents an $id$ value for a statement found elsewhere within the WrapITK XML; a child of the current class instance. These $id$ values are resolved such that all input, output, method, method argument, and superclass information can be stored for future placement within the final “refined” SimWrapITK XML. Frequently, the associated superclasses contain methods that exist within the accessibility scope of the class instance and require inclusion. As such, all superclasses are also recursively scanned for their own respective inputs, outputs, methods, and method arguments for inclusion within the final XML representation.

Once the hierarchy of $id$ values is resolved, the information is written to a final, refined SimWrapITK XML representation of the class that serves as the main reference point for the keyword substitutions to take place on the various templates later
3.3 XML SUBSTITUTION

in the wrapping procedure.

One final step is required before the final file can be used within the rest of the wrapping procedure: template variable resolution. While each class instance has its associated inputs, outputs, methods, and method arguments, some vary depending on the class instance. Comparisons between each class instance are made to observe if/when class instance information changes depending on the instance. In such cases, the retrieved information is replaced with a template variable to be recognised and accounted for later in the wrapping procedure. An example of the final SimWrapITK XML can be seen in Appendix F, Listing F.3.

3.3 XML Substitution

With the XML representation for the classes created, the templates for the Virtual Block, MATLAB, and Simulink files can be substituted with the XML information and then compiled to complete the wrapping and integration process. This section will describe the various templates and the role they serve within SimITK. A description of the perl scripts used to substitute the information is outlined in Section 3.4.1.

The technique employed in substituting the various templates with the appropriate information seeks out special keywords within the templates and replaces them with appropriate XML information and/or extra code. To ensure unique keywords, the variables were prepended and appended with an ampersat, or “at symbol” (@). Examples of these variables can be seen within the file templates included in Appendix F (Appendix Section F.4 provides a clear example of the custom-keyword usage). The various file templates and their role in the wrapping procedure will be outlined next in the following sections.
3.3. XML SUBSTITUTION

It is worth noting that though the Image File Readers/Writers, CenteredTransformInitializer, and NodeContainer classes were treated as special cases, they follow the same overall method with the only exception of having their own specific file template and respective substitution series, respectively.

3.3.1 Virtual Block .tpp Generation

As outlined in Figure 3.1, the “Virtual Block” .tpp is the first file template substituted with the class XML data and subsequently compiled. The .tpp file serves as a communication layer between the Simulink and ITK workspaces. The Virtual Block is aptly named as it exists transparently to the user yet serves the important task of converting special ITK-datatypes into appropriate Simulink-datatypes, and vice versa, as Simulink is only capable of manipulating a few specific datatypes.

On a technical note with regards to the reading and writing of image data, the primary data representation within MATLAB is a pointer to a flat array in memory. As expected, any loaded image data exists within MATLAB-allocated memory. Since SimITK generates ITK code that also uses flat arrays to store the pixel values of image data, it was critical to ensure no duplication between Simulink and ITK memory occurred as this would introduce significant slowdown and redundancy. To allow the quickest access to the same image data by both Simulink and ITK, the Virtual Block was purposefully designed such that the image data are passed as memory pointers from the MATLAB-allocated Simulink memory to ITK (i.e. the image data are only ever loaded once and both ITK and Simulink operate on that same block of memory). Two ports are created for each block that handles image data. The “info” port contains image data details, such as image size and pixel/voxel spacing,
while the “data” port contains the actual data values stored within each pixel. Each output port must be connected to its subsequent corresponding input port to ensure the image data are properly passed between blocks.

Furthermore, the technique of supplying memory pointers across the ITK and Simulink spaces extends to cases like Metric, Transform, and Optimizer blocks that have a “Self” output port. These blocks do not need their outputs converted between ITK and Simulink to function within a workflow; rather, these blocks require their datatypes to be supplied as-is. In these cases, the “Self” port acts to supply the subsequent block with the memory address of the previous block to provide its functionality within the workflow.

The Virtual Block is also responsible for instantiating the ITK class that is created at simulation runtime as well as ensuring that all the arguments specified within the workflow are established as specified by the user. The .tpp file for a given class is required as a dependency in the creation and generation of all subsequent files in the wrapping procedure. The .tpp template used for wrapping filters is included in Appendix F, Listing F.2.

3.3.2 Simulink S-Function .cpp Generation

With the .tpp file generated, the next file to be keyword-substituted is the Simulink S-Function .cpp file template. This file represents the code that will be executed by MATLAB/Simulink when running a block workflow. The reason for the future tense is that the generated .cpp file must still be compiled before it can be used by Simulink. This file is divided into multiple methods that are executed at different steps when executing a Simulink workflow. The methods of particular interest are:
3.3. XML SUBSTITUTION

mdlInitializeSizes, mdlInitializeSampleTimes, mdlStart, mdlOutputs, mdlSetInputPortDimensionInfo, mdlSetOutputPortDimensionInfo, mdlSetDefaultPortDimensionInfo, and mdlTerminate, which will be elaborated upon below.

On a technical note, there are constant calls to the ssGetUserData(S) and ssSetUserData(S, filter) methods throughout the .cpp file. This is to ensure that all the block information is carried between the aforementioned list of Simulink methods. The block information is stored within the S-Function itself, not within any one method, so it is necessary to Get and Set the status of the Simulink block as changes are made (e.g. virtual block instantiation, selection of methods, setting method parameters, etc).

The mdlInitializeSizes method is executed at the 0th iteration of the workflow and is a compile-time requisite that initialises the sizes of the block ports: the inputs and outputs to a given block that can be connected to other blocks to form the pipeline. Simulink requires that the “width” of this port (the number of elements that comprise the generated/received signal) be declared at compile-time, not runtime. This can lead to difficulties when the port size must vary with the dimensionality of the block. For example, a 2D block will have fewer elements than a 3D block as there is one less dimension to handle; thus, the number of elements cannot remain a fixed value. In such cases, like Transform classes, it was necessary to create a temporary instance of the class, retrieve the appropriate number of elements, create the port with this width and then delete the temporary instance.

The mdlInitializeSampleTimes method is also executed at the 0th iteration of the workflow and instructs Simulink to treat the block as if it has a continuous signal with no time offset. This ensures that the block will always be taking all inputs and
sending all outputs continuously as the workflow is run without interruption.

The **mdlStart** method is executed once at the 0\(^{th}\) iteration of the workflow and is used within SimITK as the location where the instance of the Virtual Block representation of the class is created as well as where specific ITK-type definitions are declared.

The **mdlOutputs** method is executed on every iteration of workflow execution, including the 0\(^{th}\) iteration. This is where the bulk of the executed code can be found. All methods with corresponding user-selected parameters within the Block Mask are transferred to the Virtual Block representation for execution within the ITK workspace. It is also within this method that the ITK class is executed. It is worth noting that all methods require two parts to function with the Simulink GUI: an indicator variable, to determine whether or not the user wishes to use said method within the workflow; and the actual method argument itself. Unless the indicator variable is set, the user will be unable to set the given method argument. This will be explained further within the description of the `.m` file in Section 3.3.3.

The **mdlSetInputPortDimensionInfo**, **mdlSetOutputPortDimensionInfo**, and **mdlSetDefaultPortDimensionInfo** methods are included within the default Simulink S-Function template. Though they remain empty and unused, they are required to be within the template to ensure proper Simulink execution.

The **mdlTerminate** method is executed once at the final iteration of the workflow and deletes all of the instantiated virtual block representations created during workflow execution. The signal propagates throughout the workflow such that the first blocks in the pipeline are deleted and causes a cascading effect through the rest of the workflow until all Virtual Block representations have been removed so the workflow
Table 3.1: Execution Times of S-Function Methods

<table>
<thead>
<tr>
<th>Name of Method</th>
<th>When Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdlInitializeSizes</td>
<td>$0^{th}$ iteration</td>
</tr>
<tr>
<td>mdlInitializeSampleTimes</td>
<td>$0^{th}$ iteration</td>
</tr>
<tr>
<td>mdlStart</td>
<td>$0^{th}$ iteration</td>
</tr>
<tr>
<td>mdlOutputs</td>
<td>Every iteration ($0^{th}$ inclusive)</td>
</tr>
<tr>
<td>mdlTerminate</td>
<td>Final iteration</td>
</tr>
</tbody>
</table>

can finish in a completed state.

For reference, the iterations at which the various S-Function methods are executed can be seen in Table 3.1.

Once substituted with the XML information, the .cpp file is ready to be compiled into its final .mex file — the file executed at Simulink runtime.

3.3.3 MATLAB Callback .m Generation

The MATLAB Callback .m file contains the code necessary to alter the GUI elements within the block “mask”: the dialog box the user controls to customise a given ITK class’ method arguments. This includes the enabling of desired class methods and, if required, establishing additional input ports (in the case of Transform blocks) or output ports (in the case of Transform and Optimizer blocks).

3.3.4 Simulink Mask .mdlpart Generation

Coupled with the .m file that contains the code for manipulating these GUI elements such that they appear and disappear, the .mdlpart file contains the actual
code representation the GUI elements themselves such as text-entry fields, checkboxes, pull-down menus, and input/output ports. Figure 4.5 (found on page 58) is an example mask dialog box of a `itkCenteredEuler3DTransform3D` block.

### 3.3.5 Simulink Library `.mdl` Generation

One `.mdl` file is created for each combination of Image Filter datatype and dimensionality, Transform dimensionality, and one file for all Optimizers. The `.mdlpart` files are joined into one consecutive string that is then substituted into the `.mdl` library file template to generate the final library file (Figure 4.1, page 54); completing the library generation process.

