Unified Framework for Development, Deployment and Robust Testing of Neuroimaging Algorithms

**Alark Joshi · Dustin Scheinost · Hirohito Okuda ·
Dominique Belhachemi · Isabella Murphy ·
Lawrence H. Staib · Xenophon Papademetris**

Abstract Developing both graphical and command-line user interfaces for neuroimaging algorithms requires considerable effort. Neuroimaging algorithms can meet their potential only if they can be easily and frequently used by their intended users. Deployment of a large suite of such algorithms on multiple platforms requires consistency of user interface controls, consistent results across various platforms and thorough testing. We present the design and implementation of a novel object-oriented framework that allows for rapid development of complex image analysis algorithms with many reusable components and the ability to easily add graphical user interface controls. Our framework also allows for simplified yet robust nightly testing of the algorithms to ensure stability and cross platform interoperability. All of the functionality is encapsulated into a software object requiring no separate source code for user interfaces, testing or deployment. This formulation makes our framework ideal for developing novel, stable and easy-to-use algorithms for medical image analysis and computer assisted interventions. The framework has been both deployed at Yale and released for public use in the open source multi-platform image analysis software— BioImage Suite (bioimagesuite.org).

Keywords Neuroimaging software ·
Open source medical imaging software ·
Software development framework ·
Comprehensive software testing

A. Joshi (B) · D. Belhachemi · I. Murphy ·
L. H. Staib · X. Papademetris
Department of Diagnostic Radiology, Yale University,
300 Cedar Street, New Haven, CT 06520, USA
e-mail: alark1@gmail.com

D. Scheinost · L. H. Staib · X. Papademetris
Biomedical Engineering, Yale University, 300 Cedar Street,
New Haven, CT 06520, USA

H. Okuda
GE Healthcare, Tokyo, Japan

L. H. Staib
Electrical Engineering, Yale University, 300 Cedar Street,
New Haven, CT 06520, USA

Present Address:
A. Joshi
Boise State University, 1910 University Drive, MEC 302A,
Boise, ID 83725, USA

Introduction

Image analysis algorithms are typically developed to address a particular problem within a specific domain (functional MRI, cardiac, image-guided intervention planning and monitoring, etc.). Many of these algorithms are rapidly prototyped and developed without considerations for a graphical user interface (GUI), robust testing, and integration into a large cohesive software package. Sometimes these features are added later, but require considerable effort on the part of the developer of the original algorithm. This situation makes it difficult for deployment and widespread adoption of novel algorithms.

We present the design and development of a novel framework in our image analysis suite—BioImage Suite. In our experience, users may be divided into three categories: (i) full-time "naïve" users—those who interact with the software through its GUI, (ii) full-

time "expert" users—those who use a mixture of GUI and command-line tools for more efficient processing and *critically* (iii) part-time users—users who may use a combination of tools to accomplish their work, parts of which may be from this software package and parts from others. This last group is increasing in size with the availability of greater numbers of high quality tools.

A critical aspect of being able to support all three groups of users is the ability of the software to make available almost all of its functionality both as GUI applications *and* as command-line tools while maintaining consistency of performance between the two, i.e. a user should get exactly the same results performing a given task in either mode. The key issue is that as the software evolves and additional functionality is requested, often the changes get incorporated into either the GUI or the command-line version as needed without synchronizing the two. For example, the units of a smoothing kernel may be "voxels" in the GUI and "mm" in the command line interface. While the two invoke, deep down, the same algorithm, it is invoked in slightly different ways leading to different results and a major source of user frustration. While such synchronization is easy to fix for

1–2 modules, it is a non-trivial task when one considers more than 100 modules (Fig. 1).

To avoid this problem of integration/ resynchronization, we have designed and implemented a new object-oriented framework that enables us to integrate the command line and GUI of BioImage Suite— bioimagesuite.org within the same codebase. With this framework, the developer can focus on the creation of the algorithmic component and not worry about software engineering aspects needed for image analysis algorithms such as testing, integration, and creating customized workflows. The GUI is automatically generated by the algorithm object when the object is invoked. However, the developers may customize the GUI by overriding the appropriate methods. Testing is handled by the algorithm object simply by specifying the inputs, expected outputs, and the test flag. Additionally, in this new framework, it is possible to create data workflows where the output of an algorithm can then be used as the input of another algorithm. Thus, developers can reuse existing algorithms, saving time and reducing programming complexities. The simplified graphical user interface creation, robust testing and easy creation of
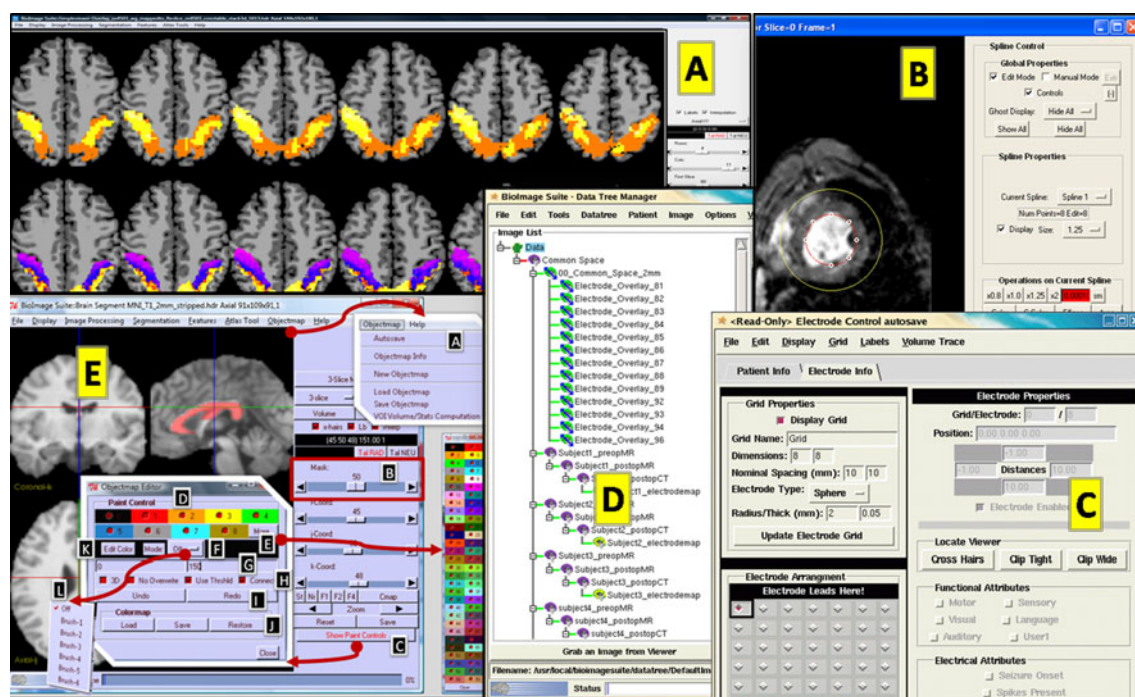
Fig. 1 Views of the BioImage Suite GUI. **a** The mosaic viewer which enables the display of multiple parallel images or multiple images in different rows (e.g. multiple fMRI tasks). **b** The 4D cardiac surface editor. **c** The intracranial electrode editor which is used to localize intracranial electrodes from CT/MRI images—this module will be enhanced by adding more automated local-

