SCDU testbed automated in-situ alignment, data acquisition and analysis

Thomas A. Werne[1], Udo J. Wehmeier[2], Janet P. Wu[3], Xin An[4], Renaud Goullioud[5], Bijan Nemati[6], Michael Shao[7], Tsae-Pyng J. Shen[8], Xu Wang[9], Mark A. Weilert[10], Chengxing Zhai[11]

[1] *Thomas.A.Werne@jpl.nasa.gov*, [2] *Udo.J.Wehmeier@jpl.nasa.gov*, [3] *Janet.P.Wu@jpl.nasa.gov*, [4] *Xin.An@jpl.nasa.gov*, [5] *Renaud.Goullioud@jpl.nasa.gov*, [6] *Bijan.Nemati@jpl.nasa.gov*, [7] *Michael.Shao@jpl.nasa.gov*, [8] *Tsae-Pyng.J.Shen@jpl.nasa.gov*, [9] *Xu.Wang@jpl.nasa.gov*, [10] *Mark.A.Weilert@jpl.nasa.gov*, [11] *Chengxing.Zhai@jpl.nasa.gov*

Jet Propulsion Laboratory, 4800 Oak Grove Dr, Pasadena, CA, USA

ABSTRACT

In the course of fulfilling its mandate, the Spectral Calibration Development Unit (SCDU) testbed for SIM-Lite produces copious amounts of raw data. To effectively spend time attempting to understand the science driving the data, the team devised computerized automations to limit the time spent bringing the testbed to a healthy state and commanding it, and instead focus on analyzing the processed results. We developed a multi-layered scripting language that emphasized the scientific experiments we conducted, which drastically shortened our experiment scripts, improved their readability, and all-but-eliminated testbed operator errors. In addition to scientific experiment functions, we also developed a set of automated alignments that bring the testbed up to a well-aligned state with little more than the push of a button. These scripts were written in the scripting language, and in Matlab via an interface library, allowing all members of the team to augment the existing scripting language with complex analysis scripts. To keep track of these results, we created an easily-parseable state log in which we logged both the state of the testbed and relevant metadata. Finally, we designed a distributed processing system that allowed us to farm lengthy analyses to a collection of client computers which reported their results in a central log. Since these logs were parseable, we wrote query scripts that gave us an effortless way to compare results collected under different conditions. This paper serves as a case-study, detailing the motivating requirements for the decisions we made and explaining the implementation process.

Keywords: SCDU, SIM-Lite, Automation, Automated Analysis, Distributed Processing

1. INTRODUCTION

The Spectral Calibration Development Unit (SCDU) is one of a series of testbeds for SIM-Lite. Its purpose is to demonstrate that the stellar spectral effects which SIM-Lite will be subject to can be calibrated to better than one microarcsecond. Results from work performed two years ago on the testbed[1,2,3] were successful, and the work documented in this paper assisted in continuing the progress.[4,5,6] For a more detailed description of the testbed, see Nemati, et. al.[4]

The purpose of this paper is to demonstrate that for complicated systems such as SCDU, computing resources can be used to automate many tasks—both mundane and complex—to drastically simplify and improve data collection, analysis, and system alignments. An additional gain is improving and rapidly-evolving science-data understanding: by systematically processing the data and storing the data using automated analysis methods, the results are easily indexed and weeks of data can be reprocessed with updated analyses and viewed with ease. Finally, combining these techniques and programs creates a very real potential to command and control the entire testbed from a Matlab interface.

