Next Generation of the Java Image Science Toolkit (JIST): Visualization and Validation

Release 1.0

Bo Li[1], Frederick Bryan[2], Bennett A. Landman[1,2]

August 15, 2012

[1]Computer Science, Vanderbilt University, Nashville, TN, USA 37235

[2]Electrical Engineering, Vanderbilt University, Nashville, TN, USA 37235

Abstract

Modern medical imaging analyses often involve the concatenation of multiple steps, and neuroimaging analysis is no exception. The Java Image Science Toolkit (JIST) has provided a framework for both end users and engineers to synthesize processing modules into tailored, automatic multi-step processing pipelines ("layouts") and rapid prototyping of module development. Since its release, JIST has facilitated substantial neuroimaging research and fulfilled much of its intended goal. However, key weaknesses must be addressed for JIST to more fully realize its potential and become accessible to an even broader community base. Herein, we identify three core challenges facing traditional JIST (JIST-I) and present the solutions in the next generation JIST (JIST-II). First, in response to community demand, we have introduced seamless data visualization; users can now click 'show this data' through the program interfaces and avoid the need to locating files on the disk. Second, as JIST is an open-source community effort by-design; any developer may add modules to the distribution and extend existing functionality *for release*. However, the large number of developers and different use cases introduced instability into the overall JIST-I framework, causing users to freeze on different, incompatible versions of JIST-I, and the JIST community began to fracture. JIST-II addresses the problem of compilation instability by performing continuous integration checks nightly to ensure community implemented changes do not negatively impact overall JIST-II functionality. Third, JIST-II allows developers and users to ensure that functionality is preserved by running functionality checks nightly using the continuous integration framework. With JIST-II, users can submit layout test cases and quality control criteria through a new GUI. These test cases capture all runtime parameters and help to ensure that the module produces results within tolerance, despite changes in the underlying architecture. These three "next generation" improvements increase the fidelity of the JIST framework and enhance utility by allowing researchers to more seamlessly and robustly build, manage, and understand medical image analysis processing pipelines.

Keywords: JIST next generation, medical image analysis, visualization, fidelity, continuous integration

Contents

1. Introduction

The Java Image Science Toolkit (JIST) framework is the cumulative result of over 15 years of image science research with MRI data [1-4], and is based on the National Institutes of Health Center for Information Technology (NIH/CIT) Medical Image Processing And Visualization (MIPAV) program [5]. It has been a part of the Neuroimaging Informatics Tools and Resources Clearinghouse (NITRC) since September 2008. The JIST model's success is evidence by over 7,940 downloads as of August 2012, its active developer and user community, and its consistent listing in the most active NITRC projects.

JIST was developed as an open-source multi-platform tool specifically to facilitate prototyping, sharing, and easy usage of community created image processing tools [1-4]. JIST is based on the Java programming language (Sun Microsystems, Santa Clara, CA) and able to run on almost any platform without recompilation [6, 7]. In the JIST software engineering model, designers of medical image processing tools submit their new developments as individual modules which end-users can then readily access through the use of the JIST library. JIST enables users to piece multiple modules from different developers together into a single pipeline ("layout") while ensuring compatibility between modules and providing for reasonable behavior when a module fails. Hence, JIST provides a mechanism for the end-user to push massive numbers of datasets through tailor designed pipelines using either multi-core or grid architectures. JIST facilitates cross community sharing, supports new pipeline development, promotes best-practices, and provides a straightforward path for the medical image community to utilize cluster based parallel processing. The basic structure and organization of Jist is presented in [1].

Although the initial release of JIST ("JIST-I") has been highly successful, there were important gaps to be filled to fulfill its potential and become more accessible to medical image community members. In this manuscript, we discuss these challenges and present solutions to the core gaps in JIST functionality. Specifically, in the next generation of JIST ("JIST-II"), we incorporate:

(1) Improved data monitoring by implementing an interactive visualization capability for module input/output data that can be viewed in real-time as data is processed.

- There has been a specific demand to improve the user's ability to manage/understand the progression of data as it moves through modules within a layout. In response to this

demand, we consider the key issue of data visualization. JIST is integrated with MIPAV's extensive visualization tools, which includes both slice-based and renderer-based display methods. It stores output and input data from each module within a pipeline and organizes them into a systematically named tree hierarchy of directories. In the initial JIST release ("JIST-I"), the output/input data could be viewed but required that user first find the correct files on disk. Because JIST is designed to enable complex layouts and facilitate the processing of large numbers of image datasets, finding the correct file could be tedious and hampered the user from fluid troubleshooting or the understanding of a layout. Improving upon this visualization experience by making input/output data visualization streamlined with processing would address user demand for improved ability to track data progression as the data moves through a layout.

(2) Nightly building of the JIST framework with Hudson on large clusters to ensure proper JIST functionality with the incorporation of new or updated modules. To keep the advantages of open-source software while ensuring the validation of the framework, we have concentrated on developing automatic continuous integration for JIST [8, 9].

- Broader adoption of JIST is hindered by the difficulties in maintaining stable modules, pipelines, and utility source code in a shared source base for multiple developers, diverse development environments, and different use cases. Open-source challenges are not unique to JIST and debate over whether open-source software development leads to more or less secure software has raged for years. So, although open source may be less secure and lead to stability problems, the advantages that users can build their own image modules, openly dialogue about technical problems, and freely reveal their innovation make the framework more flexible, adaptable, and useful. Two key points of entry can be particularly destabilizing to JIST functionality. First, a developer may update or add a module which causes the JIST framework to crash when the module is included in the JIST library. Second, even when the JIST framework remains valid, layouts designed by users may no longer be compatible with updated modules. As JIST is intended to thrive on community contributions and new modules, clearly JIST needs a method that keeps JIST open-source yet also ensures the stability of the JIST framework and user created pipelines. Such a method would result in less troubleshooting and would help draw a broader community to JIST and the user-developed modules.