ization algorithms. **d** The datatree tool which enables the easy management of large studies and the flexible transformation and visualization of datasets. This tool serves as the underlying data model for the more complex applications. **e** The objectmap editor that enables interactive segmentation in all three orthogonal planes
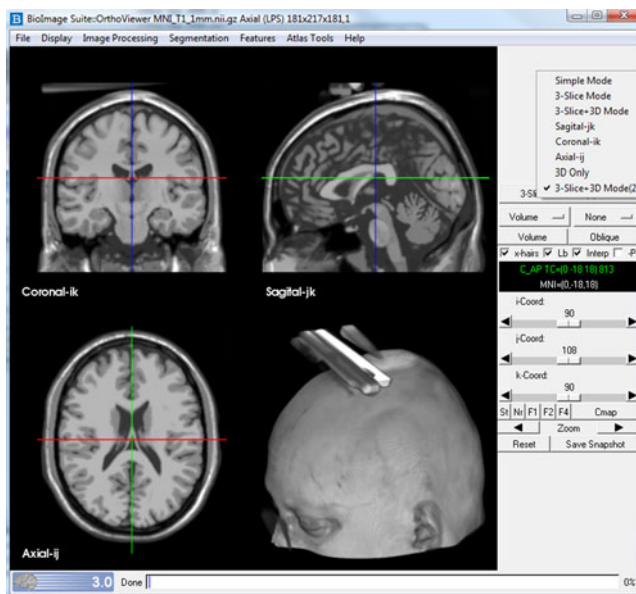
Fig. 2 Orthogonal viewer of BioImage Suite. Three views show 2D slice views of the MNI brain, whereas the *bottom right* view shows a volume rendered 3D representation of the data. The menu bar at the top allows a user to perform basic file operations (File tab), analyze images using the image processing tab, perform level set segmentation, histogram segmentation, math morphology etc. on the current image (Segmentation tab), add and perform operations on surfaces and landmarks (Features tab), use Atlas tools such as the WFU Atlas or the Yale Brodmann Atlas (Atlas Tools tab). The *drop down menu* shows various options in this viewer that allow a user to visualize a single 3D slice in a particular orientation (Axial, Coronal, Sagittal), 3D only or the configuration shown here (3-slice + 3D Mode (2)). More details regarding the viewers can be found in our manual online at www.bioimagesuite.org. Chapter 5 in the manual is dedicated to viewers

complex workflows are the technical innovations in the design of our new framework. The aim of this paper is to present the design of the framework and share our experience of developing and testing the framework with the neuroimaging community for further development of similar open source software (Fig. 2).

Additionally, this framework facilitates communication with other software tools. The parameter handling code, as defined, enables BioImage Suite components to output descriptions for both the Slicer Execution Model (Pieper et al. 2004) and the LONI Pipeline (Rex et al. 2003) at no extra work to the developer. This functionality is handled by the abstract parent class of the component hierarchy. See Fig. 10.

Related Work

In the field of medical image analysis, software engineering-based research has focused on describing a customized architecture for a unique setting. Coronato et al. (2006) developed an open-source architecture for immersive medical imaging that used 3D graphics and virtual reality libraries. Additionally, they also include ubiquitous computing principles for context-aware interaction with mobile devices. Shen et al. (2008) discuss their system which works with stereoscopic displays and uses projectors to provide an immersive experience in environments such as the CAVE.

Medium to large imaging software projects such as 3D Slicer (Pieper et al. 2004) have a component-based approach to developing software that allows for the easy development of user interfaces. 3D Slicer's modules have a thin interface and in theory the implementation of each module is completely unconstrained other than for the definition of the I/O in the XML structure. BioImage Suite modules derive from an object oriented architecture/base class and hence can also be independent mini-GUI applications (with full 3D viewer capabilities). The Slicer module provides more freedom/flexibility to the end-programmer whereas our approach provides richer functionality and the ability to write a custom application with the full viewing and interaction capabilities included. This design also ensures additional consistency, since each instance creates and manages its own user interface when included in a larger "managed" application.

Medical Imaging Interaction Toolkit (MITK) (Wolf et al. 2005) is a toolkit for medical image analysis that has some features similar to those in our framework. However, it is intended to be used as a toolkit and "is not intended as an application framework" (Wolf et al. 2005) that can be used for development of novel image analysis algorithms.

Table 1 shows a detailed comparison that facilitates the comparison of capabilities of medical image analysis workflow environments. The workflow environment most comparable to BioImage Suite is the LONI Pipeline. We can import the pipelines created in LONI pipeline to work with BioImage Suite and have worked with Slicer to make BioImage Suite functionality available as modules within 3D Slicer as well. Some of the other workflow environments provide the ability to create graphical user interfaces automatically, but they lack either robust cross platform testing and functionality, self contained installations, extensive documentation or are not being actively developed.

There are some key differences in our module approach to that of 3D Slicer which is the most similar package in many respects. The first lies in the structure of the modules themselves. Slicer modules are (in theory at least) independent programs whose sole

Table 1 Comparison of workflow environments for medical image analysis

	LONI pipeline	NiPype	CamBAFx	Fiswidgets	JIST	Slicer	BIS
Medical image analysis API	No	Nipy	CamBA	No	MIPAV	ITK/VTK	VTK/ITK
Self contained installation	Yes	No	Yes	No	No	Yes	Yes
Extensions	Binaries	N/A	N/A	Binaries	Plugins & Binaries	Plugins	Modules
Primary language	Java	Python	N/A	N/A	Java	C++	C++, Tcl
Distributed computing	Yes	Yes	No	No	Yes	Yes	No
GPU computing	Yes	No	No	No	No	N/A	Yes
Licensing	Research only	BSD	GPL v2	GPL	LGPL	BSD	GPL v2
Multi-platform	Yes	Yes	No	Yes	Yes	Yes	Yes
Testing	Yes	N/A	N/A	N/A	N/A	Yes	Yes
GUI framework	Vis prog	Desktop	Desktop	Desktop	Vis prog	Desktop	Desktop
Active devpt.	Yes	Yes	Yes	No	Yes	Yes	Yes

JIST—Java Image Science Toolkit (Lucas et al. 2010), *BIS*—BioImage Suite

interface with the main application is via the provision of an XML description. BioImage Suite modules, by contrast, are objects (deriving ultimately from a single parent class) which are accessed by the main application via the parent class API. The Slicer approach allows for greater flexibility in terms of the implementation of the modules, whereas our approach provides greater functionality to the module writer. For example, the module parent class in BioImage Suite implements functionality to output a Slicer-style XML description for the module which is automatically inherited with no modification by the more than 100 modules in BioImage Suite. Furthermore while Slicer modules are strictly commandline applications, BioImage Suite modules can be used either as (i) parts of a main application, (ii) as commandline executables and (iii) as standalone applets with their own graphical user interface and 3D viewer. The plumbing for all of this functionality is in the abstract parent class of the module hierarchy and is simply inherited by the derived modules.

A secondary difference is in the relation of the main application or module harness to the module. In the case of 3D Slicer the modules are queried by the main application for their description (in XML) and then the main application creates the user interface for the module. In BioImage Suite, by contrast, the modules are asked by the main application to create their own graphical user interfaces.

This key contribution of this paper lies in the description of how an established (over 10 years old) software package has evolved to meet the needs of its users with a focus on the underlying software architecture that enables us to provide this type of functionality. It is less an attempt to entice others to use this framework and more an attempt to present an overall strategy (a "design" pattern) that could be adapted in similar situations.

A Brief Overview of BioImage Suite

BioImage Suite (www.bioimagesuite.org) is an open source (GPL v2) multi-platform image analysis software that has found successful usage among users with a wide range of backgrounds and expertise. The key functionalities of BioImage Suite (Fig. 3) includes Anatomical image analysis (segmentation and registration tools), functional MRI analysis tools (single/multi subject analysis), Diffusion Weighted Image analysis (tensor analysis, fiber tracking), Neurosurgery tools (intracranial electrode localization, vvlink tools, differential SPECT tool), cardiovascular image analysis (4D surface editor, estimation of LV deformation, angiography tools).

BioImage Suite has been used in a number of peer-reviewed publications in high-impact scientific journals. These papers span both neuroimaging applications and applications in the rest of the body including diabetes, molecular imaging development, and tissue engineering. We would like to highlight two applications in particular to illustrate some of the unique aspects of our software.

Meltzer et al. (2008) used intracranial electro- encephalography to study working memory load on oscillatory power. These studies were performed on patients that had intracranial electrodes implanted as part of the workup for epilepsy neurosurgery. BioImage Suite was used for (i) intracranial electrode localization from CT brain images, (ii) mapping these CT images to anatomical MRI images of the same person and (iii) mapping these MRI images to a single brain template to enable the analysis of data from multiple patients in a common coordinate system.