### 3.4 Build Automation

Creation of the various files are handled individually by the previously mentioned collection of `perl` scripts and modules. In order to automate the process, a build script was written in CMake, the same build-language used when configuring ITK, that takes all the input XML and processes them into each template by executing the `perl` scripts/modules in sequence, appends the `.mdlpart` files together into libraries, and finally compiles each Simulink S-Function `.cpp` file into its final `.mex` MATLAB-executable library.

#### 3.4.1 Perl Scripts

A series of `perl` scripts and modules were written to substitute the keywords within the templates with the appropriate XML information. The main `perl` script responsible for managing the substitutions is `BlockGenerator.pl` that requires a “filetype
3.4. BUILD AUTOMATION

Table 3.2: Perl Scripts/Modules and Associated Filetypes

<table>
<thead>
<tr>
<th>Name of Script/Module</th>
<th>Filetype</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockGenerator.pl</td>
<td>All Files</td>
</tr>
<tr>
<td>TPPGen.pm</td>
<td>.tpp Files</td>
</tr>
<tr>
<td>SFunctionGen.pm</td>
<td>.cpp Files</td>
</tr>
<tr>
<td>MatlabCallbackGen.pm</td>
<td>.m, .mdlpart Files</td>
</tr>
<tr>
<td>itkLibraryGen.pl</td>
<td>.mdl Files</td>
</tr>
</tbody>
</table>

flag” associated with the file to-be generated (.tpp, .cpp, .m, or .mdlpart). The corresponding module is loaded and executed. The perl modules required to generate each respective file can be seen in Figure 3.1.

Creation of the final .mdl files is handled by another script, itkLibraryGen.pl that takes all corresponding .mdlpart files for the given .mdl (i.e. all .mdlparts for the same Image Filter datatype and dimensionality, Transform dimensionality, or Optimizers), and concatenates them into the data that will be substituted into the .mdl template using the same custom-keyword substitution technique as outlined previously.

3.4.2 Compiled MATLAB .mex Files

These are the final .cpp files compiled into MATLAB .mex files executed at runtime. Once the .cpp files are created, they require being compiled by MATLAB in order to complete the integration. This is handled by the CMake build script outlined in the following section.
3.4. BUILD AUTOMATION

3.4.3 CMake

CMake\(^2\) [27] is a build language created by Kitware Inc. used to automate the build and compilation of ITK. Because of this dependency, CMake was selected as the natural build-language for SimITK.

A CMake build script, CMakeLists.txt, was written that processed the SimWrapITK XML file names and created the build rules required to generate the final Simulink blocks. As outlined in Figure 3.1, certain files (like the .cpp, for instance) require its .tpp file to be first generated. CMake allows for the construction of a “dependency tree” that instructs the system to build and compile files in the given order. As outlined within the build workflow as shown in Figure 3.1, this same build order is reflected within CMake-code within CMakeLists.txt.

\(^2\)http://www.cmake.org/
Chapter 4

Results

The results from the SimITK wrapping procedure are divided into several sections: the generation of the SimWrapITK XML, the creation of the final Simulink Block libraries, the discussion of a custom-class created for use in SimITK: `StepByStepImageRegistrationMethod`, presentation of the runtime performance of SimITK compared to pure C++ ITK implementations, as well as a brief overview of the website created to promote and support the project.

4.1 SimWrapITK XML Generation from WrapITK

As previously described in Section 3.2, and shown in greater depth in Appendix F, Listing F.1, the `convertWrapITKtoSimITK.pl` script is executed to generate SimWrapITK class representations of previously generated WrapITK XML documents. From these generated SimWrapITK files, the information therein can be successfully substituted into the templates for the several required files, and compiled when necessary, to complete the SimITK wrapping procedure. Following this step, construction of SimITK workflows is possible within MATLAB/Simulink.
4.2 Simulink Block Libraries

The primary result from the SimITK wrapping procedure is the generation of multiple Simulink Block libraries. Eight Image Filter libraries are created: one for each combination of datatype (float, short, unsigned short, unsigned char) and dimensionality (2D or 3D); two Transform libraries: one for each dimensionality; and one library for Optimizers used in registration frameworks. A total of 129 Image Filters, 6 Transforms, and 7 Optimizers were wrapped in entirety. A screenshot of an example Image Filter library can be seen in Figure 4.1 and a list of all the wrapped classes can be found in Appendix C.

4.2.1 Workflow Creation

With the libraries generated, it is possible to create SimITK “workflows”, Simulink models composed of SimITK blocks that perform a given task. The first required step is to select and place desired blocks as objects within a new Simulink model file. This is accomplished through clicking a given block and dragging it from the library onto the Simulink model workspace, or “canvas”, as shown in Figure 4.2.

With the desired blocks placed on the canvas, inter-block connections can be established that will pass information (like image data) from block to block in order to process the data as directed by the workflow. This is achieved by first selecting where a connection is to be made (Figure 4.3(a)), hovering the cursor on the output port of the origin block (causing a visual change from a pointer in Figure 4.3(b) to a cross in Figure 4.3(c)), clicking and dragging the cursor to the desired input port on the destination block (Figure 4.3(d)), and releasing the mouse button to complete the connection (Figure 4.3(e)). Alternatively, the origin block can be selected, the
4.2. SIMULINK BLOCK LIBRARIES

Figure 4.1: An Example SimITK Library.
Figure 4.2: A graphical representation of adding a block to a workflow canvas.

Control (\texttt{Ctrl}) key held, and the destination block then clicked. This process must be repeated until all blocks within a workflow are fully connected. Figure 4.4 is an example of a fully-connected workflow.

The final step in workflow creation, prior to execution, is to set the desired method arguments for each class. This is accomplished through double-clicking a given block to reveal its Simulink “mask dialog”, the control interface for the block (Figure 4.5).
4.2. SIMULINK BLOCK LIBRARIES

Figure 4.3: The steps to establishing an inter-block connection.

(a) Selecting a connection to be made.

(b) Move to the output port on the origin block.

(c) The cursor becomes a cross instead of a pointer.

(d) Click and drag to the input port on the destination block.

(e) Release the mouse button and the connection is created.
4.2. SIMULINK BLOCK LIBRARIES

Figure 4.4: An graphical demonstration of established block connections on a completed workflow.

In this dialog box, desired methods can be enabled for use within the workflow. When enabled, a text-input field is revealed where desired method arguments can be entered. In cases like the Transform and Optimizer blocks, the mask contains checkboxes that can enable/disable the input or output of additional information. Figure 4.5 is an example of an itkCenteredEuler3DTransform3D block mask where multiple inputs and an output can be specified, if desired.

As a further example, Figure 4.6 shows the same Transform block in several configurations depending on the enabled checkboxes within the mask.

Once a given workflow has been built of properly-connected blocks, and corresponding mask dialogs adjusted as needed, the Simulink timer, or “clock”, needs to be set to the desired number of full workflow executions. It is worth noting that the 0th iteration is defined in Simulink to be the first iteration, so a clock count of 0 runs a workflow through one complete execution. With the clock set, Simulink can be instructed to start the simulation (i.e. execute the workflow and perform the ITK image processing task).
Figure 4.5: An example mask dialog.
4.2. SIMULINK BLOCK LIBRARIES

Two in-depth examples are presented as case studies for SimITK. One demonstrates segmentation using pixel-intensity thresholding and another demonstrates MRI-to-CT registration using a Mattes Mutual Information Metric, Gradient Descent Optimization, and Nearest Neighbour Interpolation. The reader is encouraged to revisit Section 2.3 for an overview of the terminology and components involved in typical segmentation and registration programs, if needed.

With respect to the data used in the following examples, the segmentation cranial data (Figure 4.7, lower-left) can be found within the Examples directory of a base ITK installation, and the registration cranial CT (Figure 4.9(a)) and MRI data (Figure 4.9(b)) can be acquired from the American National Institute of Health Visible Human Project.

4.2. SIMULINK BLOCK LIBRARIES

4.2.2 Case Study Example: 3D Cranial Segmentation Workflow

An example SimITK workflow that segments the skull from cranial CT data is shown in Figure 4.7. The first block is a file reader used to load the 3D image data and pass the information to a threshold filter that replaces the pixels below the user-set pixel-intensity threshold level with black, while highlighting the bony areas above the threshold in white. This modified volume is then written to disk for future use.

As a code comparison, the 86 lines of C++ are represented by the connection of three (3) SimITK blocks to form this workflow. Appendix A contains a comparison of the required ITK-C++ code and equivalent SimITK workflow.

4.2.3 Case Study Example: 3D Cranial MRI to CT Registration Workflow

Another workflow example performs a registration of MRI and CT data of the same cranium is shown in Figure 4.8 (corresponding data and output can be seen in Figures 4.9 and 4.10). Like in the previous example, one file reader is used to load each of the MRI and CT data as separate inputs to a registration method using a Centered 3D Euler transform, Nearest Neighbour interpolation, Mattes Mutual Information metric, and Gradient Descent registration optimisation. Following registration, the MRI data are processed with the optimally-calculated registration parameters before being written to disk as a new file. Upon completion, the final value for the Metric (in violet) and the Transform parameters (in yellow) used to create the optimal alignment will be printed in their respective labelled blocks in the workflow.

As a code comparison, the 296 lines of C++ are represented by the connection of fifteen (15) SimITK blocks to form a workflow. Appendix B contains a comparison
Figure 4.7: An example workflow segmenting the skull from 3D Cranial Volume data. This three-block workflow replaces approximately 90 lines of C++ code.

between the required ITK-C++ code and equivalent SimITK workflow.