(3) Provide nightly testing for each output of every module in layouts through addition of the test case functionality – test cases file tested on the Hudson server, which are generated in the JIST-II Generate Test Case GUI by end-users.

- A basic concern of JIST end-users has been how to guarantee their designed pipelines would remain working, valid, and up to date when their efforts rely on common libraries, which may be altered by the larger community. A method that allowed end user to test existing, working modules on current releases of JIST testing would allow the discovery of bugs as early as possible, reduce frustration of end users, and encourage use of the most recent JIST release.

By addressing these specific concerns of JIST-I users, JIST-II has become more accessible to the broader neuroimaging community. In turn, JIST-II now provides an ever-more-useful infrastructure to support significant contributions in medical imaging research.

This manuscript is organized as follows. In Section 2, we present an overview and guide to the improved visualization features of JIST-II. In Section 3, we discuss both the user-visible and backend aspects of the nightly build testing feature of JIST-II. Section 4 describes the methodology for users generating Test Case files for module validation and for managers adding those test jobs into Hudson server with simple commands. Section 5 includes our verification testing and a reader demonstration of these new tools. Finally, we conclude with summary observations in Section 6.

2. Visualization for Neuroimaging

JIST has several existing, advantageous visualization and GUI features that form the foundation for our visualization improvements. Here, we provide an overview of the basic JIST interface. For viewing user designed layouts, JIST provides two updated key graphical tools. Each tool enables developers to interact with the system:

- The Pipeline Layout Tool is a visual design environment for custom processing pipelines. There are three parts within the Pipeline Layout Tool: the Module Library, the Layout Panel, and the Parameter Panel. The Module Library is a list containing each module. The Layout Panel is where users can drag and drop from the Module Library to create a streamlined pipeline formatted graphically as a layout. Each module contains user customizable parameters that (edited in the Parameter Panel) that are specified and saved for pipeline execution. Taken together, developers can directly and intuitively create pipelines by selecting and connecting individual modules that have been developed by community members for use within the JIST framework. The Pipeline Layout Tool detects and prepares each module for incorporation into the user designed pipeline.

- The Process Manager is the control center of the pipeline. In the Process Manager, all of the computation tasks in a pipeline are seen as "experiments." There is an experiment for each module and for each input to be used, (e.g. multiple DTI sets). It enables multiple processing tasks (e.g., running multiple datasets through the same pipeline) to be performed in a multi-processor computing environment or through the Distributed Resource Management Application API (DRMAA) [10] which supports processing grids. Usually, more than one experiment (where an experiment consists of a single module coupled with an input file) can be run simultaneously and the process manager (scheduler) attempts to distribute tasks across multiple processors. During execution, the Process Manager collects information about the speed and memory performance of each experiment, retains any debugging information, shows the experimental results in spreadsheet format, and stores the image data result files into proper directory. The Process Manager displays one of six statuses for each experiment: READY, RUNNING, COMPLETE, NOT READY, OUT_OF_SYNC, FAILED, or COMPLETE. In a series of dependent tasks, the first task will display READY before it is run. Running tasks are listed as RUNNING until the process is finished or is canceled. Note that for RUNNING tasks, the Process Manager also displays real-time information about process status, computation time, memory usage, algorithm progress, task dependencies, and algorithm arguments. If the algorithm dependency is satisfied and the experiment is completed successfully, the experiment is listed as COMPLETE. Tasks that are NOT_READY are dependent on tasks earlier in the pipeline and should become READY when those tasks COMPLETE, fulfilling its dependencies. An experiment can become OUT_OF_SYNC if the input/output data on disk is not consistent with the input/output data used in the rest of the pipeline. (This occurs if a layout is changed after the pipeline has

already been run, and output files generated.) Experiments that have crashed or otherwise failed to produce outputs listed as "mandatory" by the module developer are FAILED. Debugging information generated by the STDOUT and STDERR streams for selected tasks is also captured and available to be viewed directly for inspection. The ability to click and view in the Process Manager therefore provides real time debugging and analyzing ability for users.

For viewing image data (as opposed to the layouts and modules), JIST is integrated with MIPAV's extensive visualization tools, which includes both slice-based and renderer-based display methods. As data is processed by a layout, output data for each module within the layout is stored by JIST and can be viewed using MIPAV's functionality. However, users of JIST-I must first find the correct files on the disk by navigating through JIST output directories, which, though structured by JIST into a systematically named tree hierarchy, can quickly become time consuming and tedious as larger datasets and more complex layouts are used. The distraction and time-consumption of locating data inhibits understanding of data progression through the layouts.

To address this concern, we developed the "show this data" functionality for the JIST-II user interface. This next generation visual functionality increases developer interaction with both The Pipeline Layout tool (Section 2.1) and Process Manager (Section 2.2) by allowing a click on an area which logically corresponds to a data processing step to trigger a much more accessible viewing of the processed data outputs. This option allows for easier viewing, analysis, and checking of input and output data for each module, thus improving management and understanding of the progression of data through the pipeline and addressing this demand of current users.

2.1. User Notes for Integrated Visualization Capabilities into the Pipeline Layout Tool

In JIST-II, to view both the input and output results, users can right click on the module and select certain input information of the module to view it in the Pipeline Panel even before the module has completed (**Figure 1**). This ability to effortlessly view input data provides great convenience for trouble shooting and analyzing the whole pipeline design.