Petersen et al. (2007) and Taksali et al. (2008) used (among other datasets) T1-weighted abdominal MRI images to quantify fat in the abdomen. BioImage Suite was used for slice inhomogeneity correction,

Key functionalities of BioImage Suite

Anatomical Image Analysis
- Interactive Segmentation Tools
- Tissue Classification (Histogram Segmentation, Bias Field Correction)
- Linear and Non-linear Registration
- Landmarks, Surfaces and Point-based Registration

Functional MRI Analysis
- Single Subject GLM
- Multi Subject Analysis / Multi Image Analysis
- Multimodal data management tool (Data Tree Manager)

Diffusion Weighted Image Analysis
- Diffusion Tensor Analysis
- Fiber Tracking

Neurosurgery Tools
- Intracranial Electrode localization
- VVLink Tool (in collaboration with BrainLAB AG)
- Differential SPECT tool

Cardiovascular Image Analysis
- 4D Surface Editor
- Estimation of LV Deformation
- Angiography Tools

Fig. 3 Key functionalities of BioImage Suite. This table lists all the functionalities in BioImage Suite that includes anatomical image analysis tools, functional MRI analysis tools, diffusion weighted image analysis tools, neurosurgery tools and cardiovascular image analysis tools. Specifics regarding the tools and their use can be found in our manual on our website at www. bioimagesuite.org

bias field correction, outline of the peritoneum (the bounding surfaces of the abdomen) and for interactive segmentation of fat layers using the object map editing tool.

System Design

Overview We present the design of our new framework that allows for easy development, deployment, and overall packaging of image analysis algorithms. Using this framework user interfaces and testing capabilities are created for the developer. Novel algorithms can be added (in C++) and custom workflow pipelines can be constructed (in Tcl) where each piece of the pipeline is an algorithm that takes an input and performs an operation. Figure 5 shows a flowchart for an image analysis algorithm. Each new algorithm takes a combination of images, surfaces, transformations, and input parameters and produces a combination of images, surfaces, and transformations as outputs.

An overview of the framework design is provided in Fig. 4. At the algorithm level of the architecture (Fig. 4c), novel algorithms are written in C++ and packaged as classes deriving from appropriate VTK-derived parent classes (Schroeder et sl. 2003). CUDA (NVIDIA 2007) and ITK (Ibanez and Schroeder 2003) functionality is hidden within these VTK-style C++ classes, which are wrapped and exported as Tcl functions. If a user does not have a modern CUDA capable graphics card, we seamlessly fall back to CPU versions of the same code at runtime instead of the CUDA accelerated code. Each algorithm is wrapped as an Incr-Tcl module using our new object-oriented framework (Fig. 4b). Incr Tcl http://incrtcl.sourceforge.net/itcl/ (Smith 2000) is an object-oriented extension to Tcl. These modules can function as standalone command-line applications, plugins/modules for other software and as GUI applets. Modules can also be incorporated into more complex applications for specific tasks (for e.g. surgical planning data tree). The application structure (Fig. 4a) consists of a combination of viewers, an appropriate data model and a set of module containers which manage the communication of the application with the modules. This framework enables us to create custom applications for different types of processing (e.g. our cardiac segmentation editor, or our fMRI Tool—Fig. 1) that share large aspects of their underlying codebase. Different data models are used in different applications to enable flexibility. For example, single-algorithm applications such as image analysis algorithms (registration, segmentation etc), bias field correction contain a fixed model with enumerated variables. Complex applications such as the surgical planning data tree contain a collection of pre-operatively
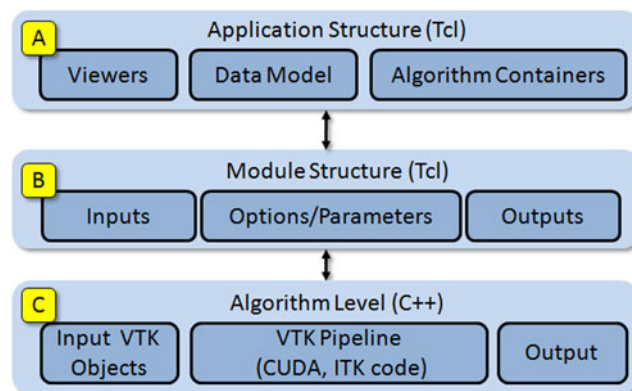
Fig. 4 BioImage Suite architecture. Main application structure **a**, module structure **b** and Algorithm level data processing (C++ Implementation) **c** all parts of A+B are written in Incr Tcl and invoke C++ code wrapped with a TCL access layer as pioneered by VTK
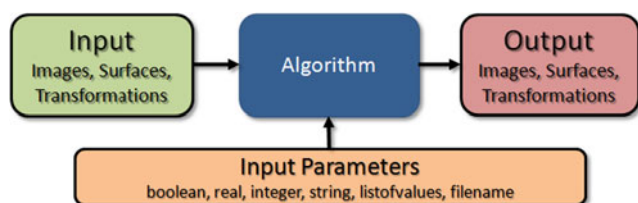
Fig. 5 This diagram provides an overview of the new unified framework for image analysis algorithm development. Any image analysis algorithm has a combination of images, surfaces, and transformations (from registrations) that serve as input to the algorithm. The algorithm has some input parameters (the values for which can be specified on the command line or using GUI components). In our framework, the input parameters can be one of *boolean, real, integer, string, listofvalues* (for drop down options when using a GUI) or a *filename*. The output can be a combination of images, surfaces and transformations

acquired images of varied modalities (CT, MR, DTI, fMRI, electrode mappings) as well as linear/non-linear transformations of each acquisition to the same space. In some cases, surfaces are required and can be added to the surgical planning data tree (Fig. 5).

Framework Design Details

Modules are implemented as derived classes of `bis_algorithm` (or a more specialized subclass such as `bis_imagetoimagealgorithm`.) There are two key methods that must be implemented: `Initialize` and `Execute`. In `Initialize`, the module interface is formally specified in terms of three sets: (i) inputs, which are objects such as images, surfaces, landmarks, etc., (ii) parameters, which are single values such as integers, strings, filenames and (iii) outputs, which are also objects like the inputs. In addition, developers can set other variables (e.g. category, and description) which are used in automatically generating the module documentation (Papademetris et al. 2009b).

Based on the definition of the input and output sets, the base abstract classes have functionality (which need not be touched by more concrete implementations) to (i) parse command line arguments if the algorithm class is invoked as an application; (ii) automatically create a GUI using the `CreateGUI` method (this method can be overridden by some algorithms to generate a more customized interface); and (iii) perform testing by parsing a test file. These classes can then be used (i) to invoke the algorithm (using an `Execute` method), (ii) to become a component of other algorithms (e.g. the image smoothing algorithm is invoked by the edge detection algorithm), (iii) to create standalone applications with an image viewer and a GUI, and (iv) to in-

tegrate individual components into a larger application (Fig. 6).

Core Classes

The new framework has at its core the following [Incr Tcl] classes:

1. `bis_option` encapsulates an option value (e.g. smoothness factor, etc.). An option can have a type of: listofvalues, boolean, real, integer, string or filename. Within this class there is functionality for creating an appropriate GUI for each option. For example, if the boolean option is specified a checkbox is created that allows the user to toggle the variable.
2. `bis_object` encapsulates the input and output objects of the algorithms. The core objects supported are: image, transform (both linear and non-linear), polygonal surface, landmark set and electrode grid.
3. `bis_basealgorithm` is the core algorithm class from which all algorithms are derived. It has all the functionality for manipulating options, inputs and outputs.
4. `bis_algorithm` is derived from `bis_basealgorithm` and adds the functionality needed for taking an algorithm and making it into a component or an executable. More specialized classes are derived from `bis_algorithm` such as `bis_imagetoimagealgorithm` which serves as a base for algorithms which take a single image as an input and produce a single image as an output.
5. `bis_guicontainer` is a derived class of `bis_algorithm` and serves as a parent class for creating multi-algorithm containers (e.g. a tabbed-notebook style GUI where each tab is a separate algorithm).