4.3 StepByStepImageRegistrationMethod

To complement the automatically wrapped built-in ITK classes, a custom ITK class, StepByStepImageRegistrationMethod, was created for inclusion with SimITK that allows Simulink to collect information like Metric value(s), Transform parameters,
Figure 4.8: An example workflow registering 3D MRI cranial data to 3D CT data. The fifteen blocks illustrated here replace approximately 300 lines of C++ code.
4.3. **STEPBYSTEPIMAGEREGISTRATIONMETHOD**

(a) The CT data were processed into a `.vtk` volume file of 129 slices (at 1 mm thickness) of 245-by-255 pixel images. One pixel is equivalent to 0.898438 mm.

(b) The fresh T1 MRI data were processed into a `.vtk` volume file of 34 slices (at 4 mm thickness) of 128-by-128 pixel images. One pixel equivalent to 1.01562 mm.

Figure 4.9: Reference images of the initial CT and MRI data used in the Registration example (Figure 4.8).

and any other desirable intermediary results, during workflow execution. Functionally, the class operates identically to the standard ITK-supplied `itkImageRegistrationMethod` except that the iteration of the ITK optimizer is tied to the iteration of the Simulink clock. Typically, the Simulink clock is responsible for establishing the number of desired full workflow executions; `StepByStepImageRegistrationMethod` modifies this behaviour such that one Simulink clock iteration is equivalent to one ITK registration iteration. This allows the user to retrieve and store registration parameters from within the MATLAB/Simulink workspace in real time (i.e. with each iteration progressing towards registration completion).

On a technical note, `StepByStepImageRegistrationMethod` is only compatible with `itkRegularStepGradientDescentBaseOptimizer`-derived optimizer classes as they are capable of starting, stopping, and resuming ITK registrations.

Runtime performance comparisons of both image registration methods (built-in
4.3. STEP BY STEP IMAGE REGISTRATION METHOD

(a) Unregistered MRI data visualised in red.  
(b) The registered MRI data visualised in green.  
(c) An overlay of the pre- and post-registration MRI data.  
(d) The CT data and unregistered MRI data overlay in red.  
(e) The CT data and registered MRI data overlay in green.

Figure 4.10: A comparison of the CT and MRI datasets pre- and post- execution of the registration implementations.
and custom) are presented next in Section 4.4.

4.4 Run-time Performance

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 the same execution point within each respective code to ensure comparability.

For the segmentation workflow (Figure 4.7), the run times are considered equivalent in both pure, written C++ ITK-code and SimITK flowchart. The determined difference in runtime was on the order of milliseconds. As such, implementing code to calculate a sub-millisecond time difference would have introduced time overhead well above this requirement rendering any results without merit.

The registration example (Figure 4.8) results for both the standard itkImageRegistrationMethod and the custom StepByStepImageRegistrationMethod are shown in Table 4.1, which compares SimITK workflow runtimes as compared to their pure-C++ equivalent. All implementations were executed for the same number of iterations and yielded the same final registration parameters.

It is worth noting that the decrease in runtime observed in Table 4.1 was not anticipated. Since the runtime comparison code behaved identically in each case (as the timestamp code executed was verbatim at equivalent points in each respective code), it is hypothesised that any discrepancy is due to MATLAB pre-allocating memory for all SimITK image storage, while pure ITK must make system calls to initialise, allocate, and manage image memory.
### 4.5. PROJECT WEBSITE (WWW.SIMITKVTK.COM)

Table 4.1: Comparison of MRI to CT Registration code runtimes (100 executions, each consisting of 100 iterations, timed until completion)

<table>
<thead>
<tr>
<th></th>
<th>itkImageRegistrationMethod</th>
<th>StepByStepImageRegistrationMethod</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pure C++ ITK Program</strong></td>
<td>170 ± 1 seconds</td>
<td>600 ± 1 seconds</td>
</tr>
<tr>
<td><strong>SimITK Workflow</strong></td>
<td>158 ± 1 seconds</td>
<td>450 ± 1 seconds</td>
</tr>
<tr>
<td><strong>Time Difference</strong></td>
<td>12 seconds (8.8%)</td>
<td>150 seconds (25%)</td>
</tr>
</tbody>
</table>

Longer registration times are expected for the StepByStepImageRegistrationMethod compared to the standard itkImageRegistrationMethod. The former runs the registration for only a single optimizer iteration before returning control to Simulink in order to facilitate any information retrieval prior to resuming registration, resulting in a bottleneck and overall slowdown of program execution. The latter is coded to run the registration until completion without interruption. While it is possible to include command/observer classes in pure C++ programs that can retrieve information as the registration executes, these classes cannot be used to serve this same purpose within Simulink.

#### 4.5 Project Website (www.SimITKVTK.com)

A website\(^2\) was created to promote SimITK, as well as its sister-project SimVTK\(^3\), and provide user and developer-level support. Present and past releases can be found on the website as well as extensive user and developer documentation.

---

\(^2\)http://www.simitkvtk.com/

\(^3\)A project that aims to create Simulink blocks of functionalities from the Visualization Toolkit (VTK), another C++ library produced by Kitware Inc. used to visualise image data, in the same way SimITK blocks are created.
The documentation includes installation and configuration instructions for Sim-ITK, as well as any prerequisite software, as well as tutorials and examples that use provided image data to guide a user through workflow creation. All examples are also supplied as pre-built Simulink files for immediate use. Demonstration videos, previous publications, and contact information are also included.
Chapter 5

Conclusions and Future Work

5.1 Summary of Conclusions

In this work, SimITK was presented: a collection of block libraries that can be connected within the Simulink visual programming environment to create workflows that parallel their equivalent pure C++ ITK code. Constructing the workflow visually, instead of building a written pipeline of code, allows the user to focus on solving the image processing problem with simple, intuitive graphical representations while not having to be concerned with potentially difficult programming nuances.

SimITK allows for the image processing capabilities of ITK to be made readily available to users that would normally have to invest substantial time and effort into learning the details of ITK, and potentially high-level C++. This makes SimITK an ideal environment for educators hoping to teach the concepts of image processing within a classroom setting. Medical research groups can also use SimITK to test hypotheses by rapidly develop workflows as a starting point for experimentation.

SimITK was achieved through the accomplishment of four distinct goals:
5.2. FUTURE WORK

1. The creation of an organisational schema, or information layout, that was capable of storing a representation of required ITK class information (such as name, input(s), output(s), permitted datatype(s), acceptable dimensionality/ies, class method(s), method argument(s), etc) for later retrieval when creating Simulink blocks.

2. Automatic generation of one schema file per desired ITK class.

3. Creating Simulink Block Libraries from these previously generated schema files, as well as any and all additional files required to complete integration, for future use in SimITK workflows.

4. Evaluating that SimITK workflow and pure C++ ITK program performance was equivalent in terms of functionality for the selected case studies.

While this work is proof-of-concept that a subset of approximately 130 classes of ITK can be successfully wrapped and integrated into SimITK, there are areas for further development that would increase the impact and overall abilities of SimITK.

5.2 Future Work

In this section, areas for future development will be outlined including their priority, the benefits the development would bring, as well as implementation recommendations.
5.2. FUTURE WORK

5.2.1 Increase Number of Wrapped ITK Classes

At present, only a subset (approximately 130 of 2700) of ITK classes have been automatically wrapped into SimITK. A large, required development will be the expansion of the Virtual Block .tpp files as they are presently capable of handling only a subset of all the defined data structures and types specific to ITK. Presently, this consists of image data in the form of ImageType, transform parameters in the form of ParametersType; metric values as a ValuesType; InterpolatorType, MetricType, OptimizerType, and TransformType as required for registrations. Many more types, on the order of tens, if not a couple hundred, still require appropriate conversion rules before they can be properly integrated into SimITK. As such, this has limited the total number of classes within SimITK to being approximately 130 including Image Filter, Optimizer, Metric, Interpolator, and Transform classes. Expanding SimITK such that all ITK types have been accounted for and can be converted to and from MATLAB/Simulink datatypes would be a logical next step in development and should also be considered a top priority.

5.2.2 Registration Workflows on non-Windows Platforms

Currently, SimITK is only capable of performing registration framework-based workflows on Windows-based computers. UNIX-based environments, including Mac OS X, are incapable of running these workflows due to the nature in which the final MATLAB .mex files are handled by the system. The root of the problem, described in the following paragraph, is complex and highly technical — as is the optimal solution.
The official FAQ for gcc\(^1\) outlines the issue that stems from resolving type equality. Specifically, gcc uses memory addresses for comparison instead of strings when determining type equality. Before an ITK class can be instantiated from within Simulink, its type must be first resolved. gcc requires that this resolution happen directly, without passing through an intermediary, because of the memory address comparison. Because the declaration of the ITK type is made within the Virtual Block, found as a part of the .mex file, gcc attempts to resolve the type through the intermediary, finds conflicting memory addresses, and fails.

The solution to this issue is equally as technical and is split into two parts. First, all current .tpp files must be split in two: a .h header file that contains the interface, or declarations, for the execution-code defined in a new, complimentary Virtual Block .cpp file. With .h files created for each wrapped ITK class, they can be combined with all of the files comprising ITK to create one all-encompassing ITK/SimITK shared library that the .mex files can directly link to without the need for an intermediary when resolving types. Thus, proper memory addresses will be found and type resolution will succeed.

It is worth reiterating that this is a complex issue that stems from a security consideration within the operating system and not Simulink, ITK, or MATLAB. Though the solution to this problem will likely be time-consuming, as it will require a total change in the implementation of the Virtual Block files and the base ITK installation one uses with SimITK. Nevertheless, running registration frameworks on UNIX-environments is a highly desirable functionality and should be considered a top priority.