For output, once a module is finished successfully, users can right click on the module in the Pipeline Layout Tool and select a proper experiment to view the corresponding output data. (If multiple datasets are being processed then each dataset is assigned by JIST a unique experiment number.) The generated output image data will be loaded to the GUI window automatically allowing for fast examination of the processing algorithm (**Figure 1**). This in-program visualization capability can be used to view the input/output of each module for every experiment on each step.

2.2. User Notes for Integrated Visualization Capabilities into the Process Manager

In JIST, for a loaded layout, the Process Manager displays all of experiments (which are the permutations of the modules and inputs) in that layout and their statuses. In JIST-II, each experiment can be real-time monitored and managed in the Process Manager. When a pipeline is loaded, a directed graph of module dependency is constructed for each experiment. In JIST-II, input data and the information of parameters can be accessed by right clicking on the corresponding module in the Process Manager, regardless of the Process Manager experiment status (**Figure 2**). Output data, naturally, cannot be viewed until the module is finished and the status is listed as COMPLETED in the Process Manager (**Figure 2**). During the status of NOT READY, FAILED, OUT_OF_SYNC, and RUNNING, the output of the corresponding experiment is listed in grey to demonstrate that the output is not ready to be viewed. A task's priority can also be modified in this way.

3. Continuous integration of JIST framework

As reasoned in the introduction, JIST is made open source to allow neuroimaging community members to build their own algorithms, share techniques, problems, and make contributions to the neuroimaging system. Because of the pace of development in image processing algorithms, critical design adjustments and, in turn, vast code modifications or additions are necessary. These necessary code modifications sometimes affect the reliability of critical systems. The more active community members are, the more often stability testing is required to ensure swift detection of problems. However, manual testing is a laborious and time consuming process. In addition, it may not be effective in finding certain defects. Therefore, we have developed a Continuous Integration (CI) testing technique to achieve a robust multi-platform system JIST-II.

We desired an automated testing framework that enabled rapid validation of high quality and robust software, and that also reduced the of the integration procedure (a common problem for many open source projects). We chose to deploy a Hudson Continuous Build (Oracle Corp., Redwood City, CA) server on a CentOS Linux Server (http://www.centos.org/). Hudson is an open source initiative with strong corporate and community support. Hudson directly interfaces with the existing JIST-II build scripts which are written in ant (Apache Software Foundation, http://www.apache.org). The continuous build server provides for automated validation and testing of large-scale projects. Before each test build, the server retrieves the most recent source code from the NITRC source code management system, resolves all dependencies, builds an executable, and performs testing of individual modules. This ensures that any developer test scripts or modifications are checked soon after their inclusion. The results are displayed in a publicly viewable web portal so that anyone can monitor both the current and past status of the project (**Figure 3**). Users that have submitted test-jobs can subscribe to receive e-mails to be notified of the status of their test case jobs. When a developer introduces a bug that breaks existing functionality, all users and developers can pinpoint when this change occurred and can thus fix the problem in a timely manner. (Administrators are notified when a JIST build fails). Note that others' test-case data is not openly available because many users do not wish release their data.

Current generation source control systems supported by NITRC (i.e., concurrent version system – cvs – and subversion – svn) do not support fine grained access control – a user may have read access and/or write access to the whole repository or not at all. Changes to the "core" of JIST (i.e., user interface, file i/o, interface definitions) can cause cascading problems across the entire system, so commits to this codebase must be *very* carefully vetted. Hence, only JIST team leaders at primary sites are given permission to commit to the code repository (NITRC source project "jist") other users must send proposed changes to an experienced administrator. However, we encourage open and free exchange of algorithm modules, so we created a second official repository where we permit any NITRC user both read and write access (NITRC source project "jhumipavplugins"); this repository compiles against the core. Finally, some groups have ported and/or developed major packages to the JIST framework and want to keep write access to their modules restricted to their groups. These groups are stored in separate group-specific projects on NITRC (e.g., source projects "toads-cruise" and "dots") and compile against both the core and the plugins repositories. The build files in the "build" module in the NITRC project "jist" automatically resolve the dependency structure, and we have posted instructions on how to replicate the dependency structure in graphical code development environments. However, humans are fallible and occasionally invalid code (non-compiling) is committed. The nightly build structure keeps all code bases up to date and identifies points of failure.

We realize that the open-source updates to JIST are not the only mechanism of JIST destabilization. JIST is linked to MIPAV, so changes or failures in MIPAV can cause JIST to fail to compile. We therefore

include in the continuous build process, not only the main JIST source code, which is the JIST nightly build (**Figure 3A**), but also the MIPAV nightly build (**Figure 3B**). This ensures that JIST is validated using the most up-to-date MIPAV for common libraries (**Figure 3C**).

This continuous integration system helps automate the distribution, execution, and results validation analysis of the JIST-II system. It uses the best practices on automated CI solutions to provide developers and/or testers with a better idea of progress and code integrity throughout the project lifecycle, allowing them to direct their time and expertise to more important, challenging issues, rather than debugging. The solutions presented herein may also be generally applicable to other open-source software programs facing similar challenges as JIST.

4. Testing and Validation with Modules Use Test Cases

The CI framework provides for nightly testing, but the administrative backend of the CI server is not intuitive for end users and would present a security problem if it were made publicly accessible. To provide shared CI functionality for testing without requiring that end users setup their own CI servers, we developed a test case generation system, allowing the end user to submit working modules to the CI system for nightly testing on the most updated version of JIST. There are two aspects of the continuous test case system in JIST-II:

- Generate Test Cases GUI (**Figure 4**): A newly designed GUI allows users to specify the modules, parameters, and test types they want to test within their pipeline, add their contact information and have the relevant information automatically packaged to be sent to a JIST-II manager for inclusion into the testing queue (details below). The results of these test-cases are available on the Hudson server (http://masi.vuse.vanderbilt.edu/hudson/view/Validation/).