Application Architecture

Structure Applications in general have three parts (Fig. 4a). Part I is the data model which can either be the full-blown XML-based tree-like `bis_datamanager` or a custom array for storing data as needed. All applications in BioImage Suite can export their datamodels in the XML format used by `bis_datamanager` to improve flexibility and data reuse. Part II is a set of viewers and editors for visualizing as well as editing images and polygonal objects in a variety of ways. A core strength of BioImage Suite is that the viewers are designed to handle 4D (3D+t) data, a legacy of the origins of BioImage Suite as a

```
itcl::class bis_smoothimage {
    inherit bis_imagetoimagealgorithm
    constructor { } {    $this Initialize  }
    public method Initialize { }
    public method Execute { }
}

itcl::body bis_smoothimage::Initialize { } {
    #name,description,type,object,filename(if applicable),priority (optional)
    set inputs { { input_image "Input Image" pxitclimage   ""   0 } }
    set outputs { { output_image "Output Image"  pxitclimage   "" }}

    #commandswitch,description,shortdescription,optiontype,defaultvalue,valuerange,priority
    set options {
        { blursigma "kernel size [mm/voxel] of FWHM filter size" "Filter Size"  { real triplescale 100 }    2.0 { 0.0 20.0 }  0 }
        { unit      "kernel size unit mm or voxels " "Units" { listofvalues radiobuttons }  mm { mm voxels }  1}
        { radius    "radius factor of the gaussian in voxel units " "Filter Radius" real  1.5  { 0.0 5.0 }  -1 }
        { dimension  "2 or 3 to to do smoothing in 2D or 3D" "Dimensionality"  { listofvalues radiobuttons }   3 { 2  3 }  -999 }
        { testlog   "0: no test log , 1: output test log  " "testlog" boolean   0    { 0 1 }  -1000 }
    }

    #document
    set description "Smoothes an image with a specific gaussian kernel."
    set description2 "Smoothing kernel size blursigma (in mm by default ) represents the FWHM filter size."
    $this InitializeImageToImageAlgorithm
}
#  This checks if executable is called (in this case bis_smoothimage.tcl) if it is execute
# --------------------------------------------------------------------------------------
if { [ file rootname $argv0 ] == [ file rootname [ info script ] ] } {
    # this is essentially the main function
    set alg [bis_smoothimage [pxvtable::vnewobj]]
    $alg MainFunction
```

Fig. 6 Sample algorithm implementation. A new algorithm usually requires the implementation of two methods. The first is initialize (shown in detail in this figure) where the inputs, outputs and parameters are defined. The second is execute (not shown) which simply takes the specified inputs and parameters and runs the actual algorithm to generate the desired output. Derived classes can have customized graphical user interfaces by overriding the CreateGUI method. See Appendix B for a complete example on integrating an algorithm into our framework

cardiac image analysis package. Part III is the set of modules that are used to perform the task at hand in customized pipelines.

Invocation Modes

An algorithm can be invoked in three ways: (i) command line, (ii) GUI and (iii) "managed" graphical interface (see Fig. 8). The framework facilitates the invocation of the same code regardless of the manner in which the script is invoked. In Fig. 7, we can see an example of a non-linear registration script being invoked in three different ways. Labels A, A1 and A2 show a GUI with different components showing the input parameters. Label B in the figure shows a command line invocation which also provides Unix-style help to the users. Additionally, the same script can be contained in a managed container for a larger application (as shown by label D).

Using this framework the user can use the "Show Command" button embedded in the GUI (shown in Fig. 7c). The user can familiarize themselves with the algorithm at the GUI level. Then, the user can press this button and get a detailed command line specification for performing *exactly* the same task by invoking *exactly* the same code at the command line. This feature makes it easier for end-users to develop customized batch jobs/pipelines (Fig. 8).

In "standalone" mode, the module can either function as a command line only tool (depending on the setting of the dogui flag), or as an applet with its own GUI and viewer. There are four more global command-line settings as follows:

1. -xml,-logo: used to generate Slicer XML descriptions,
2. -loni: used to output LONI pipeline XML module descriptions,
3. -pset: used to query the SQLite-based parameter database for a complete parameter set, and
4. -ctest: used to run regression tests as part of nightly testing.

The modules in "managed" mode can also be invoked directly inside other modules. For example, the linear intensity registration module functions either as a standalone module, or is called by the nonlinear intensity registration module to estimate a registration to be used as an initial condition for the subsequent nonlinear registration. The fundamental idea behind all of this is ensuring consistent performance. Each
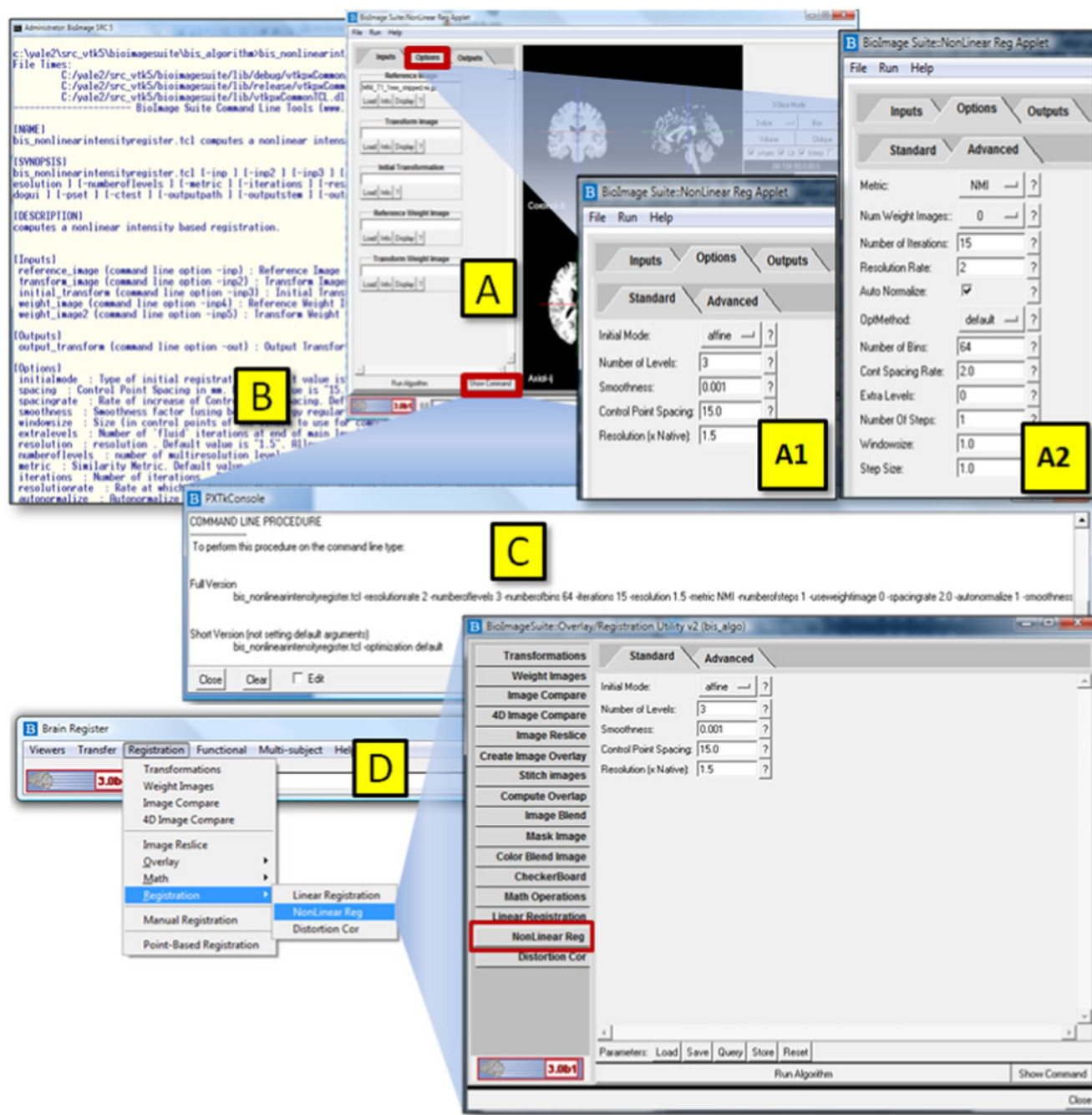
Fig. 7 This figure shows all the different ways in which a script can be invoked. Inset (A), (A1) and (A2) shows the graphical user interface (A) with the parameters in the standard tab (A1) and the advanced tab (A2). Additionally, the user can click on the "Show Command" button highlighted with a red rectangle that shows how the script can be invoked on the command line (Inset C). The script can also be invoked on the command line (Inset B) and in the situation where incorrect input parameters are provided, a Unix-style help is shown that shows the format for the input and input parameters. Additionally, the script can be contained in a managed framework (Inset D) where it becomes a menu item that invokes the same graphical user interface options as in Inset (A)