Send correspondence to: Thomas A. Werne
E-mail: thomas.a.werne@jpl.nasa.gov
Telephone: 1 818 354 3008
Address: Jet Propulsion Lab, 4800 Oak Grove Dr. M/S 168-514, Pasadena, CA, 91109, USA

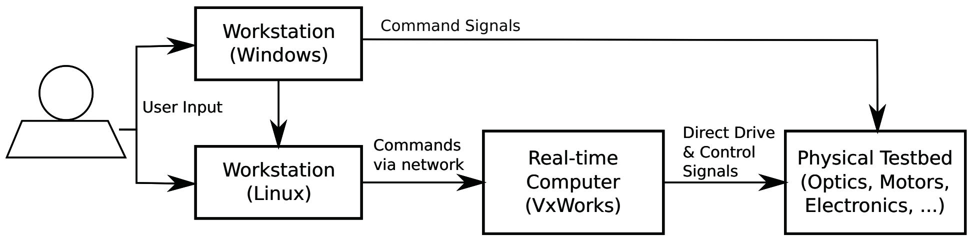
Workstation (Windows)
Command Signals
User Input
Workstation (Linux)
Commands via network
Real-time Computer (VxWorks)
Direct Drive & Control Signals
Physical Testbed (Optics, Motors, Electronics, ...)

Figure 1. High-Level SCDU Computing Testbed Architecture

2. SIMPLIFIED SCDU PHYSICAL ARCHITECTURE

To appreciate the reasons for introducing automation into the SCDU testbed and see where the different solutions fit, it is important to understand the physical architecture of the system. A simplified schematic of the testbed is shown in Figure 1, where the emphasis is on the different computers and communication schemes that make up the command and control system.

All the positionable components of the testbed are controlled directly by a computer running the VxWorks real-time operating system. Typically, the cameras—an angle-tracking camera (ATC) and a fringe-tracking camera (FTC)—are also commanded by the real-time computer, although occasional special experiments require a third camera which was controlled by a Windows PC. Computation on the real-time computer is reserved exclusively for command sequencing, control loops, and data collection. A separate Linux workstation is used to send low-level commands over a network interface.

For a typical experiment, the operator writes a script on the Linux workstation—using the Tool Command Language (TCL) scripting language—that contains motion commands (e.g. open/close shutters, change filter wheel settings, set motor positions) and computational commands (e.g. define control system track points, active tracking loops, enable data collection). A custom function library translate these script commands into a sequence of rudimentary commands and ships them across a network connection. The real-time computer acts appropriately on these commands (updating internal control loop state variables, driving discrete lines, sending commands over specialized communication links, etc.).

Figure 2 shows the flow of data through SCDU from sensor collection to storage for processing. Raw sensor data is captured by the real-time system and stored locally in RAM. Data required for the local control systems is analyzed onboard. If the data is relevant to the experiment, it is sent across the network to the local Linux workstation and stored on a local hard disk. A program intermittently that scans through the data on the local disk, converts it to a Matlab-readable .mat file, then sends the final data product off to network storage, where it can be analyzed at will.

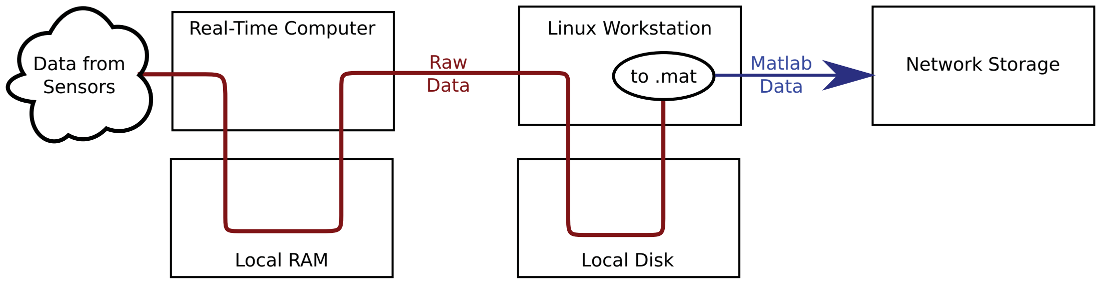

Figure 2. SCDU Data Collection Path

3. MOTIVATION FOR AUTOMATION

The main reason for introducing computerized automation into the SCDU testbed is that the testbed operator and analysts spent much of their time performing repetitious, mundane tasks and unspecialized analyses that required little human intervention. In particular, the primary data product of interest is a six-channel narrow-angle bias estimate, an analysis that requires several different data sets and a twenty minute computation. Since the siderostat servos have a short range of travel, the testbed operator would often need to perform a coarse pointing alignment using a different set of motors. Because this alignment need not be real-time and is only done when data is not collected, it was not implemented in the real-time system. This task was somewhat onerous because the second servo set had a hysteretic response.