- JIST-II manager implementation of Test Case job: The JIST-II manager provides a simple interface for CI administrators to insert test case jobs into a Hudson server and then build/delete them.

4.1. For the End-User: Generate Test Case GUI

For experiments that have successfully run (COMPLETE) on the user's machine, test cases can be generated by selecting the desired experiment and creating a test case from the drop-down menu. Multiple test cases can be selected at once, and each one will result in a separate test case file package (.zip). Using the GUI, the user inputs his contact information, reviews the automatically populated input/output information and selects the test types that should be run.

The development of these test types is an open project, but the existing tests should allow most users to effectively generate failure cases. Detecting broken modules when a program crashes or throws an error code is relatively straight forward, but detecting whether or not a module "worked" is a more subtle matter. The challenge is that output 'sameness' is relative to data type and may be use-case dependent [10]. For example, a user may include in their pipeline a noise generating module, causing output data of the pipeline to naturally be varied. Our solution was to specify categories of output 'sameness' and allow the users creating each test-case to specify their desired category and tolerance. For each module within a submitted pipeline, JIST-II offers users five kinds of test types. In each case, "observed" output refers to newly-computed data while "desired" output refers to the user-submitted output data.

- None: Run the module but do not evaluate the output.

- FileExistence: Test if the module generates an output file.
- Volume (Set the maximum number of different voxels): The test fails if the number of different voxels between the observed output and desired output exceeds the threshold set by the user. (The test output will show the actual number of different voxels.)
- Volume (Set the maximum difference for voxels): The test fails if the difference of any voxel between the observed output and desired output exceeds the threshold set by the user. (The test output will show the actual maximum difference of the voxels.)
- Surface (Set the maximum number of different voxels): The surface file will be converted to a level set. The test fails if the number of different voxels between the observed data and desired data exceeds the threshold set by the user. (The test output will show the actual number of different voxels.)
- Surface (Set the maximum difference for voxels): The surface file will be converted to a level set. The test fails if the difference of any voxel between the observed data and the desired data exceeds the threshold set by the user. (The test output will show the actual maximum difference of the voxels.)

After completion of the GUI form, test files (containing, among other things, the user's input data and the "correct" output data) are generated for each experiment. Email (or otherwise transfer) the test file packages to a JIST manager. After receiving the .zip files, the JIST manager will add the test case to the queue, to be re-tested every night. Results of the most recent and historical tests can be checked on the public website.

4.2. For the Manager: JIST-II Manager Implementation of Test Case Job

After receiving the zip files, JIST-II managers are able now able to add the test case job into Hudson server from command line. In support of this effort: the JIST command line infrastructure of the modules was extended for easier integration with the build server command line syntax, the run job system was extended to run more general test cases of layout files (in addition to all modules), and the layout file format was made more robust to changes in module definitions. JUnit (http://www.junit.org/) is used to define test cases for individual modules. While JUnit already provides extensive capabilities for defining appropriate pass/fail conditions, it can be quite cumbersome for developers to write test cases for specific modules. Users can define test cases simply by specifying sets of input resources (e.g., publically available datasets and parameters), correct results (either numeric results or files), and acceptable tolerances (e.g., numeric precision). All other details of JUnit test case creation, file parsing, and validation is handled by the JIST infrastructure so that developers will require no specific knowledge of the JUnit architecture. To add test case jobs into Hudson server and run them, managers need only need to complete three steps:

- Create job: The process of adding the test case jobs to the Hudson server has been made quite convenient and effective. Managers can download/retrieve the test suite zip folder, unzip the folder, and pick a unique name for this folder, say FILENAME. With the `create_job` shell script, managers can run the script in command line and this will automatically create a pbs file with all the test and qsub content and a config.xml file in the test case folder, which is used to create the Hudson job. In the meantime, the test job will automatically be added to the Hudson server with the default configuration. Each test job in Hudson has the job name as JistTest-FILENAME.

- Build job: With the `build_job` script, users can run the script to build the corresponding job in the Hudson server. This will automatically make the qsub task to send the test case jobs to the large clusters. The test case can then be run on the grids for each build. Developers are always encouraged to deploy computationally efficient examples, but this is not always possible or practical in the image science effort. Therefore, the testing system is implemented on existing grid processing resources.

- Delete job: If users want to delete jobs on Hudson, there is the `delete_job` script.

The workflow of tests and test environments can be graphically expressed in the GUI of JIST, Hudson time trend portal, or as text files, conveniently. Based on this continuous validation tests, users are be able to extend the functionality of the image processing algorithm as well as develop continuous build server test cases, validation procedures and reporting mechanisms. Additionally, interested parties are able to precisely duplicate the JIST build server for their own internal validation.

5. Work Validation and Demonstration of New Features

To validate the JIST-II framework, we evaluated DTI processing pipeline on a public multi-modal dataset [11]. This processing layout used in the test is based on the previously reported CATNAP software [12]. After successfully running the layout on a local machine, we created test-cases for each module using the Generate Test Case GUI to select the test type, set the threshold, and generate test case zip folders for each output (**Table 1**). The test suites compare the expected (user-generated) results with the actual (up-to-date JIST generated) one and generate a report on whether the algorithm is valid and the output as expected, as well as a quantitative figure for how similar they are. Results of recent tests in action may be viewed on our public CI server (http://masi.vuse.vanderbilt.edu/hudson/view/Validation/).