algorithm in BioImage Suite should be accessible in exactly one way and packaged in a proper module where the parameters are formally specified. Hence if there is an error, that module can be invoked directly using the exact same parameters that were used by another module or component to debug the issue. Even in managed mode, a module can generate (via the API call GetCommandLine) a detailed command-line
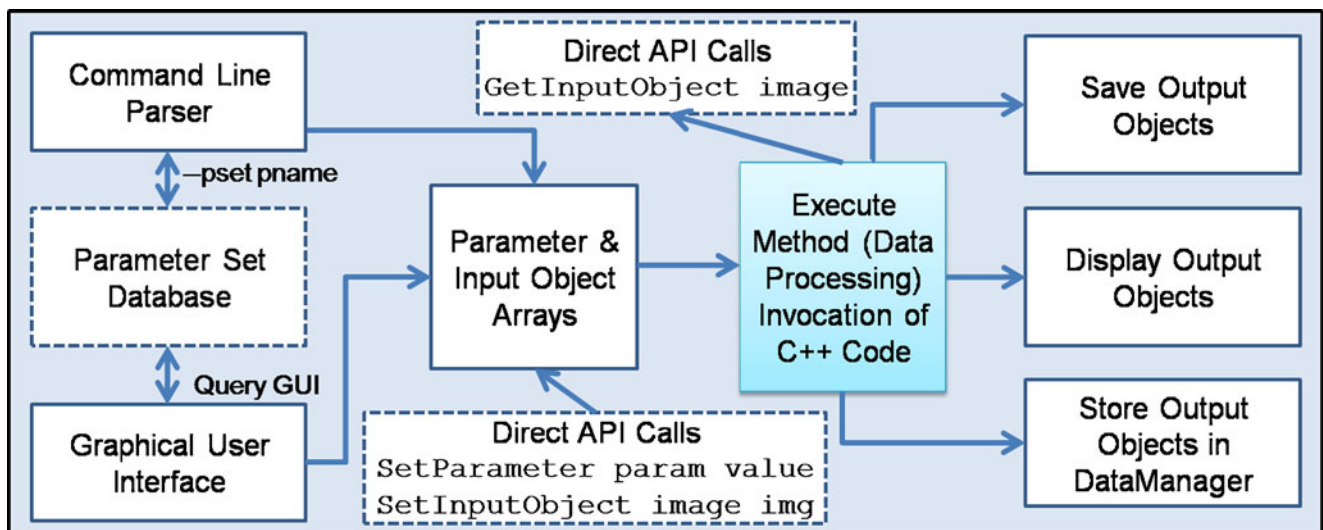
Fig. 8 BioImage Suite module architecture—detail view of Fig. 4b: the structure of a module. The inputs and parameters can be specified in three ways: (i) command line, (ii) GUI, (iii) direct application programming interface (API) calls. Then the exact same Execute method is invoked and the outputs stored/displayed as appropriate

specification that can be used to invoke this module directly.

Algorithm Implementation (see Fig. 4c)

Overview All core algorithms in BioImage Suite are implemented in C++. These often have as an access point a single C++ class which derives from a VTK parent class (e.g. vtkAlgorithm, vtkImageAlgorithm, etc.) that is wrapped to allow it to be accessed from Tcl code.

Dataflow Data and parameters are supplied to the algorithm from the module superstructure written in Incr Tcl (Fig. 4b). BioImage Suite modules exchange data in objects that are wrappers around VTK data sets. These contain both a pointer to a VTK-derived object (e.g. vtkImageData in the case of images), as well as auxiliary information such as filenames, colormaps, image headers, etc. There are five core objects that are used to represent (i) images, (ii) transformations (either linear or non-linear), (iii) surfaces, (iv) landmark sets and (v) electrode grids (landmark set can be tailored to the representation of implanted recording electrodes). All five of these are implemented as Incr Tcl classes and derive from a single parent class that specifies a common interface (e.g. Clear, Load, Save, Copy, ShallowCopy, Get/Set FileName, Get/SetVTKObject, etc.). The input data and input parameters obtained from the user interface (written in Tcl—Fig. 4b) are passed onto the underlying algorithms written as C++ classes. Within the C++ class, we have short VTK pipelines, possibly augmented with ITK-derived code or CUDA accelerated code. The outputs are made available as the outputs of the core C++ class and these are copied into the OutputArray objects (which use VTK data structures) of the module, prior to the deletion of the C++ object.

Base Functionality In addition to VTK and ITK, BioImage Suite C++ code leverages functionality from Lapack (Anderson et al. 1999) and the GDCM DICOM library http://www.creatis.univ-lyon1.fr/software/public/Gdcm/Main.html. A large amount of BioImage Suite specific core functionality (e.g. optimizers) is also available to speed up algorithm implementations.

Customized Workflow Example: Diff-SPECT Processing for Epilepsy

Using this framework, customized workflows can be created to enable the development of complex and streamlined algorithms. In these customized workflows, the output of one algorithm can be used as the input to another algorithm. Here we present an example of a customized workflow for Ictal-Interictal SPECT Analysis by SPM Half Normalized (ISASHN) algorithm (Scheinost et al. 2009) used to assist image-guided surgery research. First, two SPECT images are linearly registered to each other and then nonlinearly registered into MNI (Montreal Neurological Institute) space. The registered images are then masked, smoothed, and intensity normalized. A t-test is performed comparing these images to a healthy normal population. The resul-
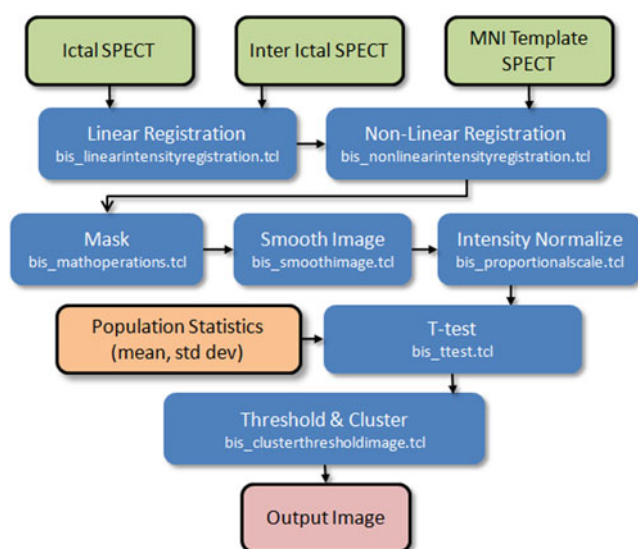
Fig. 9 Customized workflow using the unified BioImage Suite framework. Here the algorithm modules are depicted in blue (with the actual script name below it). In this workflow, the interictal and ictal SPECT are first linearly registered and output is then non-linearly registered with the MNI Template SPECT. The result of the registration is then processed using various algorithms (mask, smooth and intensity normalized). Then a t-test is performed with the mean and standard deviation from a control population. The output tmap is then thresholded and clustered to get the final output image

tant image, containing a t-value at each image location, is thresholded and clustered to produce the final output. This workflow can be implemented as a single algorithm object with its own GUI and testing protocol that sequentially calls other algorithm objects as presented in Fig. 9. The algorithm object can be instantiated from our BioImage Suite VVLink gadget to connect to the BrainLAB Vector Vision Cranial system for integration into neurosurgical research (Papademetris et al. 2009a). With the interoperability features that this new framework provides (Fig. 9), we can create complex workflows, such as the one presented here, using a graphical tool such as the LONI Pipeline (Rex et al. 2003).

Interoperability

This framework supports easy interoperability of BioImage Suite components with other software environments. For example, all command line tools (over 90 of them at this point) support the Slicer 3 execution interface by providing an XML description when invoked using the -xml flag (See Section "Invocation Modes"). This allows Slicer to scan the BioImage Suite binary directory and find all its components as plugins. Panel B in Fig. 10 particularly shows BioImage Suite

modules being recognized (red rectangle in Panel B) and available in Slicer. Similarly, we can recognize other command line tools that adhere to this interface and use them as plug-ins within some of the BioImage Suite GUI applications/applets.

In addition (via the use of the -loni 1 construct), BioImage Suite components output an XML description that is compatible with the LONI pipeline environment (Rex et al. 2003).

Testing and Verification

Regression and Unit Testing

Research software is continually evolving and adapting to meet the ever changing needs of its users. These needs put unique strains on the conventional software engineering process of first designing the software, then implementing it and finally thoroughly testing it prior to its release to the community at large. In practice, in research-related software, the design and implementation processes are always ongoing and the user inevitably gets a "cut" of the software that is frozen for only a short amount of time. This problem has led to the development of methodologies falling under the paradigm of "extreme programming" (Beck and Andres 2004) in which the components of the software are continuously tested via a set of test functions and the output of each test compared to the correct (manually generated) output for each component.

