NPL Report CMSC 43/04

Report to the National Measurement System Directorate, Department of Trade and Industry

Techniques for Validation of Measurement Software – without specialised tools

Robin Barker and Graeme Parkin

March 2004

Techniques for Validation of Measurement Software – without specialised tools

Robin Barker and Graeme Parkin
Centre for Mathematics and Scientific Computing

March 2004

ABSTRACT

When the techniques in the Best Practice Guide on Validation of Software in Measurement Systems are to be applied in practical validation, almost all of the techniques, even for low assurance, require the use of specialist tools. These tools are not available to measurement scientists developing and validating software for experimental measurement systems and the scientists do not have access to the necessary experience in applying the tools. There are some techniques that can be used in the validation of such measurement systems and some readily available general tools that can be used to support these techniques.

We report on the application of the Best Practice Guide on Validation of Software in Measurement Systems to various measurement software projects at low levels of assurance that have raised issues concerning the lack of tools. This covers the general application of the guide to measurement systems developed at NPL and the application of the guide to the specific areas of remote calibration systems and self-calibrating instruments. We describe the techniques and tools that were successfully applied in these validation case studies and make recommendations for techniques and tools for the general application of the guide at low assurance levels.

ISSN 1471-0005

National Physical Laboratory
Queens Road, Teddington, Middlesex, TW11 0LW

Approved on behalf of the Managing Director, NPL
by Dave Rayner, Head of the Centre for Mathematics and Scientific Computing

Contents

1 Introduction

This report aims to supplement the Best Practice Guide on Validation of Software in Measurement Systems ("the Guide") [1] by describing the techniques and general-purpose tools that can be used for software validation when specialised validation tools are not available. When the techniques in the Guide are to be applied in practical validation, almost all of the techniques, even for low assurance, require the use of specialist tools. These tools are not available to measurement scientists developing and validating software for experimental measurement systems and the scientists do not have access to the necessary experience in applying the tools. Lack of specialized tools or lack of experience of such tools may be the case for other measurement system developments, where the development "team" may consist of only one developer-user [2]. Work on case studies to apply the Guide in such environments has shown that there are some validation techniques that can be used and some readily available general tools that can be used to support these techniques.

This report describes the case studies applying the Guide to various measurement software projects at low levels of assurance that have raised issues concerning the lack of tools. The case studies cover the general application of the Guide to measurement systems developed at NPL and the application of the guide to the specific areas of remote calibration systems and self-calibrating instruments. We describe the techniques and tools that were successfully applied in these validation case studies and make recommendations for techniques and tools to facilitate the general application of the guide at low assurance levels. The report also includes some guidelines for coding standards and code review in a number of languages covered by the case studies.

This report is the output of two projects from the current SS*f*M programme (2001-2004): the project on *Maintenance of the Measurement System Validation Best Practice Guide*, and the project on *Validation of Self-Validating Instruments*. There were inputs to this work from two further projects: the project on *Numerical Software Testing* contributed to the numerical testing of remote calibration systems; and the project on *Maintenance of Existing Software Best Practice Guides* contributed to the coding and review guidelines. The report constitutes the deliverables from three milestones all of which were to report on tools and techniques to extend the Guide in the different application areas.

1) *Applying the BPG when no tools are available* (within the project on *Maintenance of the Measurement System Validation Best Practice Guide*) was to report on case studies on the application of the Guide to measurement systems developed at NPL and report on the use of tools and on techniques to be applied when specialist tools were not available.

2) *Remote calibration case study* (within the project on *Maintenance of the Measurement System Validation Best Practice Guide*) was to report on case studies on the application of the Guide to internet-enabled calibration systems developed at NPL and report on tools and techniques that were appropriate.

3) *Techniques for validation of self-validating instruments* (within the project on *Validation of Self-Validating Instruments*) was to report on the tools and techniques that were applicable in a collaborative case study on the validation of a self-validating instrument.

The first aspect of the first of these two projects was on *Maintenance of the best practice guide and training course* and was mainly concerned with the maintenance of the guide applying the safety-related extensions developed in the project on *Validation of safety-critical measurement systems*. The case study on the validation of a self-calibrating instrument itself was the subject of the second project above, and is the subject of a separate report.

2 Requirement for tools for software validation

The application of the Guide to measurement software requires the following three steps:

1. Determination of the integrity or quality of the measurement software to be developed. This is called the Measurement Software Level (MSL) of which there are five levels 0, 1, 2, 3, 4.
2. Selection of the recommended techniques for the determined MSL.
3. Application of the selected recommended techniques with auditable evidence of use for the measurement software.

There are currently twenty-five techniques recommended in the Guide. Case studies in SS*f*M projects, and other examples of applying the Guide to measurement software, have shown that most of the recommended techniques require the use of tools, see Table 1.