A second impetus suggesting automation is that, as described in Section 2, the computing environment for commanding the SCDU testbed is reasonably heterogeneous, requiring multiple computers and producing several different data products. Occasionally, the team performed special experiments with different hardware (e.g. higher performance cameras) that required the use of additional computers. While the operator or a small group of team members could certainly coordinate actions across these different platforms, the process is error-prone. For example, if the experiment requires the operator to initiate five different actions across three different computers, the process is much slower and it may take a few tries to find the correct sequencing. It is better to bring the command and control interface to a single computer program that still gives the operator complete control over the entire testbed.

4. DISTRIBUTED PROCESSING

Section 3 briefly notes that scientifically-meaningful results are extracted using a lengthy Matlab analysis which, requiring several data collection runs, is typically initiated by the operator or an analyst. This process is time consuming and requires diligence on the part of the user to continuously process data, so that there is never a backlog of analyses.

In reality, the only role that the human operator has in this process is to collect the proper data sets, package them into a single analysis command, and summarize the results. With the help of a domain-specific language (described in Section 6), it is very easy for a computer to automatically create the analysis command. However, instead of executing that command on the testbed workstation—which should only interface with the real-time system and not be burdened—it is preferable to delegate these analyses to a different computer. The implementation of this idea is a custom-written distributed processing system that sends out the analysis command strings to any of a set of network-connected computers. This distributed processing "farm" turns out to be an extremely powerful tool that greatly decreased the time spent on manual data analysis.

4.1 Algorithmic Description

Figure 3 schematically describes how the distributed processing system works. Once sets of data for an entire experiment have been collected and are ready for analysis, a process command is generated and appended to a "command list" queue file. Multiple client machines (or many processes, in the case of multi-processor computers) wait, each periodically requesting a command to process. When the analysis server process receives such a request, it pops the first entry from the queue and sends it out to the client machine (or a null command if the queue is empty). The analysis server also collects identifying information about the requesting process and computer, then logs the command to another file.

The received command is interpreted by the client machine. In general, the command could be a program execution, a request to summarize data that has already been analyzed, a script call, etc. In the case of SCDU, since all of the data analysis is done with Matlab, these received commands are always strings which can be run in Matlab with the "exec" function. Typically the string is a single analysis run, although multiple analyses can be written into a single script, whose call is then farmed out. Usually, the function(s) called by the command contain code that write the analysis results in a central location on a server in an ASCII log file that can easily be parsed for quick data summarization (e.g. the log is a comma/tab/pipe -separated-variable file).

When the client machine finishes running the command, it communicates the result back to the server (e.g. successful completion, erroneous completion) and requests a new command. If the command queue is empty, the client goes to sleep and wakes up at predetermined intervals to query the server.