Our object-oriented framework enables an expanded testing setup and eliminates the need for individual testing of scripts for each module—*unit testing*. The process of creating tests boils down to the following straightforward *two-step procedure*: (i) creating *gold-standard* results and (ii) adding a line in the test-definition file.

The ease of this framework has enabled the addition of over 300 tests (in this newer format). We use the following test flags that allow robust testing of the algorithms.

– ctestexact: If enabled then comparison is done via subtraction; otherwise correlation is used. Allowed values are either 0 (Off) or 1 (On—default). This only applies to images and is useful in cases such as image registration where we can never warp the target image to look exactly like the reference.
– ctestthr: Threshold for subtraction or correlation. Default value is "0.001". Allowed range is from 0 to 100000. If ctestexact ==1 then this will cause the test to fail if $Max|I_1 - I_2| > thr$, where
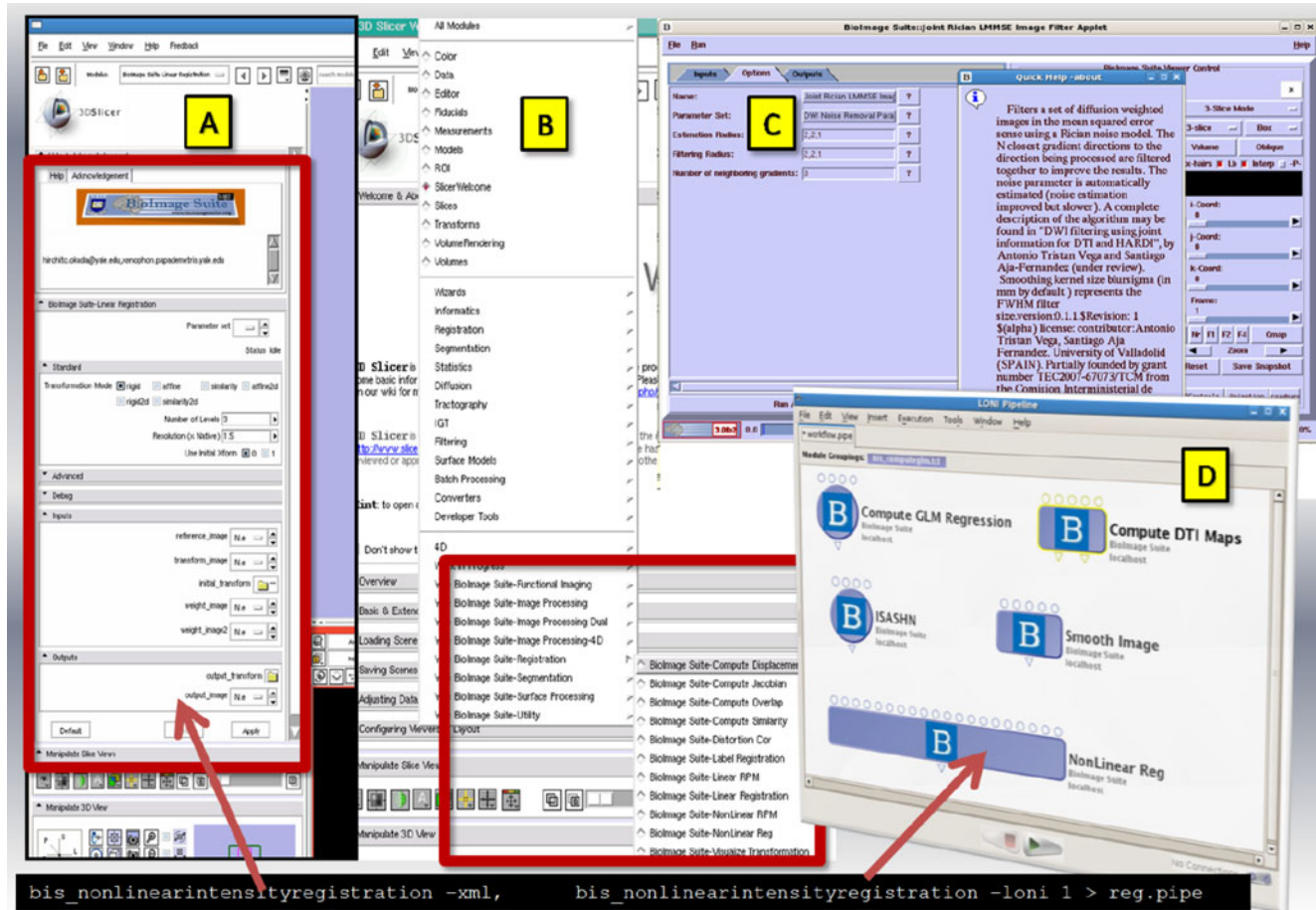
Fig. 10 The new framework facilitates interaction with 3D slicer. Panel **a** shows the autogenerated BioImageSuite user interface components in 3D Slicer. Panel **b** shows BioImageSuite modules being identified and loaded directly into slicer's user interface. Panel **c** shows a 3D slicer command line module recognized and loaded in BioImageSuite. Panel **d** shows a LONI pipeline form where inputs can be connected to various algorithm using a user interface

I_1, I_2 are image intensity values and *thr* is the threshold value. If ctestexact ==0 then failure will be declared if the output of the correlation value $r < thr$.

– `ctestsave`: If `ctestsave` is enabled, then the results of the computation are saved. Allowed values are either 0 (Off—default) or 1 (On). If the flag is set to 0 (Off), testing output files are not saved, i.e. the comparison is done (as always) in memory and then these objects are simply deleted. If the flag is set to 1 (On), the output is saved to automatically generated filenames.

– `ctesttmpdir`: This flag specifies the output directory to save output files generated during testing. The default value is " ". If the default value is given, then an automatically generated filename is used to save the file in the current directory; otherwise, the directory of the automatically generated filenames is changed to that specified by the `ctesttmpdir` flag.

– `ctestdebug`: If the name of a text file is specified for `ctestdebug` variable, then a textfile with that name is created with debug output. Default value is " ". If no name is specified for the text file, no log file is created.

All of these can be specified either on the command line e.g.

```
bis_castimage.tcl
--ctest testlist.txt \\
--ctestsave 1 --ctestdebug test2.log \\
--ctesttmpdir /tmp
```

or inside the testfile. This command takes the complete test description file (testlist.txt), extracts the relevant tests for the module at hand and runs them.

The regression testing is particularly valuable as we migrate BioImage Suite to newer versions of the base libraries (e.g. VTK and ITK). Regression testing can highlight for us which aspects of the software are

"broken" by migration and help with the upgrade. For example, our testing suggests that the current version runs just fine with Tcl 8.5/VTK 5.6—though we will still use Tcl 8.4.14/VTK 5.2 for the upcoming 3.0 release.

Nightly Testing

Nightly testing is done with the help of the functionalities in CDash (Martin and Hoffman 2009). The implementation of our testing framework allows for easy addition of test cases. For testing, we maintain a list of all the test cases which have the following format:

```
algorithm name : \\
input parameters and their values: \\
input files : \\
expected output file

bis_smoothimage: -blursigma 2.0: \\
MNI_T1_1mm.nii.gz: MNI_T1_1mm_sm2.nii.gz
```

When the nightly testing process starts, it goes through and tests each algorithm. Since in the new framework each algorithm is a unit of BioImage Suite, our nightly regression testing is similar to unit testing. For each algorithm, it looks up its name in the first column of the list, and if the name matches then it reads in the remaining arguments and performs the test. As shown above, to test the image smoothing algorithm we specify the name of the script, the input parameters and their values (blursigma = 2.0 in this case), the input file name and the expected output file name to compare the output with. The obtained output is compared with the expected output and based on the comparison a "test

passed" or "test failed" result is obtained. Therefore, adding more test cases is as simple as adding another line to the list of nightly tests for that algorithm.

Virtualization Virtual Machines (VM) allow for completely controlled, hardware-independent operating system environments with minimal software installation (e.g. Windows 2000, Visual Studio 2003.NET, Emacs, Subversion, base libraries) avoiding any unnecessary software that might interfere with development (especially on Windows). We use VWware Server (2005) to run multiple operating systems on a single machine. We have a number of VM's, including an Open Darwin virtual machine that is binary compatible with the x86 version of Mac OS X. This arrangement is used for testing on a multitude of Linux, Mac and Windows distributions which would be impractical on actual machines.