5.1. Tutorial/Software Instructions

JIST-II aims to be a continuously stable framework for driving continuous innovation. Therefore we recommend retrieving and using the most recent releases of both JIST and MIPAV. We have indicated that these recent additions to JIST will result in backward incompatibility by a major version number change. *However, note that we do not label builds as "JIST-II" – this nomenclature is only used for clarity in this manuscript to distinguish between the original feature set and the newly presented one.* "Stable/Frozen" versions of JIST (without JIST-II features) are available for legacy MIPAV releases on the NITRC website (http://www.nitrc.org/projects/jist).

This section is designed to help you (the reader) test JIST and its new features. If you are a first time user, it is recommended that you *thoroughly read* the instructions here and on the NITRC:JIST wiki: http://www.nitrc.org/plugins/mwiki/index.php/jist:MainPage.

- MIPAV: As JIST is a plugin to MIPAV, having MIPAV is a prerequisite. It is available here: http://mipav.cit.nih.gov/. *The JIST wiki includes important recommendations that should be read before installing MIPAV for JIST.*

- JIST: JIST can be downloaded from the NITRC webpage (link above) or directly from http://masi.vuse.vanderbilt.edu/hudson/view/Builds/.

    - Specific, accurate installation instructions are available on the JIST wiki.

- For all users: A relatively simple processing layout that demonstrates the new features of JIST-II is available on the wiki. Tutorial-like instructions for running and demonstrating the new features are given. (http://www.nitrc.org/plugins/mwiki/index.php/jist:Tutorial)
- For advanced users: This paper used a modified version of an MRCAP DTI processing layout for validation of the new features. Advanced configuration is necessary to run this layout. Please see: http://www.nitrc.org/plugins/mwiki/index.php/jist:DTI_Tutorial
    - The DTI data and processing layout used in this manuscript may be downloaded there.

6. Conclusions

JIST-II maintains the advantages of rapid prototyping and large-scale processing while introducing the visualization, continuous integration, and user-test-case capabilities. Users can now view the processing information and data of input/output in real time both from the pipeline layout and process manager, and developers can conveniently generate test cases to run on the large clusters continuously (nightly) for any pairing of input and output data. These functionalities permit users to focus on implementing innovative design for their image processing pipelines rather than troubleshooting module incompatibility, spending time searching for data, or implementing common functionality that is desirable for all image analysis tools.

7. Acknowledgements

This research was supported in part by NIH/NINDS 1R03EB012461.