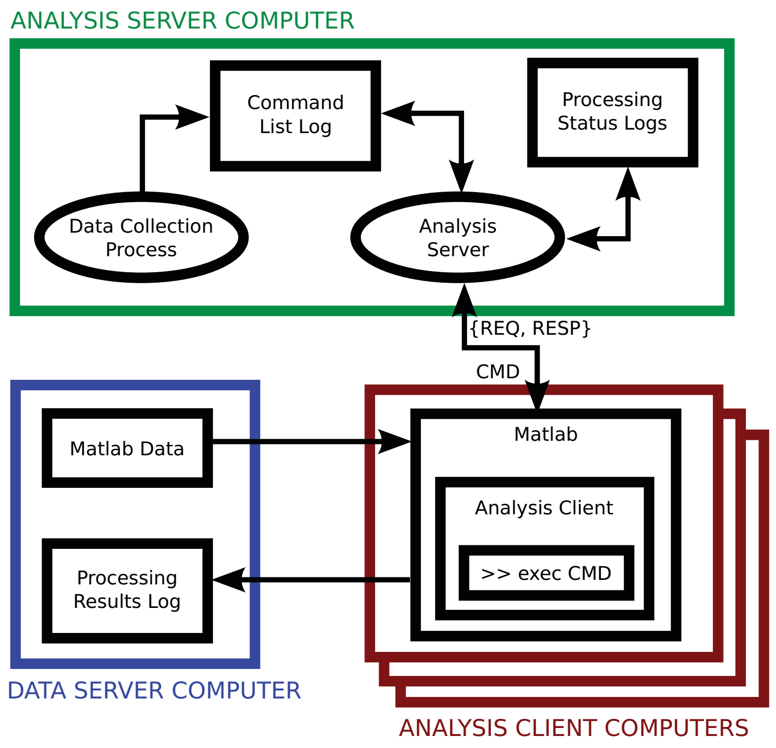

Figure 3. Distributed Processing Program Logic

4.2 Results

This addition to the analysis system was surprisingly useful. By distributing the standard analyses out to several computers, the results were available essentially in real-time without requiring a dedicated analyst. Secondly, since all of the analyzed data were logged in a parsable ASCII log file, it was trivial to write a data-summarization Matlab script. The script produces numerous graphs—one for each standard analysis result—of the time-evolution of the analysis results. The uniform data logging and real-time reporting of the distributed processing system plus this summary script allows for an instant look at the history of the testbed.

Secondly, by leveraging a large collection of client processes—approximately one process per core on numerous quad-core and dual-quad-core computers—the team was able to very quickly complete re-analysis campaigns. For instance, the team would occasionally need to recompute the data product with an improved analysis technique using many months of data. With the distributed analysis, the result was available in hours instead of days.

5. TCL + MATLAB

A second automation that the team applied to the testbed was incorporating Matlab analysis into the TCL scripts. While TCL is capable of performing the same operations as Matlab, the latter is better suited for numerical calculation and is simpler for algorithmic development. The team used TclMatlab[*] to provide the interface layer between TCL and Matlab.

5.1 TclMatlab

TclMatlab is a TCL library that allows the user to spawn a Matlab session from a TCL script, pass variables back and forth, and call functions on the Matlab variables. See Figure 4 for an example with trivial code that shows how the interface layer works. In words, after linking the TCL session with TclMatlab, the user creates

[*]available: `http://tloona.sourceforge.net`

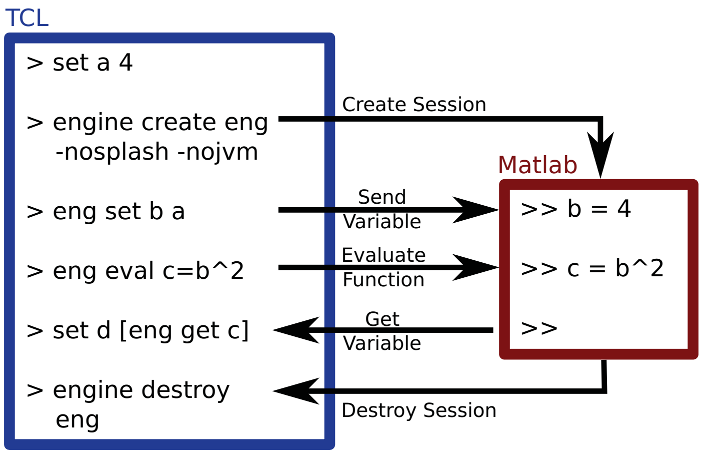

Figure 4. TclMatlab Code, Results, and Algorithm

a Matlab session (with a handle "eng") using the `engine create eng` command and shuts down the session with `engine destroy eng`. While the session is active, variables can be sent to the Matlab session using the command `eng set matlab_variable value` and retrieved with `eng get matlab_variable`. Function calls are executed by first defining an string in TCL with the desired function call, then running `eng eval func_string`.