Dashboard Setup Figure 11 shows a screenshot of the nightly dashboard which shows the platforms that the scripts are tested on. As of now, the nightly tests run on Linux (CentOS), Windows (7, Vista and XP), Power Mac and Mac OS X. Figure 12 shows a screenshot of a list of some of the scripts as can be seen on the dashboard. This list allows us to readily know whenever a script fails on a particular platform.

Using our extensive virtual machine setup, we perform nightly testing on all three major platforms (Linux, Windows, Mac OS X) as well as variations within these (e.g. Windows XP/VISTA/7 with both Visual Studio 2008 and the mingwin gcc compiler, Linux

Fig. 11 In this figure, we can see the various platforms that the scripts are tested on—Linux, Windows, Mac OS X and their variations (e.g. Windows XP/VISTA/7 with both Visual Studio

2005/2008 and the mingwin gcc compiler, Linux 32-bit and 64-bit with various versions of gcc, Mac OS X Intel and PowerPC). This happens on a nightly basis and ensures multi-platform operability

Testing started on 2010-03-01 06:12:50

Site Name: node8 (MiniMac Intel OS X 10.4)
Build Name: Darwin-g++-4.0-i386-internal
Total time: 147:55:00
OS Name: Darwin
OS Platform: i386
OS Release: 8.11.1
OS Version: Darwin Kernel Version 8.11.1: Wed Oct 10 18:23:28 PDT 2007; root:xnu-792
Compiler Name: unknown
Compiler Version: unknown

304 tests passed.

Name	Status	Time	Details
C-bisVersionTest	Passed	3.00	
C-Test_vtkpxMatrix	Passed	3.00	
L005-bis_testviewer	Passed	17.00	
L006-bis_testviewer	Passed	15.00	
L007-bis_testviewer	Passed	22.00	
L008-bis_testviewer	Passed	12.00	
L009-bis_testviewer	Passed	12.00	
L010-bis_testviewer	Passed	12.00	
L014-bis_testdatabase	Passed	9.00	
L015-bis_testdatabase	Passed	8.00	

Fig. 12 This snapshot shows a sample list of scripts being tested and their status on a particular platform. As of now 304 total tests are being executed daily, as shown here

32-bit and 64-bit with various versions of gcc, Mac OS X intel and powerpc). Specifically nightly testing is performed on the following operating systems (all are 32-bit unless otherwise stated):

1. Windows 2000—using the MinGW gcc 3.4 compiler.
2. Windows XP—using Visual Studio 2008.
3. Windows VISTA—using Visual Studio 2008.
4. Mac OS X 10.4 powerpc—using gcc 4.0.
5. Mac OX X 10.4 intel—using gcc 4.0.
6. Mac OX X 10.5 intel—using gcc 4.0.
7. Linux (CentOS 4.3)—using gcc 3.2.
8. Linux (CentOS 5.4)—using gcc 3.4.
9. Linux 64-bit (CentOS 5.4)—using gcc 3.4.
10. Linux (Ubuntu 6.06 LTS)—using gcc 4.0.
11. Linux 64-bit (CentOS 5.4)—using gcc 4.1.
12. Linux 64-bit (KUbuntu 9.10)—using gcc 4.4.

Most of the above testing setups are run as virtual machines with the exceptions of Windows VISTA and the MacOS X test machines. We use a mixture of real and synthetic data with total testing time ranging from about 1 h on the newer machines to over 7 h on our G4 MacMini.

Testing the Binary Version A common problem with many software testing setups is the fact that the testing is done from the build directory. While this is useful, most users will not have a build directory, rather they will be using the binary version of the software which is the result of a 'make install' type process. As part of our testing setup we have the capability to run testing from this binary version of the software. Specifically, a CMakeLists.txt file is created as part of the installation and included with this in a 'testing' subdirectory which can subsequently be configured to run the tests. In this way we can verify that libraries and other scripts do not rely on the existence of the build directory to satisfy dependencies, something that would be problematic in real world use where the build directory does not exist.

Other Issues The use of multiple operating systems and compilers enables improved quality assurance as, for example, questionable code constructs which do not result in problems in one compiler might cause failure using another. When a test passes on all compilers/operating systems it suggests that the code is reasonably clean.

In this multi-platform testing a key problem is roundoff errors especially with respect to resampling images. A voxel intensity value could be 1.0000001 in platform A, whereas it ends up as 0.999999 in platform B which when truncated to an integer results in a difference of 1. While these events are rare, given an image of tens of millions of voxels (e.g. $256 \times 256 \times 256$ voxels) such round off errors need only have a probability of one in a million before they are practically guaranteed to occur in an image. To overcome such issues, the regression testing often will compare images by computing the correlation between them (with a high threshold e.g. 0.999) as opposed to simply subtracting them and looking for the maximum value—this is controlled by the setting of the `ctestexact` and `ctestthr` parameters.

Another related issue is roundoff errors in iterative processes such as non-linear registration which only become an issue after running multiple iterations. These only become apparent when the testing is run with real data (e.g. large images) and realistic numbers of iterations. This is why we use full size images so as to be able to discover errors of this type; small images and low iteration counts never quite reveal problems of this nature.

Discussion & Conclusions

A core strength of this framework is that most developers can simply work at the lower two levels (Panels B and C of Fig. 4) to produce new modules which are also easily accessible as standalone applications (both GUI and command-line). These modules can be tested and used separately and then, if needed, migrated to "larger" BioImage Suite applications later. A second advantage of this setup is that users can learn how to invoke these modules at the Graphical User Interface level and then get a detailed command-line specification using the "Show Command Line" option embedded in the GUI. This command will perform *exactly* the same task by invoking *exactly* the same code; customized batch jobs/pipelines are easily created this way.

The strengths of BioImage Suite's new framework are that it facilitates easy development and encapsulation of image analysis algorithms which allows the developer to focus on the development of the algorithm. It allows for easy creation of user interfaces and robust testing of the algorithms on multiple platforms. Additionally, customized workflow pipelines have been created by developers to allow for the creation of complex algorithms.

Information Sharing Statement

The software described in this paper can be freely downloaded at http://www.bioimagesuite.org. We would greatly appreciate any suggestions, comments or feedback to improve the architecture and future releases of BioImageSuite.

A Obtaining and Compiling the Source Code

Since we cater to a wide variety of users, we provide binary distributions as well as source code for expert users. We provide binaries for Windows, Linux and Macs that can be easily installed. On Microsoft Windows, we provide a self-contained installer file that completely automates the procedure. On UNIX derivatives (this includes Linux and the Apple Macintosh Operating System) the procedure involves uncompressing one file in the /usr/local directory or another location of your choice.

A.1 Compiling the Source Code

To compile the source, we use CMake to configure the user environment which then creates appropriate project files (Solutions for Visual Studio, Makefiles for Unix derivatives). To ensure complete compatibility, we ship BioImage Suite with the exact versions of all the software it depends on. This avoids compatibility issues and reduces installation overheads for the user. BioImage Suite includes compatible versions of the Visualization Toolkit (VTK), Insight Toolkit (ITK), Cross Platform Make (CMake), CLAPACK, MySQL, TCL and Grassroots DICOM (GDCM). In the CMake process, these libraries need to be accurately found to ensure an compatibility issues. Once the project files are created, compiling and source code is similar to compiling VTK, ITK and other software that uses CMake. More details including step-by-step instructions and screenshots can be found in *Chapter 3—Starting and Running BioImage Suite* of our 343-page manual available in HTML and PDF format on our website—www.bioimagesuite.org.

B Integrating an Algorithm into the Framework

In order to use the framework, the actual algorithm has to be written as a C++ class similar to any VTK class. Once that class is compiled with Tcl wrapping on, it is ready to be tested. Here we look at an example where we create a application that creates a module for the Gaussian Smoothing class in VTK (*vtkImageGaussianSmooth*).