Ref. to Guide	Recommended technique	Measurement Software Level				Requirement for tools *
		1	2	3	4	
12.1	Independent audit	**Used at all levels**				**Not required**
12.2	Review of informal specification	**Yes**	**Yes**			**Not required**
12.3	Software inspection of specification		**Yes**	**Yes**		**Not required**
12.4	Mathematical specification	**Yes**	**Yes**	**Yes**	**Yes?**	**Not required**
12.5	Formal specification				**Yes?**	**Required**
12.6	Static analysis		**Yes**	**Yes**	**Yes?**	**Required**
12.6	Boundary value analysis		**Yes**	**Yes**		**Required**
12.7	Defensive programming	**Yes**	**Yes**			**Not required**
12.8	Code review	**Yes**	**Yes**			**Useful**
12.9	Numerical stability		**Yes**	**Yes**	**Yes?**	**Not required**
12.10	Qualification of microprocessor				**Yes?**	**Not required**
12.11	Verification testing			**Yes**	**Yes?**	**Required**
12.12	Statistical testing		**Yes**	**Yes**		**Required**
12.13	Structural testing	**Yes**				**Required**
12.13	Statement testing		**Yes**	**Yes**		**Required**
12.13	Branch testing			**Yes**	**Yes?**	**Required**
12.13	Boundary value testing		**Yes**	**Yes**	**Yes?**	**Required**
12.13	Modified Condition/Decision testing				**Yes?**	**Required**
12.14	Regression testing	**Used at all levels**				**Required**
12.15	Accredited testing		**Yes**			**Required**
12.16	System-level testing	**Yes**	**Yes**			**Required**
12.17	Stress testing		**Yes**	**Yes**		**Required**
12.18	Numerical reference results	**Yes**	**Yes**	**Yes?**	**Yes?**	**Required**
12.19	Back-to-back testing		**Yes**	**Yes**		**Required**
12.20	Source code with executable				**Yes?**	**Required**

Explanation of the "Requirement for tools" column

Required – definitely required.

Useful – not absolutely required but can ease the process.

Not required – not necessarily required, but yet there may be useful tools to support the technique.

Table 1 Tools required for recommended techniques from the Guide

In this context, tools cover the following:

- A tool that a software developer uses to support some aspect of the development of the software; but excluding tools (or parts of tools) directly supporting the writing and compiling of the software (which are obviously always needed).

So, for an Integrated Development Environment (IDE), we are not interested in the parts of it that support compilation and editing.

- A tool not developed by the software developer but usually brought into the software development project either from a supplier or public domain.
- A tool that would require significant development to enable use across several software development projects.

Tools that appear be to be useful for software development and software validation can have limitations that make them impractical. For tools to be useful, there must be:

- The possibility of efficient application of the tool's function to the software as its been developed, and when modifications are being made, e.g., a tool to run suites of tests and collect the results must run on new revisions of the software and the test suites, without reconfiguration;
- Evidence of the use of the tool and a measure of the success of each application;
- Provision for the development of software that would be too large to validate manually, either because it is too time consuming or too error prone (or both).

Using tools is a barrier to software validation: developers (especially those developers of measurement systems who are also the users of the system) often find some problem in using tools, causing them to reject the tools and hence not apply validation techniques, resulting in less assurance of the software.

We list some of the problems with tools and reasons why they are not being used.

1. Difficulty in selecting an appropriate tool (assuming such a tool exists) for a particular function.

   It is very time consuming to assess tools and even then it is easy to get it wrong. Suppliers of tools do not use common terminology to describe their tools (such terminology may not exist). Recently when trying to select a coverage tool for Java it was unclear what the suppliers meant by branch coverage; some included Java's exception handling (*try – catch*) others did not, but this was only discovered during assessment, as it was not documented.

2. Usually each tool only supports one language (or family of languages).

   This becomes difficult when several languages are being used in the software development project. Internet based projects can typically be using up to three languages e.g. PHP, SQL, HTML and maybe JavaScript. Even a typical C development for measurement software can use assembly code or calls to libraries (e.g., DLLs) written in Fortran.

3. Even simple tools require some training to enable efficient and correct use.

   Limitations of the tools need to be understood so that if needed other techniques can be applied.

4. Many tools work on small software but cannot cope with large software.
5. Prohibitive costs of yearly maintenance.
6. Some languages do not have tool support e.g. this is true VBA and older scripting languages.

From the above it's clear that tools, as required by the Guide, are useful; they can be difficult to apply; and in some cases they do not even exist. Where possible the language selected for development should be chosen for its support by tools.

3 Techniques when there are no tools

As can be seen from Table 1: software that is simple and at MSL 1 can be developed with out the use of tool support. At MSL 1 the techniques that require tools or for which it would be useful to have tools are code review, (many types of) testing and numerical reference results. MSL 2 software may in some circumstances not require tool support, e.g., is the software is sufficiently simple. Software at MSL 3 and above requires tool support.