\(^1\)The specific anchor to the FAQ point in question is: http://gcc.gnu.org/faq.html#dso
5.2. FUTURE WORK

5.2.3 Detailed Code Profiling of C++ Programs and SimITK Workflows

The runtime results shown in Table 4.1 were presented using a specific set of case studies running a specific hardware configuration. While the results do not demonstrably suggest that SimITK introduces time-overhead, the runtime performance has not been further analysed to ensure that similar results would be expected on other systems with configurations that differ from the development environment.

For further validation, it is recommended that the created C++ programs and equivalent SimITK workflows are processed using a code profiler. This would provide deeper insight with respect to the time discrepancies to give a more complete understanding as to the true nature of the runtime performance comparison between pure C++ ITK and SimITK.

5.2.4 Datatype Consolidation

At present, each different datatype and dimensionality yields its own Simulink Block Library. As such, a workflow created to segment, for example, three-dimensional float data cannot be used to segment short data; a duplicate workflow composed of blocks of the latter datatype must be created.

It is recommended that instead of resolving the template variables into their specific datatypes and dimensionalities at compilation-time, they should be left as-is (i.e. unresolved). Workflows could then be coupled with an initial block (not presently developed) that would establish the desired datatype for workflow execution.

This development would greatly increase the reusability of a created workflow allowing it to work for each of the wrapped datatypes by changing only one block instead of recreating entire workflows for each datatype, when needed.
5.2. FUTURE WORK

5.2.5 Transition from perl to Python

While the perl scripts execute as desired for the current state of SimITK, rewriting the scripts that perform the custom-keyword substitutions to generate the class-specific SimITK .tpp, .cpp, .m, and .mdl files should also be considered, primarily for maintainability reasons. Inherently, perl was designed to be a flexible language in that the same task could be written in (potentially) a number of different ways. This aspect of perl can yield code extremely difficult to maintain, especially in the case where a previous developer is no longer a contributor to the project. Python is recommended as a candidate language to replace all perl code as it yields highly legible code that can be very easily understood, expanded, and maintained. While not considered as high a priority when compared to the previous areas for development, this could be implemented in parallel when more ITK datatypes are encapsulated.


[38] Ivo Wolf, Marco Nolden, Tobias Schwarz, and Hans-Peter Meinzer. Integrating the visualization concept of the Medical Imaging Interaction Toolkit (MITK) into the XIP-Builder visual programming environment. SPIE, 7625(1), 2010.


Appendix A

Simple ITK C++/SimITK Comparison:

Segmentation

Below is the code for a simple segmentation in both ITK C++ and equivalent SimITK workflow. As a code comparison, the 86 lines of C++ are represented by the connection of three (3) SimITK blocks to form a workflow.

Listing A.1: C++ Code for Creation of Pixel-Intensity Threshold Program

```c++
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkBinaryThresholdImageFilter.h"
#include "itkImage.h"
void timestamp()
{
  time_t ltime; /* calendar time */
ltime=time(NULL); /* get current cal time */
  printf("%s",asctime(localtime(&ltime)));
}
int main( int argc, char * argv[] )
{
  timestamp();
  // Verify the number of parameters in the command line
  if( argc < 3 )
    { std::cerr << "Usage:" << std::endl;
```
std::cerr << argv[0] << " inputImageFile outputImageFile " << std::endl;
return EXIT_FAILURE;
}

typedef unsigned short InputPixelType;
typedef unsigned short OutputPixelType;
typedef itk::Image< InputPixelType, 3 > InputImageType;
typedef itk::Image< OutputPixelType, 3 > OutputImageType;
typedef itk::ImageFileReader< InputImageType > ReaderType;
typedef itk::ImageFileWriter< OutputImageType > WriterType;
typedef itk::BinaryThresholdImageFilter< InputImageType, OutputImageType > FilterType;

ReaderType::Pointer reader = ReaderType::New();
FilterType::Pointer filter = FilterType::New();
WriterType::Pointer writer = WriterType::New();

const char * inputFilename = argv[1];
const char * outputFilename = argv[2];
reader->SetFileName( inputFilename );
writer->SetFileName( outputFilename );
writer->SetInput( filter->GetOutput() );
filter->SetInput( reader->GetOutput() );

const OutputPixelType insideValue = 0;
const OutputPixelType outsideValue = 255;
const InputPixelType lowerThreshold = 0;
const InputPixelType upperThreshold = 1105;

// Set Filter Parameters
filter->SetInsideValue( insideValue );
filter->SetLowerThreshold( lowerThreshold );
filter->SetOutsideValue( outsideValue );
filter->SetUpperThreshold( upperThreshold );

filter->Update();

try
{
    writer->Update();
}
catch( itk::ExceptionObject & err )
{
    std::cerr << "ExceptionObject caught!" << std::endl;
    return EXIT_FAILURE;
}
timestamp();
return EXIT_SUCCESS;
Figure A.1: Equivalent SimITK Pixel-Intensity Threshold Workflow.
Appendix B

Complex ITK C++/SimITK Comparison: Registration

Below is the code for a more complex registration framework in both ITK C++ and equivalent SimITK workflow. As a code comparison, the 296 lines of C++ are represented by the connection of fifteen (15) SimITK blocks to form a workflow.

Listing B.1: C++ Code for Creation of an MRI-to-CT Registration Program

```cpp
#ifdef defined(MSC_VER)
#pragma warning ( disable : 4786 )
#endif

// Transform
#include "itkCenteredEuler3DTransform.h"

// Interpolator
#include "itkNearestNeighborInterpolateImageFunction.h"

// Metric
#include "itkMattesMutualInformationImageToImageMetric.h"

// Optimizer
#include "itkRegularStepGradientDescentOptimizer.h"

// Registration
#include "itkCenteredTransformInitializer.h"
#include "itkImageRegistrationMethod.h"

// Filters
#include "itkResampleImageFilter.h"
#include "itkSquaredDifferenceImageFilter.h"

// IO
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkFileOutputWindow.h"
```
// Image
#include "itkImage.h"

//include <iostream>

// The following piece of code implements an observer
// that will monitor the evolution of the registration process.

#include "itkCommand.h"

class CommandIterationUpdate19 : public itk::Command
{
public:
  typedef CommandIterationUpdate19 Self;
  typedef itk::Command Superclass;
  typedef itk::SmartPointer<Self> Pointer;
  itkNewMacro(Self);
protected:
  CommandIterationUpdate19() {};
public:
  typedef itk::RegularStepGradientDescentOptimizer OptimizerType;
  typedef const OptimizerType * OptimizerPointer;

  void Execute(itk::Object * caller, const itk::EventObject & event)
  {
    Execute( (const itk::Object *) caller, event);
  }

  void Execute(const itk::Object * object, const itk::EventObject & event)
  {
    OptimizerPointer optimizer =
      dynamic_cast<OptimizerPointer > (object);
    if(! itk::IterationEvent().CheckEvent(&event ) )
    {
      return;
    }
    std::cout.precision(15);
    std::cout.setf(std::ios::fixed, std::ios::floatfield);
    std::cout << optimizer->GetCurrentPosition() << std::endl;
  }

};

void timestamp()
{
  time_t ltime; /* calendar time */
  ltime=time(NULL); /* get current cal time */
  printf("%s",asctime(localtime(&ltime) ) );
}

int main(int argc, char * argv[])
{
  timestamp();
  // Verify the number of parameters in the command line
  if( argc < 4 )
  {
    std::cerr << "Usage: " << std::endl;
    std::cerr << argv[0] << " inputFilenameCT inputFilenameMR
           outputFilename" << std::endl;
    return EXIT_FAILURE;
  }
  // Store command line arguments for use in readers and writer
  const char * inputFilenameCT = argv[1];
}
const char * inputFilenameMR = argv[2];
const char * outputFilename = argv[3];

// Define and create CT data reader
typedef short InputPixelType_CT;
typedef itk::Image< InputPixelType_CT , 3 > InputImageType_CT;
typedef itk::ImageFileReader< InputImageType_CT > ReaderType_CT;
ReaderType_CT::Pointer readerCT = ReaderType_CT::New();
readerCT->SetFileName( inputFilenameCT );

// Define and create MR data reader
typedef short InputPixelType_MR;
typedef itk::Image< InputPixelType_MR , 3 > InputImageType_MR;
typedef itk::ImageFileReader< InputImageType_MR > ReaderType_MR;
ReaderType_MR::Pointer readerMR = ReaderType_MR::New();
readerMR->SetFileName( inputFilenameMR );

// Define and create registered MR data writer
typedef OutputPixelType;
typedef itk::Image< OutputPixelType , 3 > OutputImageType;
typedef itk::ImageFileWriter< OutputImageType > WriterType;
WriterType::Pointer writer = WriterType::New();
writer->SetFileName( outputFilename );

itk::FileOutputWindow::Pointer fow = itk::FileOutputWindow::New();
fow->SetInstance( fow );

const unsigned int Dimension = 3;
typedef PixelType;
typedef itk::Image< PixelType , Dimension > FixedImageType;
typedef itk::Image< PixelType , Dimension > MovingImageType;
typedef itk::CenteredEuler3DTransform< double > TransformType;
typedef itk::RegularStepGradientDescentOptimizer OptimizerType;
typedef itk::MattesMutualInformationImageToImageMetric< FixedImageType ,
                                                      MovingImageType >
                                                     MetricType;
typedef itk::NearestNeighborInterpolateImageFunction< MovingImageType ,
                                                      double > InterpolatorType;
typedef itk::ImageRegistrationMethod< FixedImageType , MovingImageType >
                                                      RegistrationType;

MetricType::Pointer metric = MetricType::New();
TransformType::Pointer transform = TransformType::New();
OptimizerType::Pointer optimizer = OptimizerType::New();
InterpolatorType::Pointer interpolator = InterpolatorType::New();
RegistrationType::Pointer registration = RegistrationType::New();

metric->SetNumberOfSpatialSamples( 1250000 );
metric->SetNumberOfHistogramBins( 32 );
metric->SetUseAllPixels( 0 );