5.2 Usage for Alignments

The most important usage of TclMatlab for the SCDU team was to automate certain alignments and collect state information. While most of the alignments are either very coarse and need only be performed occasionally and in air (e.g. mechanical alignment of the optics) or are very precise and must be performed in vacuum throughout an experiment (e.g. angle-tracking), there are a few rough alignments that must be performed in vacuum on occasion, but do not have real-time requirements.

For example, as alluded to earlier, the angle-tracker control loop articulates the siderostats so that the reflected light impinges on the angle-tracker camera at pre-defined locations (ideally so that the reflected light follows the same path as the outbound light). Since the track points are pre-defined and fixed on the camera, they are in a sense arbitrary relative to the actual fringe-tracker system alignment. As the physical system drifts and the location of the outbound spot moves, the reflected light is still controlled to the track point, which will eventually be far from the ideal tracking location. In order to correct for this problem, an "outbound alignment" must be performed (usually about once per week). This alignment can be done using a control loop. However, achieving the necessary precision requires motor motions at their lower-limits, and as mentioned above, the response of the motors is quite hysteretic in this regime. This makes implementing a simple control loop purely using TCL quite difficult. Plus, since the collected data is stored in Matlab-accessible data files, it was much simpler to design and implement the controller using Matlab, which passed the motion commands back out to the TCL level.

5.3 State Log

The second major use for the TclMatlab interface library is to produce a running state log. For SCDU, the system state is defined to be a collection of sensitivities, alignments, and calculated values. This state is a simplified description of the testbed and can be used to diagnose problems and better understand processed output. As above, the algorithms required to compute the state are nontrivial and not suited to TCL, plus the data is all in Matlab-readable files, so it is reasonable to do the analysis in Matlab and pass parameters (e.g. data set identifiers) back-and-forth between TCL—which commands the system to produce the data sets—and Matlab.

The state log also enabled adding a procedure into the data summarization script to show the state history. Discussed in Section 4.2, the distributed processing system is useful because it allowed the team to generate a script that produces summary plots showing a time-series of the collected data results. Going beyond the scientific results in those logs, the state log permits a glance at the state of the system at any given time or over a time period, which can show correlations and drifts that may not be otherwise apparent.

6. DOMAIN-SPECIFIC LANGUAGE

When it was originally designed, the TCL frontend of the system had essentially a one-to-one relationship between commands callable by the user, and backend commands sent to the real-time system. That is to say, the basic system was functionally identical to sending single commands to the real-time computer over the network. While the testbed can be operated using this set of commands (indeed it was for several years), writing scripts using this paradigm can be likened to programming at the assembly level. The scripts are very long, fairly intricate, composed of many pieces that have been copied from other scripts, and are easily broken.

As time progressed, it eventually became clear that this method of constructing experiment scripts needed to change. The average script (which contained all of the code necessary to collect data for an entire day) was somewhere between 700-1000 lines, and was incomprehensible. Additionally, it was easy to make small errors, such as setting up the system in the wrong mode for an experiment, placing a calibration run after an experiment instead of before it, etc. The cause of these issues is that the experiments were being written at a far too low level. The solution was to design a new functional layer to sit on top of the TCL code. The result is essentially a small domain-specific language.

The team took a hard look at the daily operations of the testbed and identified a handful of experiments that were repeated, day-in and day-out. With these experiments in hand, along with a per-experiment collection of parameters and reasonable default values, a collection of new functions were written. Each function was designed so that the system could be in an arbitrary state on entry, and the system was always left in the same state on exit. This uniformity meant that the system was never in an unknown state, and so the functions could be strung together in any order, following the natural idea that it should be possible to perform the experiments in any order. By making the functions parameterizable, it was almost always possible to use the functions without falling back and using the raw code. Conversely, since a set of reasonable default values were always available, it was seldom necessary to actually provide parameters to the functions; calling them with an empty parameter list typically yielded the correct functionality.