A Figures

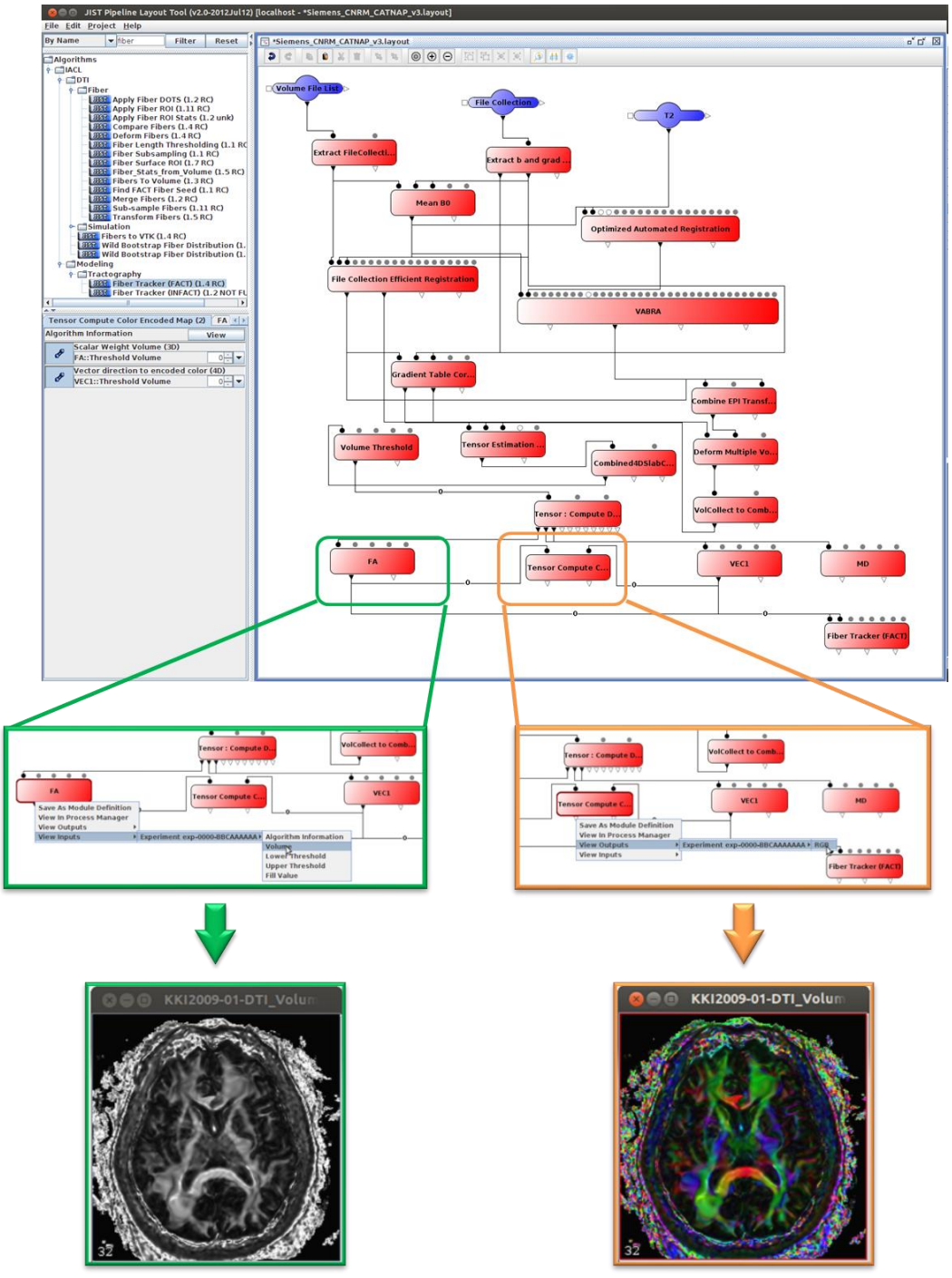

Figure 1: JIST Layout Tool For the visualization requirement, the Pipeline Layout Tool adds the functionality of "view input/output." When a module completes, it allows users to probe which datasets are processed or generated in real time. This addition provides important information in an intuitive framework.

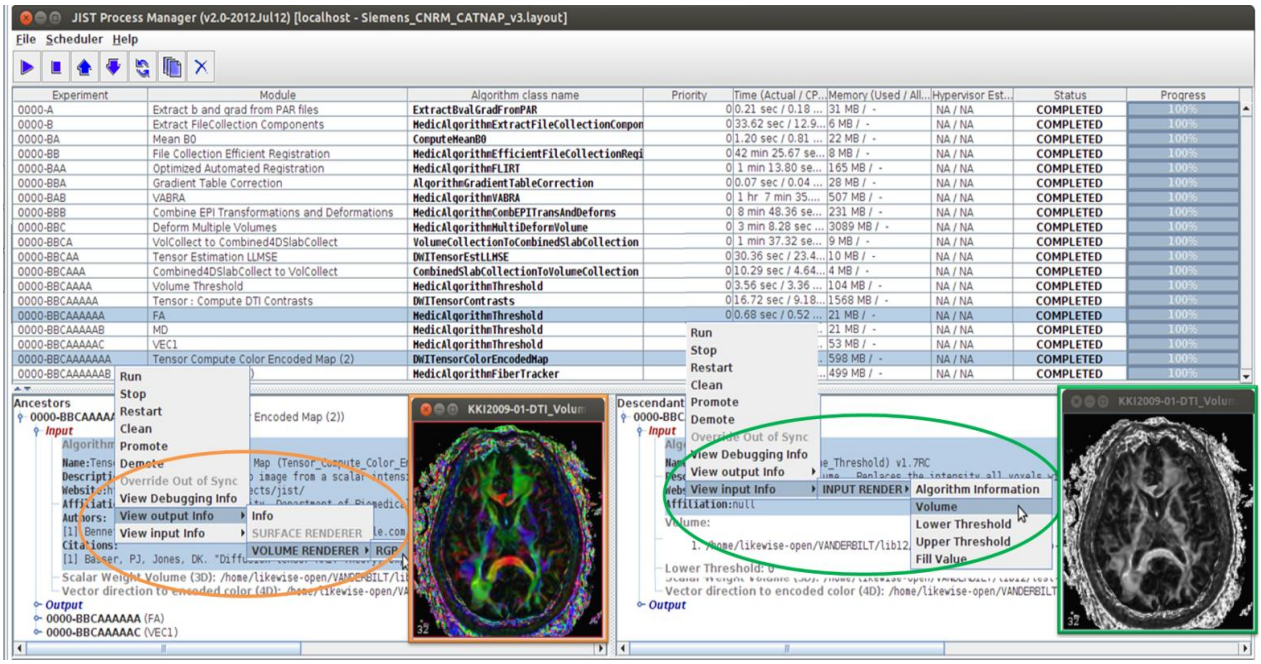

Figure 2: JIST Process Manager The JIST Process Manager integrates with the MIPAV visualization capabilities to provide viewing of input/output parameters as well as image data. Using Java user interface tool, users can right click on each row of the process manager to bring up a context menu with input/output information for each task. Incomplete outputs will be gray while completes datasets are black (able to be selected).