// Setup registration
registration->SetMetric( metric );
registration->SetOptimizer( optimizer );
registration->SetTransform( transform );
registration->SetInterpolator( interpolator );
registration->SetFixedImage( readerCT->GetOutput() );
registration->SetMovingImage( readerMR->GetOutput() );

// update Readers
readerCT->Update();
readerMR->Update();
```cpp
// Set up transform initializer
typedef itk::CenteredTransformInitializer<
    TransformType, FixedImageType,
    MovingImageType > TransformInitializerType;
TransformInitializerType::Pointer initializer =
    TransformInitializerType::New();

typedef RegistrationType::ParametersType ParametersType;
ParametersType initialParameters = transform->GetParameters();

// Set up Optimizer
const unsigned int numberOfParameters = transform->GetNumberOfParameters();
OptimizerType::ParametersType simplexDelta(numberOfParameters);
optimizer->SetMaximumStepLength(15);
optimizer->SetMinimumStepLength(0.00001);
optimizer->SetNumberOfIterations(200);
optimizer->SetRelaxationFactor(0.6);

typedef OptimizerType::ScalesType OptimizerScalesType;
OptimizerScalesType optimizerScales(numberOfParameters);
optimizerScales[0] = 1.0;
optimizerScales[1] = 1.0;
optimizerScales[2] = 1.0;
optimizerScales[3] = 0.01;
optimizerScales[4] = 0.01;
optimizerScales[5] = 0.01;
optimizerScales[6] = 0.0001;
optimizerScales[7] = 0.0001;
optimizerScales[8] = 0.0001;
optimizer->SetScales(optimizerScales);

// Observer code - Uncomment to follow the registration.
// CommandIterationUpdate19::Pointer observer = CommandIterationUpdate19::New();
// optimizer->AddObserver(itk::IterationEvent(), observer);

initializer->SetTransform(transform);
initializer->SetFixedImage(readerCT->GetOutput());
initializer->SetMovingImage(readerMR->GetOutput());
initializer->GeometryOn();
initializer->InitializeTransform();

registration->SetInitialTransformParameters(transform->GetParameters());

try
{
    // print out the initial metric value, need to initialize the
    // registration method to force all the connections to be established.
    registration->Initialize();
    metric->ReinitializeSeed(5);
    std::cout << "InitialMetricValue = "
    << metric->GetValue(initialParameters) << std::endl;
    std::cout << "InitialOptimizerGetCurrentPosition = "
    << optimizer->GetCurrentPosition() << std::endl;
    registration->Print(std::cout);
    // run the registration
    registration->StartRegistration();
    std::cout << "OptimizerStopCondition = "
    << registration->GetOptimizer()->GetStopConditionDescription()};
```
ParametersType finalParameters = registration->GetLastTransformParameters();

const double bestValue = metric->GetValue(finalParameters);

// Print out results

std::cout << "Result = " << std::endl;
std::cout << "Final Metric Value = " << bestValue << std::endl;
std::cout << "Angle X (in rad) = " << finalParameters[0] << std::endl;
std::cout << "Angle Y (in rad) = " << finalParameters[1] << std::endl;
std::cout << "Angle Z (in rad) = " << finalParameters[2] << std::endl;
std::cout << "Center Coord X = " << finalParameters[3] << std::endl;
std::cout << "Center Coord Y = " << finalParameters[4] << std::endl;
std::cout << "Center Coord Z = " << finalParameters[5] << std::endl;
std::cout << "Translation X = " << finalParameters[6] << std::endl;
std::cout << "Translation Y = " << finalParameters[7] << std::endl;
std::cout << "Translation Z = " << finalParameters[8] << std::endl;

typedef itk::ResampleImageFilter< MovingImageType , FixedImageType > ResampleFilterType ;

TransformType::Pointer finalTransform = TransformType::New();
finalTransform->SetParameters( finalParameters );
finalTransform->Print( std::cout );
ResampleFilterType::Pointer resample = ResampleFilterType::New();
resample->SetTransform( finalTransform );
resample->SetInput( readerMR->GetOutput() );
resample->SetDefaultPixelValue( 0 );
resample->SetInterpolator( interpolator );
writer->SetInput( resample->GetOutput() );
writer->Update();
timestamp();
return EXIT_SUCCESS;
Figure B.1: Equivalent SimITK Registration Workflow.
Appendix C

List of Wrapped Classes

C.1 Image Filter Classes

Total: 129 classes.

• AbsImageFilter
• AccumulateImageFilter
• AcosImageFilter
• AdaptiveHistogramEqualizationImageFilter
• AnisotropicFourthOrderLevelSetImageFilter
• AntiAliasBinaryImageFilter
• ApproximateSignedDistanceMapImageFilter
• AreaClosingImageFilter
• AreaOpeningImageFilter
• AsinImageFilter
• AtanImageFilter
• BilateralImageFilter
• BinaryContourImageFilter
• BinaryMedianImageFilter
• BinaryMinMaxCurvatureFlowImageFilter
• BinaryProjectionImageFilter
• BinaryThresholdImageFilter
• BinaryThresholdProjectionImageFilter
• BinomialBlurImageFilter
• BoundedReciprocalImageFilter
• BoxImageFilter
• BoxMeanImageFilter
• BoxSigmaImageFilter
• BSplineDecompositionImageFilter
• CannyEdgeDetectionImageFilter
• CannySegmentationLevelSetImageFilter
• CastImageFilter
• CenteredTransformInitializer
• CheckerBoardImageFilter
• CollidingFrontsImageFilter
• ConfidenceConnectedImageFilter
• ConvolutionImageFilter
• CosImageFilter
• CurvatureAnisotropicDiffusionImageFilter
• CurvatureFlowImageFilter
• CurvesLevelSetImageFilter
• DanielssonDistanceMapImageFilter
• DerivativeImageFilter
• DifferenceImageFilter
C.1. IMAGE FILTER CLASSES

- DiscreteGaussianImageFilter
- DoubleThresholdImageFilter
- ExpandImageFilter
- ExpImageFilter
- ExpNegativeImageFilter
- FastApproximateRankImageFilter
- FastMarchingImageFilter
- FastMarchingUpwindGradientImageFilter
- FlipImageFilter
- GeodesicActiveContourLevelSetImageFilter
- GradientAnisotropicDiffusionImageFilter
- GradientMagnitudeImageFilter
- GradientMagnitudeRecursiveGaussianImageFilter
- GrayscaleFillholeImageFilter
- GrayscaleGrindPeakImageFilter
- HConcaveImageFilter
- HConvexImageFilter
- HistogramMatchingImageFilter
- HMaximaImageFilter
- HMinimaImageFilter
- ImageRegistrationMethod
- ImageToImageRegistrationHelper
- ImageToVectorImageFilter
- InvertIntensityImageFilter
C.1. IMAGE FILTER CLASSES

- IsolatedConnectedImageFilter
- IsolatedWatershedImageFilter
- IsotropicFourthOrderLevelSetImageFilter
- LabelContourImageFilter
- LabelVotingImageFilter
- LaplacianImageFilter
- LaplacianRecursiveGaussianImageFilter
- LaplacianSegmentationLevelSetImageFilter
- LaplacianSharpeningImageFilter
- LinearInterpolateImageFunction
- LogImageFilter
- MattesMutualInformationImageToImageMetric
- MaximumProjectionImageFilter
- MeanImageFilter
- MeanProjectionImageFilter
- MeanSquaresImageToImageMetric
- MedianImageFilter
- MedianProjectionImageFilter
- MinimumMaximumImageFilter
- MinimumProjectionImageFilter
- MinMaxCurvatureFlowImageFilter
- ModulusImageFilter
- MultiResolutionPyramidImageFilter
- NaryAddImageFilter
• NaryMaximumImageFilter
• NearestNeighborInterpolateImageFunction
• NeighborhoodConnectedImageFilter
• NoiseImageFilter
• NormalizeImageFilter
• NotImageFilter
• OtsuMultipleThresholdsImageFilter
• OtsuThresholdImageFilter
• ParallelSparseFieldLevelSetImageFilter
• RankImageFilter
• RecursiveGaussianImageFilter
• RecursiveMultiResolutionPyramidImageFilter
• ReflectImageFilter
• RegionalMaximaImageFilter
• RegionalMinimaImageFilter
• ResampleImageFilter
• RescaleIntensityImageFilter
• ShapeDetectionLevelSetImageFilter
• ShiftScaleImageFilter
• SigmoidImageFilter
• SignedDanielssonDistanceMapImageFilter
• SignedMaurerDistanceMapImageFilter
• SinImageFilter
• SmoothingRecursiveGaussianImageFilter
C.2. TRANSFORM CLASSES

- SobelEdgeDetectionImageFilter
- SparseFieldLevelSetImageFilter
- SqrtImageFilter
- SquareImageFilter
- StandardDeviationProjectionImageFilter
- STAPLEImageFilter
- StatisticsImageFilter
- StepByStepImageRegistrationMethod
- SumProjectionImageFilter
- TanImageFilter
- ThresholdImageFilter
- ThresholdMaximumConnectedComponentsImageFilter
- ThresholdSegmentationLevelSetImageFilter
- TileImageFilter
- ValuedRegionalMaximaImageFilter
- ValuedRegionalMinimaImageFilter
- VotingBinaryHoleFillingImageFilter
- VotingBinaryImageFilter
- VotingBinaryIterativeHoleFillingImageFilter

C.2 Transform Classes

Total: 6 classes.