Section 3.1 discusses various approaches to code review and section 4 describes two types of tools that support code review. Section 3.2 discusses a simple approach to testing. Section 3.3 gives a technique which can be used to support code review, testing and specification. Section 3.4 gives another technique to support testing. The emphasis is on what can done with little or no tool support. All mentioned tools are free and in the public domain.

3.1 Code review

The following sections point out various things that could be done under code review, although some people would not define these as code review. The code review that is being suggested here is not only aimed at ensuring maintainability but also is looking for potential bugs. All of the suggested code review techniques can be undertaken without tools but tools can be very helpful. A problem with code review is deciding how much effort to put into it and if there are cost constraints what is most important to do. Unfortunately there are no rules of thumb to answer these questions.

Our experience of code review has been very positive in that it reveals problems, extends the reviewer's knowledge of the software, and enhances people's skills.

3.1.1 Investigate specific issues for the given language

This part of code review is to ensure that the programming languages used have been used in the most effective and maintainable way. Organisations developing software typically have a procedure for undertaking code review, which would list a number of aspects to be checked such as ensuring that error handling is correctly implemented. All programming languages will have areas where mistakes are commonly made, or will have features that are considered bad practice. These can be checked for under code review.

Annex B contains some code review checks and guidelines. Annex C shows some examples of the findings of code reviews, from the remote calibration case studies described in section 5.

3.1.2 Investigate some aspects of the functionality of the code

This part of the code review is to explore how the code has implemented parts of the specification. It is assumed it would be too costly to do this for all of the code. This technique depends obviously on the specification of the software but also on the design including the hardware, e.g., the software interface between two microprocessors, one taking a measurement the other providing the interface to the user. This approach is called software inspection in the Guide.

To use this approach it must be decided on which parts of the code to focus. A brain-storming approach involving developers and customers of the software and hardware has been found to be very effective in determining what to focus on. For measurement systems the focus would tend to be on ensuring that the measurement path from the sensors to the final measurement value, including any uncertainty evaluation if given, was correct. This may involve studying all of this path or only parts of it. This has been shown to be very effective in determining how good the code is.

3.2 Component testing

Component testing is the testing of one or more parts of the software, each of which has a specification and an accessible interface, e.g. a function. In terms of the Guide, this covers many of the varieties of testing techniques. Tools that support component testing refer to the element of the software to be tested as a "unit" and thus the testing supported by these tools is referred to as "Unit testing". However we prefer to use "component testing", for consistency with the Guide.

It is clear that writing tests for components is usually easy, if tedious, when the component is being developed. But writing tests is only part of what is needed for component testing, it is also necessary to have a reliable and repeatable method for running the tests and reporting the results, as the software is developed.

Some simple but effective public domain tools have been developed to support component testing for a wide range of programming languages, see [3] for a list of these. We list some component testing tools; this is based on the Kent Beck's original testing framework paper [4]:

- Fixture – Create a common test fixture i.e. a common test set up for each test.
- Test Case – Create the stimulus for a test case e.g. calls the function with the appropriate parameters.
- Check – Check the response for a test case. The tools support a range of easily applied checks.
- Test Suite – Aggregate Test Cases. The tools support the running of sets of tests.

Component testing is essential for software development and software validation and is essential component of development methodologies such as Extreme Programming [5]. Tools make it easy to implement and, by removing some of the more tedious aspects of testing, tools make repeated testing practical and effective.

3.3 Prototyping

When part of the developed software is complex in terms of its processing or control, one validation technique is to re-implement the software's functionality in another environment, enabling the operation of the two implementations to be compared. This is most effective when the re-implemented software is developed independently of the original developers (e.g., by a separate software review or validation team). This is easiest when the software can be re-implemented in a function-rich environment, e.g. Excel, MatLab, etc.

This should validate the operation of the (developed) target software, and will certainly lead to a greater understanding (on the part of the software review/validation team) of the issues that should have been addressed in the software. The re-implementation can then act as an oracle for tests. This is rather like back-to-back testing, as described in the Guide.

This process of re-implementation, which we have also called prototyping, can be used to support code review, and can also be used to specify the key software element; but is very much like testing!

3.4 Simulation

When part of the developed software has a well-defined interface to another part, it may be possible to test one of those parts in isolation by simulating the other part. This would be useful when it is known to be difficult to find a bug just by testing the two parts as one unit.

Consider an example based on some real software. The software consists of two modules, module A does some complex processing and module B takes measurements from some sensors. Before the measurements take place the sensors have to be parameterised using a feedback process. Module A sends some parameters to module B. Module B uses these to set up some of the sensors and then takes some readings and returns the results to module A. Module A applies checks on these parameters and if they pass these checks then it concludes that the sensors are set up correctly for measurements to take place. If the checks fail, module A does some complex processing and sends the new calculated set up parameters to module B. The process is repeated until the parameters that module B collects pass the checks in module A. The physics underlying the above process guarantees that the sensors will return parameters that pass the checks in module A within a sensible time period. If it is possible to simulate module B, or rather the sensors, in software so that the results are known, then module A can be tested separately from module B. Then when testing with the sensors, any bugs are more likely to be in module B. It turned out, in the real situation, that this was the case. Module A was tested in isolation and no bugs were found. When testing the two modules together, several bugs were found and these were indeed in module B.