An immediate benefit from switching to this new function set is that the experiment scripts dropped in length to around 40 lines per day. Besides making the scripts more readable by simply decreasing their length, the code was literally self documenting. Since there was now a bijection between function calls and experiments, any team member could read the scripts and immediately understand exactly what experiments were being run.

7. INTERFACING WITH WINDOWS

Near the end of SCDU's operation, a question arose whose solution involved replacing the angle-tracker camera with a new camera that had a finer pixel pitch. The camera came equipped with several software interfaces: a proprietary frame grabber from the vendor, a Matlab library, and a C-library. Since there was not enough time to develop a custom program using the C-library and the Matlab library did not have the capabilities of the vendor software, the team elected to use the vendor software. While this piece of software provided the most features, it was also Windows-specific, which meant that it could not easily be integrated into the existing testbed software system.

The experiment required a very specific synchronization between the real-time system operation and the camera software on the Windows PC. This presented two separate problems:

1. Even with the two computers next to each other, coordinating actions on each machine separately is a tedious, accident-ridden process. Yet, since the experiment required the use of two different platforms, there was not an existing simple way to combine the computer operations to a single interface.
2. Because the experiment had to be run many times, using the non-scriptable camera capture software was also tiresome. Gathering a set of data required a dozen button clicks, and there was no built-in way to automate collection.

The solution to this process ended up being threefold: communication between the computers was handled by installing Cygwin[†] and using its command-line Secure Shell (SSH); data-collection automation was made possible using the Windows-based keyboard+mouse macro recorder/compiler/player AutoHotkey[‡] (AHK); and the entire experiment was orchestrated using Matlab's `system()` command.

7.1 Cygwin + SSH

Cygwin is a collection of programs, along with a Unix emulation layer, that permits a user to execute programs using with a Unix-like command-line interface. It comes with all of the standard programs found on a Linux system (e.g. `ssh`, `vi`, `emacs`, `sed`, `awk`, `gcc`, ...). By adding the Cygwin directory to the Windows PATH variable, these programs can be called by any other program. Once Cygwin is installed on a computer, commands can be sent to a remote machine using the `ssh` command, just like in Linux. The result is that, from the Windows machine, commands to run SCDU scripts can be passed through an SSH connection to the SCDU Linux workstation, which are then executed by that machine. See Section 7.4 for an example that uses Cygwin and SSH.

7.2 AutoHotkey

AutoHotkey is a program that allows the user to write and record keyboard and mouse macros on a Windows computer, allowing them to be replayed at a later time. This means that any collection of keystrokes and mouse-clicks can be saved and recalled at any time. The macros are saved as text files in a simple-to-use scripting language, so one generic macro can be recorded, then edited to produce several different playable macros (e.g. parameters that are typed into a dialog box can be changed). Finally, the macros can be compiled to executable files which accept parameters from the command line. For SCDU, this allowed the team to record a macro that sets up the camera at different frame rates and collects data for a parameterizable amount of time, all callable from a single executable. See Section 7.4 for a trivial example macro script.

7.3 Matlab `system()` Command

While the Matlab `system()` command is perhaps not often seen in analysis scripts, combined with the above programs, it is an exceptionally powerful tool. This command takes a single string parameter executes it as a separate program on the computer, and returns the text output. On a Linux machine, this is similar to opening a new terminal and executing the string. On Windows, this is basically the same thing as opening a command prompt (i.e. running `cmd` in the Run dialog box). As long as the program can be found on the system path, it will be executed. Joining this capability with the two programs listed above, Matlab can be used to sequence the entire experiment: `system()` commands that call Cygwin's `ssh` can send data collection calls, control the testbed, and `system()` commands that invoke compiled AHK executables set up the camera and collect data locally. See Section 7.4 for examples of using this command with AHK and Cygwin's `ssh`.