- AffineTransform
- CenteredEuler3DTransform
- CenteredRigid2DTransform
C.3. OPTIMIZER CLASSES

- CenteredSimilarity2DTransform
- Similarity2DTransform
- TranslationTransform

C.3 Optimizer Classes

*Total: 7 classes.*

- AmoebaOptimizer
- ConjugateGradientOptimizer
- GradientDescentOptimizer
- LBFGSOptimizer
- QuaternionRigidTransformGradientDescentOptimizer
- RegularStepGradientDescentOptimizer
- VersorTransformOptimizer
Appendix D

User Installation Documentation

D.1 Installation Prerequisites

• 2 GB at minimum for disk space

• MATLAB R2008b (or later)

D.2 Installation Instructions

Downloading and extracting SimITK and subsequently configuring MATLAB to work with SimITK can be accomplished by the following:

1. Download latest binaries of SimITK from http://www.simitkvtk.com/

2. Expand binaries archive in a convenient directory.

3. Open MATLAB

4. Under the “File” menu, choose the “Set Path..” entry.

5. In the prompted dialog-box, add the previously extracted directory to MATLAB Path
At this point, SimITK is setup to work within Simulink. The user is free to browse the included Examples directory or open the SimITK Block Libraries to select and add desired blocks and construct SimITK workflows.
Appendix E

Developer Installation Documentation

E.1 Installation Prerequisites

- 5 GB at minimum for disk space to account for all source and build files
- 32-bit Visual Studio 9 2008 on Windows XP/Vista/7
- gcc
  - 3.6 on 32-bit/64-bit Linux/UNIX-based systems
  - 4.0 on 32-bit and 64-bit OS X systems

**NOTE:** The developer is encouraged to visit the MathWorks website\(^1\) to find the supported compiler for their version of MATLAB. The goal is to find a compiler that is both supported by the version of MATLAB that they have installed, and supported by ITK 3.20.

- CMake 2.8 (or later)
- MATLAB R2008b (or later)

\(^1\)http://www.mathworks.com/support/compilers/previous_releases.html
E.2 Installing and Configuring of Components

NOTE: It is assumed that the user is familiar with typical techniques used to build, compile, and install software from either the command line or GUI of choice. If unfamiliar with these concepts, additional documentation may need to be sought out.

E.2.1 Build, Compile, Install ITK 3.20 (or later)

ITK v3.20 is an absolute necessity as the WrapITK XML base files included with SimITK will be from this version of ITK. Other versions can be used, but it will require recompiling ITK with appropriate flags (outlined later in this appendix).

When configuring ITK, the following CMakeCache flags require setting:

- BUILD_DOCUMENTATION = OFF
- BUILD_EXAMPLES = OFF
- BUILD_SHARED_LIBS = ON
- BUILD_TESTING = OFF
- CMAKE_BUILD_TYPE = Release
- ITK_USE_OPTIMIZED_REGISTRATION_METHODS = ON
- ITK_USE_ORIENTED_IMAGE_DIRECTION = ON
- ITK_USE_PATENTED = OFF
- ITK_USE_REVIEW = ON
E.2.2 WrapITK (if not ITK v3.20)

Obtain WrapITK from http://code.google.com/p/wrapitk/ and add it to the base ITK source directory as outlined in the README. After CMake has been invoked to generate the build directory, the “USE_WRAP_ITK” flag will require enabling. At which point, extra flags will be made available where the user can set the desired datatypes to be wrapped.

At present, only the float, short, unsigned short, and unsigned char datatypes need to be enabled for wrapping with WrapITK.

E.2.3 Compiling, Building, Installing SimITK

- Download source archive from http://www.simitkvtk.com/

- Extract archive to desired directory.

- Run CMake and ensure no errors prompt from finding MATLAB, Simulink, and ITK

- Compile using Visual C++ (Windows) or make (UNIX).

Once the built SimITK directories have been added to the MATLAB Path as outlined in the previous appendix, SimITK is configured and ready for use.
Appendix F

SimITK File Examples and Templates

F.1 SimITK XML Examples

Listing F.1: Sample SimWrapITK Dumpfile Outlining Class Data Hierarchy

```plaintext
CannyEdgeDetectionImageFilter => HASH(0x1056e0ce0)
  typeDef_ID => .13
  Filter_Suffixes => ARRAY(0x1056e0ef0)
    IF3
    IF3
    Class_Data => HASH(0x105543088)
      bases => .148
        demangled => itk::CannyEdgeDetectionImageFilter<
          location => f1:92
          name => CannyEdgeDetectionImageFilter<
            file => f1
            artificial => 1
            size => 3200
            mangled => Z3itk29CannyEdgeDetectionImageFilter
              --INS_5ImageIfLj3EEES2 EE
                members => .132 .133 .134 .135 .136 .137 .138 .139 .140
                  -> .141 .142 .143 .144 .145 .146 .147 .149 .151 .152
                  -> .153 .155 .156 .157 .159 .161 .163 .165 .167 .169
                  -> .171 .173 .174 .175 .176 .178 .179 .180 .181 .182
                  -> .183 .185 .187 .188 .189 .190 .191 .192 .193 .194
                  -> .195 .196 .197 .198 .199 .200 .201 .202 .203 .204
                  -> .205 .206 .207 .208 .209 .210 .211 .212 .213 .214
                  -> .215 .216 .217 .218 .219 .220 .221 .222
                  context => .19
                    mangled => Z3itk
                    demangled => itk
                    name => itk
                    context => .2
                    tagType => Namespace
                    id => .19
                    tagType => Class
                    id => .12
                    line => 92
                    Class_ID => .12
```

Listing F.2: Sample SimWrapITK Dumpfile Outlining Method Data Hierarchy

| MEMBER | 200 ⇒ |
| inline ⇒ 1 |
| demangled ⇒ itk::CannyEdgeDetectionImageFilter< |
| −−−−−−−→ itk::Image<float, 3u>, |
| −−−−−−−→ itk::Image<float, 3u> |
| location ⇒ f1:198 |
| access ⇒ public |
| returns ⇒ ,16 |
| align ⇒ 8 |
| name ⇒ void |
| id ⇒ 16 |
| name ⇒ SetUpperThreshold |
| file ⇒ f1 |
| virtual ⇒ 1 |
| mangled ⇒ _ZN3itk29CannyEdgeDetectionImageFilter |
| −−−−−−−→ INS_5ImageIfLj3EEES2_E17SetUpperThresholdEf |
| tagType ⇒ Method |
| Argument_Data ⇒ HASH(0x10541ab48) |
| location ⇒ f1:198 |
| name ⇒ _arg |
| file ⇒ f1 |
| type ⇒ .44c |
| const ⇒ 1 |
| tagType ⇒ CvQualifiedType |
| id ⇒ .44c |
| type ⇒ .44 |
| align ⇒ 32 |
| name ⇒ float |
| tagType ⇒ FundamentalType |
| id ⇒ .44 |
| size ⇒ 32 |
| line ⇒ 198 |
| overrides ⇒ |
| context ⇒ ,12 |
| id ⇒ .200 |
| extern ⇒ 1 |
| line ⇒ 198 |
Listing F.3: SimWrapITK itkCannyEdgeDetectionImageFilter XML

```xml
<Filter_Descriptions>
  <Filter>
    <Name>CannyEdgeDetectionImageFilter</Name>
    <Template_Parameters>
      <Template_Parameter>TIInputImage</Template_Parameter>
      <Template_Parameter>TOOutputImage</Template_Parameter>
    </Template_Parameters>
    <Allowed_Datatypes>
      <Datatype>float</Datatype>
    </Allowed_Datatypes>
    <Allowed_Dimensionalities>
      <Dimensionality>2</Dimensionality>
    </Allowed_Dimensionalities>
    <Inputs>
      <Input>
        <Input_Name>Input</Input_Name>
        <Input_Type>ImageType</Input_Type>
        <Input_Dimension>ImageDimension</Input_Dimension>
      </Input>
    </Inputs>
    <Parameters>
      <Parameter>
        <Parameter_Name>AbortGenerateData</Parameter_Name>
        <Parameter_Type>bool</Parameter_Type>
        <Parameter_Size>1,1</Parameter_Size>
      </Parameter>
      <Parameter>
        <Parameter_Name>Debug</Parameter_Name>
        <Parameter_Type>bool</Parameter_Type>
        <Parameter_Size>1,1</Parameter_Size>
      </Parameter>
      <Parameter>
        <Parameter_Name>GlobalWarningDisplay</Parameter_Name>
        <Parameter_Type>bool</Parameter_Type>
        <Parameter_Size>1,1</Parameter_Size>
      </Parameter>
      <Parameter>
        <Parameter_Name>LowerThreshold</Parameter_Name>
        <Parameter_Type>ImageType</Parameter_Type>
        <Parameter_Size>1,1</Parameter_Size>
      </Parameter>
      <Parameter>
        <Parameter_Name>MaximumError</Parameter_Name>
        <Parameter_Type>double</Parameter_Type>
        <Parameter_Size>1,1</Parameter_Size>
      </Parameter>
      <Parameter>
        <Parameter_Name>MaximumError</Parameter_Name>
        <Parameter_Type>double</Parameter_Type>
        <Parameter_Size>1,1</Parameter_Size>
      </Parameter>
      <Parameter>
        <Parameter_Name>MetaDataDictionary</Parameter_Name>
        <ITK_Parameter_Type>MetaDataDictionary</ITK_Parameter_Type>
        <Parameter_Size>1,1</Parameter_Size>
      </Parameter>
      <Parameter>
        <Parameter_Name>NumberOfThreads</Parameter_Name>
        <Parameter_Type>int</Parameter_Type>
        <Parameter_Size>1,1</Parameter_Size>
      </Parameter>
      <Parameter>
        <Parameter_Name>OutsideValue</Parameter_Name>
      </Parameter>
    </Parameters>
  </Filter>
</Filter_Descriptions>
```
<Parameter_Type>ImageType</Parameter_Type>
<Parameter_Size>1,1</Parameter_Size>
</Parameter>
<Parameter_Name>Progress</Parameter_Name>
<Parameter_Type>float</Parameter_Type>
<Parameter_Size>1,1</Parameter_Size>
</Parameter>
<Parameter_Name>ReferenceCount</Parameter_Name>
<Parameter_Type>int</Parameter_Type>
<Parameter_Size>1,1</Parameter_Size>
</Parameter>
<Parameter_Name>ReleaseDataBeforeUpdateFlag</Parameter_Name>
<Parameter_Type>bool</Parameter_Type>
<Parameter_Size>1,1</Parameter_Size>
</Parameter>
<Parameter_Name>ReleaseDataFlag</Parameter_Name>
<Parameter_Type>bool</Parameter_Type>
<Parameter_Size>1,1</Parameter_Size>
</Parameter>
<Parameter_Name>Threshold</Parameter_Name>
<Parameter_Type>ImageType</Parameter_Type>
<Parameter_Size>1,1</Parameter_Size>
</Parameter>
<Parameter_Name>UpperThreshold</Parameter_Name>
<Parameter_Type>ImageType</Parameter_Type>
<Parameter_Size>1,1</Parameter_Size>
</Parameter>
<Parameter_Name>Variance</Parameter_Name>
<Parameter_Type>double</Parameter_Type>
<Parameter_Size>1,1</Parameter_Size>
</Parameter>
<Parameter_Name>Variance</Parameter_Name>
<Parameter_Type>double</Parameter_Type>
<Parameter_Name>ITK_Parameter_Type</Parameter_Name>
<ITK_Parameter_Type>FixedArrayType</ITK_Parameter_Type>
<Parameter_Size>1,ImageDimension</Parameter_Size>
</Parameter>
</Parameters>
<Outputs>
<Output>
<Output_Name>Output</Output_Name>
<Output_Type>ImageType</Output_Type>
<Output_Dimension>ImageDimension</Output_Dimension>
</Output>
</Outputs>
</Filter>
</Filter_Descriptions>
F.2 Filter Virtual Block .tpp Template