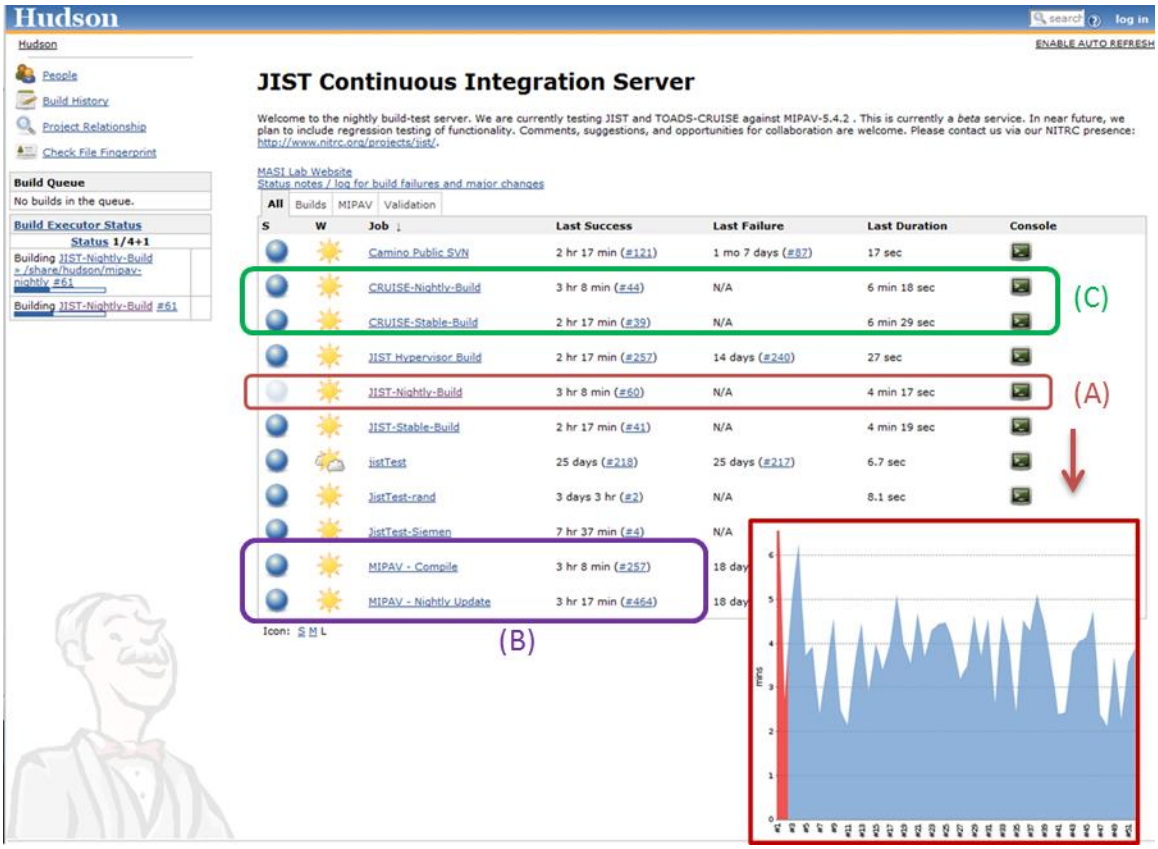

Figure 3: Hudson Continuous Integration Webpage: JIST is integrated with the Hudson server to perform continuous integration test, which can retrieve the most recent source code from the NITRC source code management system, resolve all dependencies, and build executable jobs to perform validation test for JIST framework. The Hudson graph on the right (**A**) shows the testing trend of JIST nightly build. The other tabs on the webpage give the user access to the most recent JIST-CRUISE builds (nightly and stable) (**C**), the MIPAV test builds (**B**), and the Validation page for the user submitted test cases.