7.4 Example Code

Figure 5(a) shows a trivial example of a script produced by the AHK recorder. The script runs the Windows calculator and computes $4 * 10 + 2$. The lines in black are system setup commands provided by AHK. The olive code activates the Windows Run dialog box (via Windows-key + 'R'). The teal code enters the word "calc" into the command box and runs it, which spawns the Windows calculator. The purple code enters the command into the calculator and executes it. The code in bold are user actions, and the rest are produced by AHK.

The commands in the upper half of Figure 5(b) show how to execute a compiled AHK script from within Matlab. After AHK compiles a script into an executable and the executable is saved to the hard disk, Matlab's `system()` command can run the executable. The commands in the lower half of that same figure give a trivial example of using Matlab to execute a command on a different machine by using SSH through Cygwin. The code in black is executed directly by Matlab. The code in green is executed by Cygwin (using its bash shell). The code in blue is executed by the remote computer on the receiving side of the SSH connection. The result of this command is a file named "pingfile.log" in the user's home directory that shows the output of running "ping" on the local computer.

[†]available: `http://www.cygwin.com`

[‡]available: `http://www.autohotkey.com`

```
WinWait, Program Manager,
IfWinNotActive, Program Manager, ,
WinActivate, Program Manager,
WinWaitActive, Program Manager,
Send, {LWINDOWN}{LWINUP}
WinWait, Run,
IfWinNotActive, Run, , WinActivate, Run,
WinWaitActive, Run,
Send, calc{ENTER}
WinWait, Calculator,
IfWinNotActive, Calculator, , WinActivate,
Calculator,
WinWaitActive, Calculator,
Send, 4*10{+}2{ENTER}
```

(a) Autohotkey

```
>> % Call Autohotkey Script
>> ahk_script = 'ahk'
>> system(['C:\scripts\' ahk_script '.exe'])

>> % Call Program over SSH
>> system([                                       ...
     'bash -c "                                  ' ...
     '    ssh -l `whoami` localhost              ' ...
     '       \"ping localhost\"> pingfile.log   ' ...
     '"                                          '])
```

(b) Matlab

Figure 5. Code Examples

8. TEMPERATURE LOGGING, A LA JANET

The sensitivity of acquiring measurements at picometer level precision and accuracy as stated requires fine alignment stability. Anomalies that appeared in analysis results were often traced to temperature fluctuations on the order of a few degrees. There is no doubt that temperature plays a vital role in maintaining system alignment. The price of creating such a sophisticated auto-aligning testbed is paid for with heat generated by the various moving parts of the automated components. Temperatures outside of the chamber also influenced those inside when the system is under vacuum. To monitor fluctuations both internal and external to the vacuum chamber tank, a fully automated temperature logging system was developed to record temperatures non-stop 24 hours daily. The logger runs independent of the testbed system so measurements could be made even when experiments are not running. This was primarily to monitor laboratory ambient temperature changes when the testbed chamber is open, so coarse alignment corrections could be performed. Adjustments would be fruitless if the differences between ambient room and closed chamber temperatures are relatively large. Such conditions would negatively impact the accuracy of the initial alignment for experiments once the system reentered vacuum.

The logging system employs several resistive temperature devices (RTD) placed outside of the vacuum tank and in strategic areas of the testbed optical bench. They are connected to a commercial off-the-shelf datalogger, which was managed under a Windows PC through a serial port. Matlab was chosen to automate data acquisition to take advantage of the Instrument Control Toolbox which simplified programming a remote communication interface to just three built-in functions. The logger accepts Standard Commands for Programmable Instrumentation (SCPI) language commands to configure the instrument and return data and responses. The instrument was instructed to scan all 19 RTDs five times consecutively and return the average temperature for each RTD (process takes approximately 15 seconds total). This value is recorded and scans immediately begin again. The data is archived locally on the PC and plotted every 15 minutes, then posted onto the dedicated project website allowing remote monitoring by team members to identify any gross fluctuations that may impact system stability. It was deemed unnecessary to post data in real-time since it was not expected that dramatic temperature changes would occur rapidly. At the end of each day, the control program searches for the experiment log and matches the temperature data to each corresponding testbed experiment made that day. Later analysis could then be performed to examine whether temperature had affected alignment during those experiment runs.