Listing F.4: SimITK Virtual Block Template

```cpp
#include "VirtualBlock.h"
#include "ImageConversion.tpp"

#ifndef INCHEADER_H
#define INCHEADER_H
#include "ITK_FILTER_NAME.h"
#endif

template <class InputPortType , class OutputPortType >
class SimITK@ITK_FILTER_NAME : public VirtualBlock < InputPortType , OutputPortType > {
  public:
    typedef itk::Image<typename InputPortType::PixelType ,
                   InputPortType::ImageDimension> InputImageType ;
    typedef itk::Image<typename OutputPortType::PixelType ,
                   OutputPortType::ImageDimension> OutputImageType ;

  SimITK@ITK_FILTER_NAME () {
    for ( unsigned int i=0 ; i<@NUM_INPUTS@ ; i++ ) {
      this->m_Inputs . push_back ( InputPortType () ) ;
    }
    for ( unsigned int i=0 ; i<@NUM_OUTPUTS@ ; i++ ) {
      this->m_Outputs . push_back ( OutputPortType () ) ;
    }
    for ( unsigned int i=0 ; i<@NUM_SPECIAL_INPUTS@ ; i++ ) {
      this->m_SpecialInputs . push_back ( VirtualSpecialPort () ) ;
    }
  }

  void Run () {
    Filter () ;
  }

  // mutators

  // accessors
```

private:

void Filter() {

  typename ITKFilterType::Pointer filter = ITKFilterType::New();

  // Creates the input image.

  // Sets the parameters of the ITK filter using methods from the original ITK class.
  // If necessary, parameter types are converted from Matlab to ITK

  // Get the input and output matrices

  // Translates the input matrix into an itk image.

  // Translate the filter’s output into an array.

  filter->Update();
}

};
F.3 Filter S-Function .cpp Template

Listing F.5: SimITK Simulink S-Function Template

```cpp
#include "mex.h"
#include "SimITK@FILTER_NAME@.tpp"

// include headers that will be used within the s-function
#include "SimStruct.h""'

/*------------------------------------------------*/

/* Function: mdlInitializeSizes */
/*------------------------------------------------*/

static void mdlInitializeSizes(SimStruct *S)
{
  SETUP_SPECIAL_DATATYPE

  // setup appropriate number of parameters
  ssSetNumSFcnParams(S, 2*NUM_PARAMETERS);
  if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
    {
    // Return if number of expected != number of actual parameters
    return;
  }
```
if (IMAGE_DIMENSIONALITY > 2) {
    ssAllowSignalsWithMoreThan2D(S);
}

// there are no continuous or discrete states to be used
ssSetNumContStates(S, 0);
ssSetNumDiscStates(S, 0);

// setup 2 input ports for every input image
// setup 1 input port for special type
@SETUP_INPUT_PORTS@
@SETUP_SPECIAL_INPUT_PORTS@
// set all input ports to be contiguous
@SETUP_INPUT_PORTS_CONTIGUOUS@

/*
 * Set direct feedthrough flag (1=yes, 0=no).
 * A port has direct feedthrough if the input is used in either
 * the mdlOutputs or mdlGetTimeOfNextVarHit functions.
 * See matlabroot/simulink/src/sfuntmpl_directfeed.txt.
 */

// both input ports are set to have direct feedthrough since they
// are used in mdlOutputs
@SET_INPUT_PORTS_DIRECT_FEEDTHROUGH@

@SETUP_OUTPUT_PORTS@

// setup 2 output ports for every output image
@SETUP_OUTPUT_PORTS@

// set a single sample time
ssSetNumSampleTimes(S, 1);
// no work vectors are necessary
ssSetNumRWork(S, 0);
ssSetNumIWork(S, 0);
ssSetNumPWork(S, 0);
ssSetNumModes(S, 0);
ssSetNumNonSampledZCs(S, 0);
ssSetOptions(S, 0);
}

/* Function: mdlInitializeSampleTimes */
@Abstract:
This function is used to specify the sample time(s) for your
S-function. You must register the same number of sample times as
specified in ssSetNumSampleTimes.
*/
static void mdlInitializeSampleTimes(SimStruct *S)
{
    // set sample time to be continuous
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

#define MDL_START /* Change to #undef to remove function */
#if defined(MDL_START)
/* Function: mdlStart */
@Abstract:
This function is called once at start of model execution. If you
have states that should be initialized once, this is the place
to do it.
*/
static void mdlStart(SimStruct *S){
    // Creates the filter
typedef SimTK::ITK::FILTER_NAME<
VirtualPort<INPUT_IMAGEPIXELTYPE,IMAGE_DIMENSIONALITY>,
VirtualPort<OUTPUT_IMAGEPIXELTYPE,IMAGE_DIMENSIONALITY>
F.3. FILTER S-FUNCTION . CPP TEMPLATE

```cpp
> SimITKFilterType;
SimITKFilterType* filter = new SimITKFilterType;
ssSetUserData(S, filter);
}
#endif /* MDL_START */
/* Function: mdlOutputs */
static void mdlOutputs(SimStruct* S, int T tid)
{
typedef SimITK@ITK_FILTER_NAME@<
    VirtualPort<INPUT_IMAGE_PIXELTYPE, IMAGE_DIMENSIONALITY>,
    VirtualPort<OUTPUT_IMAGE_PIXELTYPE, IMAGE_DIMENSIONALITY>>
    SimITKFilterType;
void* Pointer = ssGetUserData(S);
SimITKFilterType* filter = reinterpret_cast<SimITKFilterType*>(Pointer);
    // Gets the value from the parameters.
#ifdef MDL_SET_INPUT_PORT_DIMENSION_INFO
#define MDL_SET_INPUT_PORT_DIMENSION_INFO
#endif
    // Gets the signals from the input ports
// Set the filter block input.
    // Set the filter block output
    // gets the signals from the output ports
    // Set the filter block output
    // Will process the image.
    // the Run() method changes them
#endif
}

#define MDL_SET_INPUT_PORT_DIMENSION_INFO
#if defined(MDL_SET_INPUT_PORT_DIMENSION_INFO) && defined(MATLAB_MEX_FILE)
/* Function: mdlSetInputPortDimensionInfo */
static void mdlSetInputPortDimensionInfo(SimStruct* S, int_T port,
    const DimsInfo_T* dimsInfo)
```

/* In this function, you compute the outputs of your S-function block. Generally outputs are placed in the output vector, ssGetY(S). */
{  
    // dynamically set input port dimensions from the input signal at runtime  
    if(!ssSetInputPortDimensionInfo(S, port, dimsInfo)) return;

    /* Dynamically sets output data port dimensions at runtime.  
     * Output data ports are set to the same dimensions as the first input data 
     * port, port 1.  
     * Future work: Accommodate filters whose input and output port dimensions 
     * differ. */
    if(port == 1) {  
        int PortTotal = ssGetNumOutputPorts(S);
        for(int p=1; p<PortTotal; p+=2) {  
            // Note: Change only data ports
            if(!ssSetOutputPortDimensionInfo(S, p, dimsInfo)) return;  
        }
    }
}  

#define MDL_SET_INPUT_PORT_DIMENSION_INFO  
#if defined(MDL_SET_INPUT_PORT_DIMENSION_INFO) && defined(MATLAB_MEX_FILE)  
/* Function: mdlSetInputPortDimensionInfo */  
static void mdlSetInputPortDimensionInfo(SimStruct *S, int_T port, const DimInfo_T *dimsInfo)  
{  
}  
#endif  
*/  