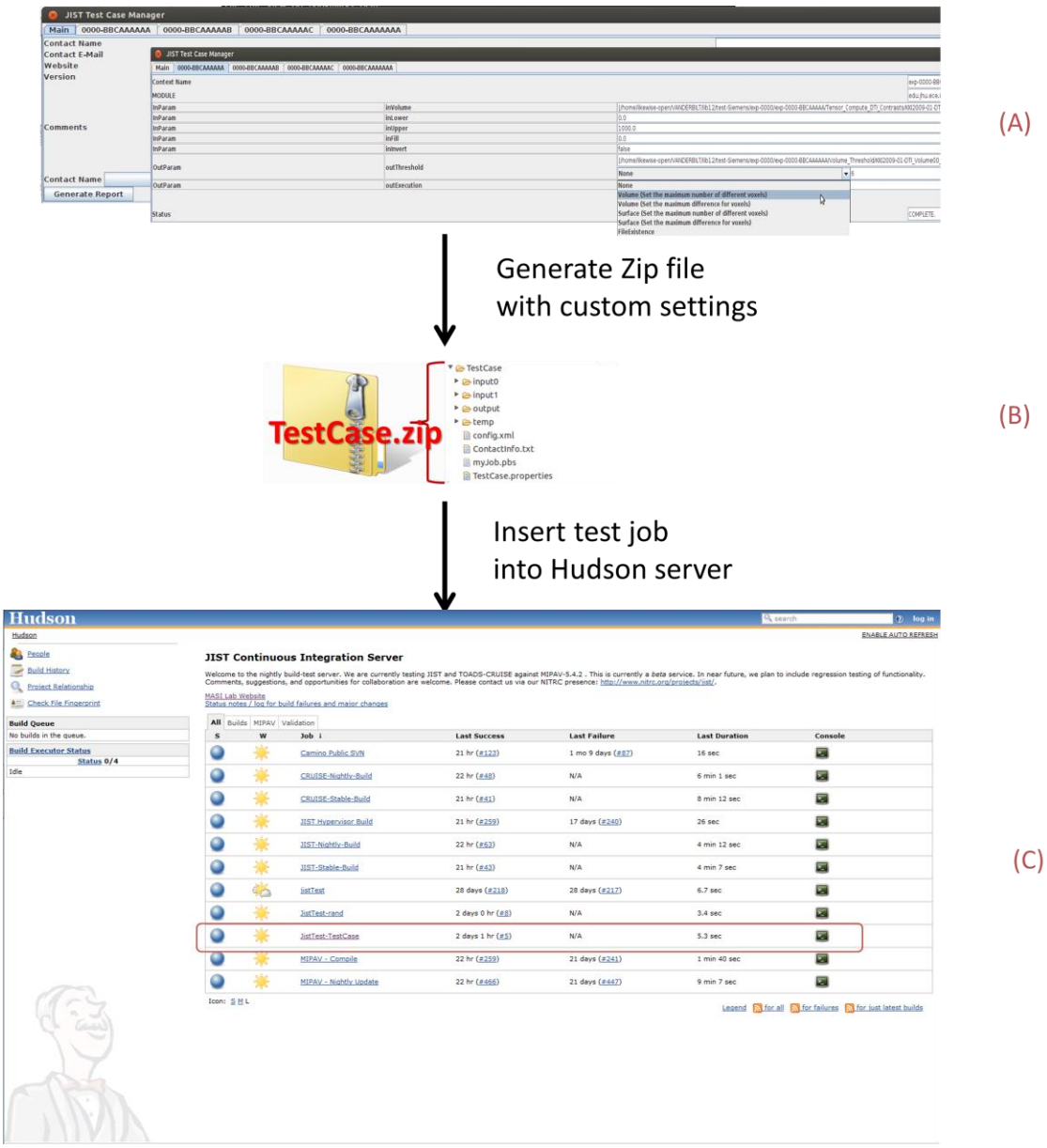

Figure 4: Test Case Explanation JIST-II provides users with the Generate Test Case GUI to perform real-time monitoring of each method's functionality with minimal additional work on a per-module basis (**A**). The two fundamental steps are: (1) users generate test case files in the Generate Test Case GUI which outputs a 'TestCase.zip' file (**B**) and email the file to JIST-manager, (2) A JIST-II manager then uses simple commands to automatically detect and add the test job into Hudson server. The module is run, the returned output results are compared to the user-submitted outputs, and the results are posted on the Hudson server webpage (**C**).

Table 1 Test case performance on large clusters

Module	Average execution time for different test types on multiple modules (*seconds*)				
	FileExistence	Volume-I	Volume-II	Surface-I	Surface-II
JistTest-IACL-MRCAP-SmoothBrainMask	34.69	5.98	5.95	106.25	106.23
JistTest-IACL-MRCAP-MultiCrop	1.89	5.09	5.08	-	-
JistTest-IACL-MRCAP-NoiseEstimation	17.30	13.12	13.10	-	-
JistTest-IACL-MRCAP-ExtractFileCollection	28.45	199.50 (34volumes)	194.69 (34volumes)	-	-
JistTest-IACL-Siemens-ExtractFileCollection	26.85	193.09 (34volumes)	195.37 (34volumes)	-	-
JistTest-IACL-Siemens-FA	1.07	5.36	5.25	-	-
JistTest-IACL-Siemens-MD	1.08	5.06	5.03	-	-
JistTest-IACL-Siemens-TensorComputeColorEncodedMap	4.89	5.19	5.12	-	-

Table 1: Sample Test-Case Run-times: As an example, Table 1 shows the execution time for testing modules in the pipelines of the MRCAP and Siemens projects [13]. The columns are the different test types that can be applied for test-cases (explained previously). All testing configurations and results can be viewed here: http://masi.vuse.vanderbilt.edu/hudson/view/Validation/. The jobs for test cases are under the Validation tab on the Hudson website and each test case job has the name format as "JistTest-LABNAME-PIPELINE-MODULE". In this example, we have tested the ExtractFileCollection module, used in both the MRCAP and Siemens pipelines, and which had 34 volumes to compare. Note the SmoothBrainMask module test, in which both surface and volume differences were compared. The NoiseEstimation module (from the CRUISE library) was tested to ensure CRUISE validation. With the tests run on large clusters results are produced in seconds. This near real-time feedback is useful for users analyzing their module performance.

References

[1] B. C. Lucas, J. A. Bogovic, A. Carass, P. L. Bazin, J. L. Prince, D. L. Pham, and B. A. Landman, "The Java Image Science Toolkit (JIST) for rapid prototyping and publishing of neuroimaging software," *Neuroinformatics,* vol. 8, pp. 5-17, 2010.

[2] B. Landman, B. Lucas, J. Bogovic, A. Carass, and J. Prince, "A Rapid Prototyping Environment for NeuroImaging in Java," *Neuroimage,* vol. 47, p. S58, 2009.

[3] B. A. Landman, "CATNAP and JIST: DTI Processing Made Ridiculously Simple," presented at the International Society for Magnetic Resonance in Medicine Educational Sessions, 2009.

[4] B. Lucas, B. Landman, J. Prince, and D. Pham, "MAPS: a free medical image processing pipeline," *Organization for Human Brain Mapping,* 2008.

[5] M. J. McAuliffe, F. M. Lalonde, D. McGarry, W. Gandler, K. Csaky, and B. L. Trus, "Medical image processing, analysis and visualization in clinical research," 2001, pp. 381-386.

[6] R. F. Boisvert, J. Moreira, M. Philippsen, and R. Pozo, "Java and numerical computing," *Computing in Science & Engineering,* vol. 3, pp. 18-24, 2001.

[7] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman, "Benchmarking Java against C and Fortran for scientific applications," 2001, pp. 97-105.

[8] P. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*: Addison-Wesley Professional, 2007.

[9] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf,* 2006.

[10] R. M. French, *The subtlety of sameness: A theory and computer model of analogy-making*: The MIT Press, 1995.

[11] B. Landman, A. Huang, A. Gifford, D. Vikram, I. Lim, J. Farrell, J. Bogovic, J. Hua, M. Chen, S. Jarso, S. Smith, S. Joel, S. Mori, J. Pekar, P. Barker, J. Prince, and P. van Zijl, "Multi-Parametric Neuroimaging Reproducibility: A 3T Resource Study," *Neuroimage,* vol. 4, pp. 2854-2866, 2011.

[12] B. A. Landman, J. A. Farrell, C. K. Jones, S. A. Smith, J. L. Prince, and S. Mori, "Effects of diffusion weighting schemes on the reproducibility of DTI-derived fractional anisotropy, mean diffusivity, and principal eigenvector measurements at 1.5T.," *Neuroimage,* vol. 36, pp. 1123-38, Jul 2007.

[13] W. R. Gray, J. A. Bogovic, J. T. Vogelstein, B. A. Landman, J. L. Prince, and R. J. Vogelstein, "Magnetic Resonance Connectome Automated Pipeline," *Arxiv preprint arXiv:1111.2660,* 2011.