4 General tools for software validation

Currently we have come across two tools in the public domain, which are simple to use to support code review. They work on Windows and Unix/Linux platforms. Some IDEs and compilers will include these features.

4.1 Ctags

The ctags (most of the following text is taken from [6]), programs generate an index (or *tag*) file for a variety of language objects found in file(s). This tag file allows these items to be quickly and easily located by a text editor or other utility. A *tag* signifies a language object for which an index entry is available (or, alternatively, the index entry created for that object).

The tag index files are supported by numerous editors, allowing the user to locate the object associated with a name appearing in a source file and jump to the file and line which defines the name. Ctags is capable of generating different kinds of tag file for each of many different languages, currently 33, including many of the well-known scripting languages.

Ctags supports code review in allowing easy jumping from a name in a source file to its definition (maybe in another file) and back again; furthermore it allows this to occur in a chain of jumps. This avoids having to have many files open up on the screen at the same time (or laid out on a table) and then using "Find" to search for the particular object of interest (or worse still, searching by eye).

Our experiences from the case studies described in this report are that ctags is very easy to set up and use, and is very effective in supporting code review.

4.2 Lint

Although the tool described here is for C, similar tools exist for other languages, e.g., Java[7]. Lint is a programming tool (definition taken from Wikipedia, the free encyclopedia [8]) that performs lexical and syntactic portions of the compilation with substantial additional checks, noting when variables have been used before being set, when they were used as a data type other than in their definition, and numerous other programming errors. The name of the program was derived from the notion that it would lead to *cleaner*, more desirable programming, by picking the *lint* (i.e., little bits of fluff) out of the code. Most modern compilers do much of the same analysis, so nowadays compiler warnings provide many of the same features. Lint also does some analysis that compilers typically don't do, such as cross-module consistency checking, and checking that the code will be portable to other compilers. Lint originated from Bell Laboratories for the language C.

A further development on lint is Splint [9], which, quoting from their website, is "a tool for statically checking C programs for security vulnerabilities and coding mistakes. With minimal effort, Splint can be used as a better lint. If additional effort is invested adding annotations to programs, Splint can perform stronger checking than can be done by any standard lint."

Splint is fairly simple to use and will uncover many simple mistakes: unused variables, unused function parameters, unused code etc. Recent use of this tool on some software in a commercial measuring instrument found all these mistakes and many more!

Splint only works on standard C, according to the ISO C99 specification, but it can usually be persuaded to work on non-standard C files (see the Splint manual). Since Splint has over 200 parameters, care must be taken when analysing the results, i.e., a check may be turned off.

5 Validation of remote calibration systems

Two case studies on internet-enabled calibration systems were undertaken as part of the *Remote calibration case study* milestone. Both studies looked at the application of the Guide [1] to internet-enabled calibration systems developed at NPL: the internet Voltage and Resistance (iVR) system, and the internet Optical Time-Domain Reflectometer (iOTDR) system. Both systems were built round the same software base (iCal) that was developed for iVR and extended for iOTDR. The iVR system was developed as part of the NMS Electrical programme, with the iCal software being developed by Adelard. The iOTDR system was an internet-enabled calibration demonstrator project in the SS*f*M programme, developed by SS*f*M software engineers working with the staff from the Photonics programme.

The case studies followed on from the validation and verification work in the development of iCal/iVR, which started with a validation plan based on the Guide. This plan identified a number of techniques that would be applied to the iCal software, and some of these techniques were applied again when iCal was extended for iOTDR.

5.1 Validation and verification plan for iCal/iVR

The production of the validation plan followed the risk assessment methodology in the Guide [1]. The plan was developed using an early edition of the Guide, but in terms of the current edition of the Guide: the criticality of usage was "business critical", the impact of the complexity of control was "moderate" and the complexity of data processing was "simple" (simple non-linear correction); the Measurement Software Level was 2. For MSL2, the Guide recommends the following techniques; we use "[Technique#*n*]" to refer to the software validation technique described in section 12.*n* of [1], see Table 1.

- [Technique#2] Software inspection (e.g., Fagan inspection) – this methodology is not in-place at Adelard or NPL and was not done.
- [Technique#8] Code review – was performed.

- [Technique#13] Structural testing – the applicability of this should be investigated.
- [Technique#16] System-level testing
  - "end-to-end"/black-box testing done as part of the integration and installation of the iVR system;
  - also, check "round trip", comparing numbers displayed on the 4950 (the remote measurement instrument) with the numbers stored in the server database.
- [Technique#5] Mathematical specification – not applicable to the control software, but the numerical correctness of the correction-coefficient algorithms should be tested. It may also be appropriate to build a finite-state machine model of the server/client interaction – could be used to guide testing/validation based on other techniques. Algorithms could be tested using numerical reference tests [Technique#18] and boundary value testing [Technique#13].