#define MDL_SET_OUTPUT_PORT_DIMENSION_INFO  
#if defined(MDL_SET_OUTPUT_PORT_DIMENSION_INFO) && defined(MATLAB_MEX_FILE)  
/* Abstract:  
 * This method is called with the candidate dimensions for an output port  
 * with unknown dimensions. If the proposed dimensions are acceptable, the  
 * method should go ahead and set the actual port dimensions.  
 * If they are unacceptable an error should be generated via  
 * ssSetErrorStatus.  
 * Note that any other input or output ports whose dimensions are  
 * implicitly defined by virtue of knowing the dimensions of the given  
 * port can also have their dimensions set.  
 * See matlabroot/simulink/src/sfun_matadd.c for an example.  
 */  
static void mdlSetOutputPortDimensionInfo(SimStruct *S, int_T port, const DimInfo_T *dimsInfo)  
{  
}  
#endif  
*/  

#define MDL_SET_DEFAULT_PORT_DIMENSION_INFO /* Change to #define to add fcn */  
#if defined(MDL_SET_DEFAULT_PORT_DIMENSION_INFO) && defined(MATLAB_MEX_FILE)  
/* Abstract:  
 * This method is called when there is not enough information in your  
 * model to uniquely determine the port dimensionality of signals  
 * entering or leaving your block. When this occurs, Simulink's  
 * dimension propagation engine calls this method to ask you to set  
 * your S-functions default dimensions for any input and output ports  
 * that are dynamically sized.  
 * If you do not provide this method and you have dynamically sized ports  
 * where Simulink does not have enough information to propagate the  
 * dimensionality to your S-function, then Simulink will set these unknown  
 * ports to the 'block width' which is determined by examining any known  
 * ports. If there are no known ports, the width will be set to 1.  
 * See matlabroot/simulink/src/sfun_matadd.c for an example.  
 */  
static void mdlsetDefaultPortDimensionInfo(SimStruct *S)  
{  
}  
#endif  
*/  

/* Function: mdlTerminate */  
static void mdlTerminate (SimStruct *S)  
{  
}  
*/
static void mdlTerminate(SimStruct *S)
{
    typedef SimITK<FILTER_NAME<
            VirtualPort<INPUT_IMAGE_PIXELTYPE,IMAGE_DIMENSIONALITY>,
            VirtualPort<OUTPUT_IMAGE_PIXELTYPE,IMAGE_DIMENSIONALITY>>
        SimITKFilterType;

    void* Pointer = ssGetUserData(S);
    SimITKFilterType* filter = reinterpret_cast<SimITKFilterType*>(Pointer);
    delete filter;
}

/* Required S-function trailer */
*/
#endif  // end of extern "C" scope
F.4  Simulink Filter Mask .mdlpart Template

Listing F.6: SimITK Simulink Block Filter Mask Template

```
Block {
  BlockType               "S-Function"
  Name                    itk@FILTER_NAME@"'
  Ports                   @NUM_INPUT_PORTS@, @NUM_OUTPUT_PORTS@]
  Position                @LEFT@, @TOP@, @RIGHT@, @BOTTOM@]
  BackgroundColor         @BACKGROUND_COLOR@]
  LoadFcn                 "val = get_param(gcb,'MaskValues');
                          \nset_param(gcb,'MaskValues', val);
                          \nSimITK@FILTER_NAME@Callback('SetPortLabels', gcb);"
  FunctionName            "SimITK@FILTER_NAME@Mat"
  Parameters              @FILTER_PARAMETERS@]
  MaskPromptString        @MASK_PROMPT_STRING@]
  MaskStyleString         @MASK_STYLE_STRING@]
  MaskTunableValueString  @MASK_TUNABLE_VALUE_STRING@]
  MaskCallbackString      @MASK_CALLBACK_STRING@]
  MaskEnableString        @MASK_ENABLE_STRING@]
  MaskVisibilityString    @MASK_VISIBILITY_STRING@]
  MaskToolTipString       @MASK_TOOL_TIP_STRING@]
  MaskVarAliasString      @MASK_VAR_ALIAS_STRING@]
  MaskVariables           @MASK_VARIABLES@]
  MaskDisplay             @MASK_DISPLAY@]
  MaskSelfModifiable      on
  MaskIconFrame           on
  MaskIconOpaque          on
  MaskIconRotate          "none"
  MaskIconUnits           "autoscale"
  MaskValueString         @MASK_VALUE_STRING@]
  MaskTabNameString       @MASK_TAB_NAME_STRING@]
}
```
### F.5. SIMULINK LIBRARY .MDL TEMPLATE

#### Listing F.7: SimITK Simulink Library Template

```plaintext
Library {
    Name "SimITKLibrary@DATA_TYPE_ID@"
    Version 6.6
    MdlSubVersion 0
    SavedCharacterEncoding "windows−1252"
    LibraryType "BlockLibrary"
    SaveDefaultBlockParams on
    SampleTimeColors off
    LibraryLinkDisplay "none"
    WideLines off
    ShowLineDimensions off
    ShowPortDataTypes off
    IgnoreBidirectionalLines off
    ShowStorageClass off
    ShowTestPointIcons on
    ShowViewerIcons on
    SortedOrder off
    ExecutionContextIcon off
    ShowLinearizationAnnotations on
    ScopeRefreshTime 0.035000
    OverrideScopeRefreshTime on
    DisableAllScopes off
    BlockNameDataTip off
    BlockParametersDataTip off
    BlockDescriptionStringDataTip off
    Toolbar on
    StatusBar on
    BrowserShowLibraryLinks off
    BrowserLookUnderMasks off
    Created "@TIMESTAMP@"
    Creator "User"
    UpdateHistory "UpdateHistoryNever"
    ModifiedByFormat "%Auto@"
    LastModifiedBy "User"
    ModifiedDateFormat "%Auto@"
    LastModifiedDate "@TIMESTAMP@"
    ModelVersionFormat "1.%AutoIncrement:27>"
    ConfigurationManager "None"
    SimulationMode "normal"
    LinearizationMsg "none"
    Profile off
    ParamWorkspaceSource "MATLABWorkspace"
    AccelVerboseBuild off
    CovSaveName "covdata"
    CovMetricSettings "dw"
    CovNameIncrementing off
    CovHtmlReporting on
    covSaveCumulativeToWorkspaceVar on
    CovSaveSingleToWorkspaceVar on
    CovCumulativeReport off
    CovReportOnPause on
    ExtModeBatchMode off
    ExtModeEnableFloating on
    ExtModeTrigType "manual"
    ExtModeTrigMode "normal"
    ExtModeTrigPort "1"
    ExtModeTrigElement "any"
    ExtModeTrigDuration 1000
    ExtModeTrigDurationFloating "auto"
    ExtModeTrigHoldOff 0
    ExtModeTrigDelay 0
    ExtModeTrigDirection "rising"
}
```
F.5. SIMULINK LIBRARY .MDL TEMPLATE

ExtModeTrigLevel 0
ExtModeArchiveMode "off"
ExtModeAutoIncOneShot off
ExtModeIncDirWhenArm off
ExtModeAddSuffixToVar off
ExtModeWriteAllDataToWs off
ExtModeArmWhenConnect on
ExtModeSkipDownloadWhenConnect off
ExtModeLogAll on
ExtModeAutoUpdateStatusClock on
ProdHWDeviceType "32−bit Generic"
ShowModelReferenceBlockVersion off
ShowModelReferenceBlockIO off
BlockDefaults {
  Orientation "right"
  ForegroundColor "black"
  BackgroundColor "white"
  DropShadow off
  NamePlacement "normal"
  FontName "Arial"
  FontSize 10
  FontWeight "normal"
  FontAngle "normal"
  ShowName on
}
BlockParameterDefaults {
  Block {
    BlockType "S−Function"
    FunctionName "system"
    SFfunctionModules """"
    PortCounts " []"
  }
}
AnnotationDefaults {
  HorizontalAlignment "center"
  VerticalAlignment "middle"
  ForegroundColor "black"
  BackgroundColor "white"
  DropShadow off
  FontName "Arial"
  FontSize 10
  FontWeight "normal"
  FontAngle "normal"
  UseDisplayTextAsClickCallback off
}
LineDefaults {
  FontName "Arial"
  FontSize 9
  FontWeight "normal"
  FontAngle "normal"
}
System {
  Name "SimITKLibrary@DATA_TYPE_ID@"
  Location [5, 0, 570, 640]
  Open on
  ModelBrowserVisibility off
  ModelBrowserWidth 200
  ScreenColor "white"
  PaperOrientation "landscape"
  PaperPositionMode "auto"
  PaperType "usletter"
  PaperUnits "inches"
  TiledPaperMargins [0.500000, 0.500000, 0.500000, 0.500000]
  TiledPageScale 1
  ShowPageBoundaries off
  ZoomFactor "100"
  ReportName "simulink−default.rpt"
  Block {

F.5. SIMULINK LIBRARY .MDL TEMPLATE
MaskTabNameString   ","

@FILTER\_MASK\_CODE@
F.6 MATLAB Filter Callback .m Template

Listing F.8: SimITK MATLAB Callback Template

function @FILTER_NAME@Callback(action, block)
    feval(action, block)

% all callbacks work almost identically so first is commented to explain
% what is happening and rest follow same principle.
% At end of file is a function to output the current ports labelled
% properly, which is called after any indicator callback to ensure that the
% port labels are always up to date. Currently goes through entire MaskValues
% list and creates a new label scheme for the current set-up, as it seems
% impossible to change them individually, as you can never be sure where in
% list it should go without actually going through all the MaskValues to
% see its proper position based on the currently showing inputs.

@FUNCTION CALLBACK STRING@

@PORT LABEL STRING@