B.1 Initial Setup

```
#!/bin/sh
# the next line restarts using wish \
    exec vtk "$0" "$@"

lappend auto_path [ file dirname
                    [ info script ]]
lappend auto_path [file join [file join
                    [ file dirname \
[ info script ]] ".." ] base]
lappend auto_path [file join [file join
                    [ file dirname \
[ info script ]] ".." ] apps]
```

B.2 Class Definition

Each class needs at least three methods (in addition to the constructor). The *Initialize* method is used to define the lists of inputs, outputs and options. This ends by

calling the initialize method of its parent class which will append to these lists and then go on to initialize everything. The *GetGUIName* method simply gives the "English" name for the class. The *Execute* method is where the actual execution happens and where the algorithm methods are invoked.

```
package provide bis_smoothimage 1.0
package require
bis_imagetoimagealgorithm 1.0

itcl::class bis_smoothimage {

    inherit bis_imagetoimagealgorithm

    constructor { }
    {      $this Initialize  }

    public method Initialize { }
    public method Execute { }
    public method GetGUIName    { }
        { return "Smooth Image" }
}
```

B.3 The *Initialize* Method

```
itcl::body bis_smoothimage::Initialize { } {

#commandswitch, description,
    shortdescription, \
    optiontype, defaultvalue, valuerange,
    priority
 set options {

{ blursigma "kernel size [mm/voxel] of FWHM
  filter size"
 "Filter Size"  { real triplescale 100 }
 2.0 { 0.0 20.0 }
 0 }

{ unit       "kernel size unit mm or voxels "
  "Units"
    { listofvalues radiobuttons }  mm
    { mm voxels }   1}

{ radius     "radius factor of the gaussian
  in voxels"
"Filter Radius" real  1.5  { 0.0 5.0 }
 -1 }

{ dimension  "2 or 3 to to do smoothing
  in 2D or 3D"
"Dimensionality" { listofvalues
 radiobuttons } 3
 { 2  3 }  -999 }

    }
    set defaultsuffix { "_sm" }
    set scriptname bis_smoothimage
```

```
set completionstatus "Done"
#
# Documentation
#
set description "Smoothes an image
with a \ specific gaussian kernel."

    $this InitializeImageToImageAlgorithm
}
```

B.4 The *Execute* Method

The *Execute* method contains the code that is executed when the user executes the algorithm. The input data as well as the input parameters are obtained and the algorithm is executed. The output image or surface, depending on the algorithm being tested, is saved as per the user's preference.

```
itcl::body bis_smoothimage::Execute {  } {
```

Part 1 — get the parameters and inputs

```
    set blursigma   [ $OptionsArray
                      (blursigma) GetValue ]
    set unit        [ $OptionsArray
                      (unit)      GetValue ]
    set radius      [ $OptionsArray
                      (radius)    GetValue ]
    set dimension   [ $OptionsArray
                      (dimension)
                      GetValue ]
    }
```

Next get the actual input image. This is of type *pxitclimage* (a wrapper around vtkImageData). For surfaces, we have *pxitclsurface*, which is a wrapped around vtkPolyData.

```
    set image_in    [ $this GetInput ]
# To get the spacing first we need a pointer
  to the
# encapsulated vtkImageData obtained using
  the
# GetImage method of pxitclimage
    set spacing [[ $image_in GetImage ]
    GetSpacing ]
# Compute proper smoothness kernels if unit
  is voxels or mm
    if { $unit == "voxels"} {
    for { set j 0 } { $j <=2 } { incr j } {
        set sigma($j) [ expr $blursigma *
        0.4247 / [ lindex \
        $spacing $j ]]
    }
    } else {
    for { set j 0 } { $j <=2 } { incr j } {
        set sigma($j) [ expr $blursigma *
        0.4247 ]
    }
    }
    set radiusz $radius
    if { $dimension == 2 } {
```

```
    set radiusz 0
    set sigma(2) 0.0
}
```

This is the actual VTK pipeline code, where *vtkImage-GaussianSmooth* is invoked with the user specified input image and input parameters.

```
# Actual vtk code
    set smooth  [ vtkImageGaussianSmooth
    [ pxvtable::vnewobj ]  ]
    $smooth SetStandardDeviations $sigma(0)
    $sigma(1) $sigma(2)
    $smooth SetRadiusFactors $radius $radius
    $radiusz
    $smooth SetInput [ $image_in GetObject ]
    $this SetFilterCallbacks $smooth
    "Smoothing Image"
    $smooth Update
```

Next we store the output resulting from the execution of the algorithm.

```
# When done store the output in the Output
  Object
    set outimage [ $OutputsArray
    (output_image) GetObject ]
    $outimage ShallowCopyImage
    [ $smooth GetOutput ]
    $outimage CopyImageHeader
    [ $image_in GetImageHeader ]
# Add a comment to the image header
  (if NIFTI!)
    set comment [ format " [ $this
    GetCommandLine full ]" ]
    [ $outimage GetImageHeader ] AddComment
    "$comment $Log" 0
# Clean up
    $smooth Delete
    return 1
}
```

This checks if executable is called (in this case bis_smoothimage.tcl) if it is, then execute

```
if { [ file rootname $argv0 ] ==
[ file rootname \ [ info script ] ] } {
    # this is essentially the main function
    set alg
    [bis_smoothimage [pxvtable::vnewobj]]
    $alg MainFunction
}
```

References

Anderson, E., et al. (1999). *Lapack user's guide*. SIAM.

Beck, K., & Andres, C. (2004). *Extreme programming explained: Embrace change* (2nd ed.). Addison-Wesley Professional.

Coronato, A., De Pietro, G, & Marra, I. (2006). An open-source software architecture for immersive medical imaging. In *Proceedings of the IEEE international conference on virtual environments, HCI and measurement systems*.

Ibanez, L., & Schroeder, W. (2003). *The ITK software guide: The insight segmentation and registration toolkit*. Kitware, Inc., Albany, NY. www.itk.org.

Lucas, B.C., Bogovic, J. A., Carass, A., Bazin, P.-L., Prince, J. L., Pham, D. L., et al. (2010). The java image science toolkit (jist) for rapid prototyping and publishing of neuroimaging software. *Neuroinformatics, 8*, 5–17.

Martin, K., & Hoffman, B. (2009). *Mastering CMake*. Kitware, Inc.

Meltzer, J. A., Zaveri, H. P., Goncharova, I. I., Distasio, M. M., Papademetris, X., Spencer, S. S., et al. (2008). Effects of working memory load on oscillatory power in human intracranial EEG. *Cerebral Cortex, 18*, 1843–1855.

NVIDIA (2007). *NVIDIA compute unified device architecture (CUDA)*.

Papademetris, X., DeLorenzo, C., Flossmann, S., Neff, M., Vives, K., Spencer, D., et al. (2009a). From medical image computing to computer-aided intervention: Development of a research interface for image-guided navigation. In *International journal of medical robotics and computer assisted surgery* (Vol. 5, pp. 147–157).

Papademetris, X., Jackowski, M., Joshi, A., Scheinost, D., Murphy, I., Constable, R. T., et al. (2009b). *The BioImage suite module description manual*. A manual for the BioImage Suite project.

Petersen, K. F., Dufour, S., Savage, D. B., Bilz, S., Solomon, G., Yonemitsu, S., et al. (2007). The role of skeletal muscle insulin resistance in the pathogenesis of the metabolic syndrome. *Proceedings of the National Academy of Sciences of the United States of America, 104*, 12587–12594.

Pieper, S., Halle, M., & Kikinis, R. (2004). 3D slicer. *IEEE international symposium on biomedical imaging ISBI 2004*.

Scheinost, D., Blumenfeld, H., & Papademetris, X. (2009). An improved unbiased method for diffspect quantification in epilepsy. *IEEE international symposium on biomedical imaging ISBI 2009*.

Schroeder, W., Martin, K., & Lorensen, B. (2003). *The visualization toolkit: An object-oriented approach to 3D Graphics*. Kitware, Inc., Albany, NY. www.vtk.org.

Shen, R., Boulanger, P., & Noga, M. (2008). Medvis: A real-time immersive visualization environment for the exploration of medical volumetric data. In *Proceedings of the fifth international conference on biomedical visualization* (pp. 63–68).

Smith, C. (2000). *[Incr-tcl/tk] from the ground up*. McGraw-Hill.

Taksali, S. E., Caprio, S., Dziura, J., Dufour, S., Cali, A. M., Goodman, T. R., et al. (2008). High visceral and low abdominal subcutaneous fat stores in the obese adolescent: A determinant of an adverse metabolic phenotype. *Diabetes, 57*, 367–371.

Rex, D. E., Ma, J. Q., & Toga, A. W. (2003). The LONI pipeline processing environment. *NeuroImage, 19*(3), 1033–1048.

VWware Server (2005) http://www.vmware.com/products/server/.

Wolf, I., Vetter, M., Wegner, I., Bottger, T., Nolden, M., Schobinger, M., et al. (2005). The medical imaging interaction toolkit. In *Medical image analysis* (pp. 594–604).