The most appropriate technique is code review applied to the server and client software; this was started as part of the iVR project and continued for iOTDR as part of the case study project. Other, more speculative, techniques for validation (e.g., structural testing and techniques based on a mathematical modelling) were investigated for their applicability as part of the case study project. This included testing to validate the fitness for purpose of the implementation languages for measurement applications.

5.2 Remote calibration case study

5.2.1 Code review

The code review of iVR started by examining the flow of the measurement data through the calibration system. This was partly a familiarisation exercise; partly following the Guide [1], which emphasises using the measurement data as a focus for validation; and partly to answer a specific question on numerical formats. The code review was then split between the client and server systems, which communicated over the internet using XMLRPC [10]. The two reviewers concentrated on the separate code units and there was interaction between the reviewers to understand the XMLRPC messages that flowed over the internet. The notes from the code review of iVR and the subsequent review of the iOTDR server software are in Annex C.

The question about numerical formats had a surprising answer: most of the numerical data is stored and transmitted as a decimal string. Although this is not a compact representation for floating-point numbers, it is robust and proof against changes in internal language formats for numbers. The numbers were output as strings by the remote measurement instrument, and were transmitted in the XMLRPC as strings. The server also stored the numbers in the calibration database as strings, and only converted the decimal strings to internal floating-point numbers when it needed to perform numerical calculations – otherwise they remained as strings. This default treatment of floating-point numbers as decimal strings is good for human readability, so aids maintenance and debugging, and avoids problems arising from converting between different internal floating-point representations in different parts of the remote calibration system.

5.2.2 Numerical testing of PHP

In both the iVR and iOTDR systems, the calibration calculations are performed on the server system and transmitted back to the (remote) client system. In both versions of the iCal software, the server system is implemented in PHP: a scripting language very suited to building systems requiring internet and database access, but little used for measurement software. PHP was not previously used at NPL for numerical computing and a natural prejudice existed against the use of scripting language for numerical computing when that was not the purpose for which these languages were designed, or for which they are typically used.

The approach to testing numerical computing in PHP was to test the implementation of some standard computations contained in the iCal/iOTDR server software. This methodology was a short cut: if the tests were successful then we could be have confidence in both the numerics of PHP and the implementation of the standard computation; if the tests were not successful, then we would not know whether the fault lay in the inherent numerics of PHP or the specific implementations.

The iCal/iOTDR server software includes implementation of the two calculations: *sample mean and sample standard deviation*, and *ordinary straight-line regression*. Both these functions have reference data sets produced by the first SS*f*M programme, so these were used to test the PHP implementations. The second SS*f*M programme has produced data set generators for these functions, and as the data set generator for *ordinary straight-line regression* has been revised to reflect changes in understanding in how to test this function, the PHP implementation was also tested against data sets from the revised generator. The details of testing these two calculations are given in Annex D.

5.2.3 Case study findings

The code review of iVR, combined with the various testing that the system has undergone, give high assurance in the operation of the system. The major weakness found in the validation exercise was the XMLRPC library used in the PHP server system: there should be an XMLRPC library provided with PHP and iCal should be changed to use the in-built library.

The review of the iOTDR server software was undertaken while the system was still being developed; in hindsight, this code review has to be seen as part of the development process – there have been too many further changes to the system for the code review to contribute to the validation of the system.

The results of the numerical testing of PHP were that testing against the various data sets showed acceptable performance, validating both the numerics of PHP and the implementation of the two calculations: *sample mean and sample standard deviation*; and *ordinary straight-line regression.* This allows us to have confidence in both the two versions of the iCal system and (to a lesser extent) any future PHP-based remote calibration systems, in their use of PHP for performing numerical calculations on measurement data.

5.3 Summary

The techniques for validation of remote calibration systems apply to many measurement systems, but the techniques must be applied to the two parts of the system and to the link between them. These techniques are described in the Guide [1],

- Code review targeted at the path taken by the measurement data, including the encoding and decoding of the data passed over the internet [Technique#8].
- Testing of the operation of the system as a whole, including direct (out-of-band) comparison of the original data produced by the remote measuring instrument with the data stored in the calibration database [Technique#6].
- Testing of the numerical algorithms and software used in the instrument, by the use of reference data sets to compare calculations against the “true” results [Technique#18].

6 Validation of self-calibrating and self-validating systems

The original project on *Validation of Self-Calibrating Instruments*: was chosen following an earlier study by NMi and others [11], as the hardware and software for self-calibration offers novel aspects for validation. The scope of the project was widened to include self-validating systems following discussions with Oxford University Department of Engineering Science, in particular about SEVA technology [12, 13]. Self-validating systems also offer novel aspects for hardware and software validation, involving a wide range of SS*f*M topics. However, for the case study milestone of this project, we attempted to validate a self-calibrating instrument in a new collaborative project with NMi. The main conclusions of the project relate to the validation of self-calibrating instruments, but we do have something to say about validation of self-validating instruments, and the majority of our findings apply to the validation of all measurement instruments with complex or novel interactions between the hardware and software.