9. FUTURE WORK

Taking all the pieces of software and techniques described to this point, there is an obvious final step to attempt. At this point SCDU still requires an operator to bring it up to an operational state. There are two primary reasons:

1. Some alignments have not been automated

2. Human decision-making is required to decide when the system is operational

Taking for granted that the remaining alignments can be automated, by introducing the TclMatlab and command-execution-over-SSH, there is no reason that the human must remain in the loop at this level. After defining a set of metrics for "acceptable operational status", for example, a turn-on sequence could be designed in Matlab and initiated using the above-mentioned `system()` command. After each step an analysis could be automatically spawned to run on a separate Matlab session from within the already-running TCL interpreter, and the results written into the state log. By examining the state log entry, the top-level Matlab session could take conditional action: report success and continue initialization, report failure and retry the last step, or report failure and send an alert.

While the final goal of having the SCDU testbed bring itself up to operational status and begin data collection autonomously seems far-fetched, realization is reasonable. The only hurdles are automating a few initialization procedures, devising reasonable metrics for acceptable state (quantitative approximations of these already exist), and spending the time to write the interface software in Matlab.

10. CONCLUSION

A contributing factor to the continued success of the SCDU testbed is the high level of computerized automation. From streamlining day-to-day operation to facilitating data re-analysis campaigns, the tools help to remove the human element from repetitive tasks and allow attention to be refocused on creative and innovative work. By utilizing an entire network of computers, analyzed data are available in minutes instead of hours, ultimately resulting in increased productivity.

ACKNOWLEDGMENTS

Copyright 2010 California Institute of Technology. Government sponsorship acknowledged.

The research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[1] Shen, T.-P. J., Zhai, C., An, X., Tang, H., Sun, G., and Demers, R. T., "Achievements of picometer performance from interferometer spectral calibration development unit (SCDU)," *Proc. SPIE* **7013** (2008).

[2] Zhai, C., Shao, M., Yu, J., Goullioud, R., An, X., Demers, R. T., Milman, M., Shen, T.-P. J., and Tang, H., "Picometer accuracy white light fringe modeling for SIM PlanetQuest spectral calibration development unit," *Proc. SPIE* **7013** (2008).

[3] Demers, R. T., An, X., Azizi, A., Brack, G., Lay, O., Ryan, D., Shen, J., Sun, G., Tang, H., and Zhai, C., "Spectral calibration at the picometer level on SCDU (spectral calibration development unit)," *Proc. SPIE* **7013**, 70132H–70132H–13 (2008).

[4] Nemati, B., An, X., Goullioud, R., Shao, M., Shen, T.-P. J., Wang, X., Wehmeier, U. J., Weilert, M. A., Werne, T. A., Wu, J. P., and Zhai, C., "SIM interformeter testbed (SCDU) status and recent results," *Proc. SPIE* (2010).

[5] Wang, X., An, X., Goullioud, R., Nemati, B., Shao, M., Shen, T.-P. J., Wehmeier, U. J., Weilert, M. A., Werne, T. A., Wu, J. P., and Zhai, C., "SCDU testbed narrow angle astrometric performance," *Proc. SPIE* (2010).

[6] Zhai, C., An, X., Goullioud, R., Nemati, B., Shao, M., Shen, T.-P. J., Wang, X., Wehmeier, U. J., Weilert, M. A., Werne, T. A., Wu, J. P., and Zhai, C., "SIM-Lite instrument calibration sensitivities and refinements," *Proc. SPIE* (2010).