6.1 Self-calibrating case study

The case study on the validation of a self-calibrating instrument was lead by NMi, working in collaboration with NPL. NMi worked on the hardware and NPL work on the software validation. We were supplied with an example of the instrument, and an external calibration set, so we could perform tests on the instrument.

On the software side, we were supplied with the code and some documentation; but we had no access to a compiler for the code, so it was not possible independently to compile and test the code. We also had access to staff of the instrument manufacturer to answer specific questions but, as the code was up to ten years old, it was not always possible to get useful answers. The aim of study was to determine whether the calibration and measurement process in the instrument were sufficiently reliable. The conclusions were that although there were no actual errors in the hardware and software processing identified by the reviewers, the (lack of) quality of the software and the apparent deficiencies in the software development process did not give us sufficient assurance in the instrument processes.

In terms of the Guide, the Measurement Software Level of self-calibrating instruments is high because of the complexity of the flow of the measurement data through the instrument. There are different flows of measurement data depending on whether the data is being used for instrument calibration or for a user measurement. Nevertheless, the software validation proceeded using techniques that did not require tools or that could be done using general-purpose tools — for similar reasons to those outlined in section 2, above. The hardware and software review teams worked together at many stages: there were requests passed back and forth, and there were joint meetings at which we discussed our findings and identified areas for further work. There would be requests for clarification to see if an obscure point in the software (for example) could be resolved by reference to the hardware. Similarly, there were suggestions from the hardware side that some feature was particularly crucial (or easy to get wrong) and so the implementation of that feature in the software should be investigated more thoroughly.

A report on the case study, covering both hardware and software aspects, is being produced by NMi collaborating with NPL. This report will be made publicly available with the consent of the instrument manufacture, and will then be available from the SS*f*M website[1].

6.1.1 Code review

The main technique used by the software validation team was code review. We started from the source files and tried to determine the organisation of the code. We familiarized ourselves with the code by reviewing modules with familiar functionality; e.g., complex arithmetic functions. Even at this stage there were indications of problems: the complex arithmetic module contained redundant code – apparently an error in copy-and-modify, where code was copied from earlier in the module but then not modified. After the initial familiarization, we identified the modules that were responsible for the calibration and measurement procedures and decided that these were the modules to focus on.

The software was in two distinct units for two different processors and built with two separate compilers. Each of the two reviewers concentrated on a different code unit: where the code interacted with the other unit, the two reviewers developed an understanding of the interaction independently and then compared the two pictures. This was typical of how the software code review proceeded, at a number of levels: an understanding of some feature was developed from some part of the code and this was compared with an understanding of the same feature developed from elsewhere: from some other part of the same code unit, from the other code unit, or from an understanding of the instrument hardware and physics. This "challenge/response" approach proceeded in interactions within the software review team and between the software and hardware reviewers: findings were reviewed and compared and new questions were posed to be answered by further review (and other validation techniques). This approach gave a structure to the code review, and ensured that the final aims of the validation were met, avoiding too much unfocussed review of probably irrelevant code.

The code review was able to verify some internal consistency between the two code units in how measurement and calibration was performed. It was also verified that certain crucial element of the processing were implemented as described in the instrument documentation. We were able to draw up detailed descriptions of some of the modelling that was implemented in the code (e.g., interpolation of calibration constants) that were not described in the documentation but were of interest to the hardware review team.

The only tool that was used here was ctags, a language-sensitive editor that could highlight syntax elements and could jump back and forth to the definitions of constants or functions (in a different file), see section 4.1. This is the sort of functionality that is found in many modern editors that can be configured for a given language and, of course, in Integrated Development Environments (IDEs) for a particular language.

[1] `www.npl.co.uk/ssfm/download`

6.1.2 Static analysis

The software review team used a public-domain static analysis tool (Splint, see section 4.2) to investigate some aspects of the software. This was not a specific validation tool, which would be used to determine branch coverage or statement coverage. Splint is intended to be used by developers to check the code for unused or uninitialised variables, code that can't be reached, functions that aren't called, etc. We used Splint on the source code, which took some work because there were some language extensions in the code that would be understood by the particular compiler but were not understood by Splint; it was also necessary to choose the "right" options in Splint to control the (amount of) output.

The Splint output was useful to guide some aspects of the code review, to confirm some of the findings of the code review, and to give an overall picture of the state of the software code. Because of all the options in Splint, and working round the language extensions, it was possible that the Splint output was not completely reliable, so any significant findings had to be confirmed by code review. The output of Splint, in terms of redundancy (variables, code and functions), showed that no similar tools had been used in the software development process.

Indeed, as a software validation tool used in this way, Splint would be far less useful if the code had already been put through such a tool during development, or if the language or compiler supported the production of the appropriate warnings.

6.1.3 Independent re-implementation

A final validation technique that was used in the software review was independent re-implementation of some key calculations. The technique is simply to re-implement the key calculations in new code, probably in a new language/package. The output of re-implemented calculations could then be compared with the original code output. If we had been able to re-compile and re-run individual software modules in isolation, this technique could have been used more widely by providing sample inputs to both implementations and comparing the results.

Instead, the technique was applied to one set of calculations that produced estimates of the uncertainties in the measurement results. There were figures presented in the documentation that did not correspond to the expectations of the hardware team and which they could not reconcile with the software code. The calculations were implemented in an Excel spreadsheet, which allowed the input and output data, together with the calculations, to be viewed in one document. The output did correspond to the figures in the documentation but it was not possible to verify independently that these figures were produced by the original software code.

6.1.4 Findings

None of the techniques revealed any clear errors in the software – although there were some misgivings from NMi that the uncertainty calculations were the right calculations. The code review and static analysis found numerous examples of poor programming: much of the code seemed to be still in a state of development rather than final production code. For instance, there was code and comments that disagreed, functions that weren't called, code that appeared to be there for the purpose of investigating bugs, etc. The results of the static analysis made it clear that no such tools had been used during the development process and reinforced the view that little attempt had been made to check or tidy up the code before putting it into production. The major negative conclusion from the software validation was that there was no proper software development process used in the original production of the code. This one conclusion was sufficient to decrease the overall assurance of the instrument that could be derived from the case study.

6.2 Self-validating (SEVA) instruments

Self-validating instruments can be used to describe a wide-range of instruments but we have concentrated our study on instruments that produce output conforming to the SEVA standard [14]. A SEVA instrument can output a number of different measurements, for each measurement the output includes the best estimate of the value of the measurement, an estimate of the uncertainty of that value, and a qualitative description of the state of the measurement: "clear", "blurred", "dazzled", "blind"; the output will also include a qualitative description of the state of the instrument as a whole.

The novel features of a SEVA instrument are:

- Employing more than one measurement model to achieve a useful (best estimate) measurement value, even if the signal is degraded; and
- The estimation of the measurement uncertainty based on the state of the measurement and the different measurement models.

In terms of the Guide, the Measurement Software Level of self-validating systems is high because of the complexity of the software to handle more than one model of the instrument operation, each with an associated statistical model for determining the uncertainties.

The aim of standardising the SEVA output is to be able to make process control decisions based solely on the SEVA output, without any further knowledge of the instrument that produced the output. For this to work, there will need to be some (third-party) validation that the SEVA output from an instrument that correctly reflects the measurement signal and the state of the instrument. There will be some tension for instrument manufacturers between the dynamic measurement uncertainties appearing in the SEVA output and the measurement uncertainties quoted by the manufacturer for the overall performance of the instrument. In order to validate a SEVA instrument, in particular the uncertainty evaluation output, it will be necessary to review the models, algorithms and code used to produce the output. A comprehensive validation of the operation of a SEVA instrument would require activity in the following areas, all of which are covered by SS*f*M "best practice":

- Validation of the models used in processing the measurement signal, including the process to establish the best estimate of the measurement value. This is covered in SS*f*M BPG4 [15] on modelling and model validation.
- Validation of the statistical model used to calculate the uncertainty associated with the measurement value. The SS*f*M best practice on uncertainty evaluation, and associated statistical modelling, is described in SS*f*M BPG6 [16].
- Validation of the numerical algorithms and software used to "solve" the models and obtain the measurement values and uncertainties. The SS*f*M approach to testing and validation of numerical algorithms and software is based on the use of numerical analysis and reference test data sets. The approach is described in a paper [17] and a number of SS*f*M reports [18-20], and this will be the basis of a good practice guide to be produced in the third SS*f*M programme (2004 – 2007).
- Validation of the software in the measurement instrument, using software engineering techniques, as described in the Guide (SS*f*M BPG1 [1]) and discussed throughout this report.

6.3 Summary

The techniques for validation of self-calibrating and self-validating instruments apply to many complex measurement instruments. These techniques are described in the Guide [1] but there are special features related to their application in self-calibrating and self-validating systems. Again, we use "[Technique#*n*]" to refer to the software validation technique described in section 12.*n* of [1], see Table 1.

- Code review targeted at the path taken by the measurement data, and directed (where possible) by some understanding of the critical aspects of the physical model/process [Technique#8].
- Static analysis of the software: this can be used to guide the code review and to gain confidence in the quality of the software development process [Technique#6].
- Mathematical analysis of the modelling and statistical modelling used in the software [Technique#4].
- Testing of the numerical algorithms and software used in the instrument, by
    - independent re-implementation of critical calculations [Technique#19]; and
    - the use of reference data sets to compare calculations against the "true" results [Technique#18].

A References

1. *Wichmann, B.A., R.M. Barker, and G.I. Parkin. Validation of Software in Measurement Systems. Software Support for Metrology Best Practice Guide No. 1, NPL, March 2004. http://www.npl.co.uk/ssfm/download/bpg.html#ssfmbpg1*
2. *Clements, T., L. Emmet, P. Froome, and S. Guerra. Test and Measurement Software. Software Support for Metrology Best Practice Guide No. 12, NPL, October 2002. http://www.npl.co.uk/ssfm/download/bpg.html#ssfmbpg12*
3. *XProgramming.com. Unit testing: (2004). http://www.xprogramming.com/software.htm*
4. *Beck, K. Simple Smalltalk Testing: With Patterns: (2004). http://www.xprogramming.com/testfram.htm*
5. *Beck, K., Extreme Programming Explained: Embrace Change. 1999: Addison-Wesley Pub Co.*
6. *Hiebert, D.B. Exuberant CTAGS: (2004). http://ctags.sourceforge.net/*
7. *Artho, C. Jlint: (2004). http://artho.com/jlint*
8. *Wikipedia. Wikipedia: The Free Encyclopedia: (2004). http://en.wikipedia.org/wiki/Main_Page*
9. *Evans, D. Splint - Secure Programming Lint: (2004). http://www.splint.org/*
10. *Winer, D. XML-RPC Specification: 1999. http://www.xmlrpc.com/spec*
11. *Rietveld, G., C. van Mullem, C. Oosterman, J. Dessens, F. Torsten, P. Simonson, H. Nilsson, K.-E. Rydler, J. Jacobson, and M. Ohlsson. Artifact Calibration - An evaluation of the Fluke 5700A Series II Calibrator. Report, NMi/PTB/SP, November 1999.*
12. *Henry, M.P., Recent developments in self-validating (SEVA) sensors. Sensor Review, 2001. **21**(1): p. 16-22.*
13. *Henry, M.P., Self-Validating Sensors - towards Standards and Products. Automazione e Strumentazione, 2001: p. 107-115.*
14. *BSI, BSI-7986: 2001 – Specification for data quality metrics for industrial measurement and control systems. 2001.*
15. *Barker, R.M., M.G. Cox, A.B. Forbes, and P.M. Harris. Discrete Modelling and Experimental Data Analysis. Software Support for Metrology Best Practice Guide No. 4, NPL, April 2004. http://www.npl.co.uk/ssfm/download/bpg.html#ssfmbpg4*
16. *Cox, M.G. and P.M. Harris. Uncertainty Evaluation. Software Support for Metrology Best Practice Guide No. 6, NPL, January 2004. http://www.npl.co.uk/ssfm/download/bpg.html#ssfmbpg6*
17. *Cox, M.G. and P.M. Harris, Design and use of reference data sets for testing scientific software. Analytica Chimica Acta, 1999(380): p. 339-351.*
18. *Cook, H.R., M.G. Cox, M.P. Dainton, and P.M. Harris. A methodology for testing spreadsheets and other packages used in metrology. Report to the National Measurement System Policy Unit, Department of Trade and Industry, from the UK Software Support for Metrology Programme. NPL Report CISE 25/99, NPL, September 1999. http://www.npl.co.uk/ssfm/download/#cise25_99*
19. *Cook, H.R., M.G. Cox, M.P. Dainton, and P.M. Harris. Testing spreadsheets and other packages used in metrology: A case study. Report to the National Measurement System Policy Unit, Department of Trade and Industry, from the UK Software Support for Metrology Programme. NPL Report CISE 26/99, NPL, September 1999. http://www.npl.co.uk/ssfm/download/#cise26_99*
20. *Barker, R.M., M.G. Cox, P.M. Harris, and I.M. Smith. Testing Algorithms in Standards and METROS. NPL Report CMSC 18/03, NPL, April 2003. http://www.npl.co.uk/ssfm/download/#cmsc18_03*
21. *Ganssle, J.G. A Guide to Code Inspections., The Ganssle Group, 2001.*

B Guidelines for coding and code review

The sections below contain elements of good coding practice from a number of sources. These lists can be used in different ways: individual programmers can use some or all of the statements as suggestions for their own code; a group of programmers can adopt selected statements as coding standards to be followed within the group; and code reviews can use these statements as a checklist when inspecting code.

B.1 Coding standards/style – for programmers/reviewers

These lists are the result of discussions at a course on "Measurement software design and development" held at NPL in 2003.

B.1.1 Text-based languages

These comments are aimed at text-based languages (i.e., excluding visually laid-out languages such as LabView). They were developed in the context of Visual Basic and C and a few comments are language specific (and are marked as such).

- (VB) use "option explicit"
- (C/C++) check that pointers are valid
- Declare all objects/variables
- Avoid "magic" numbers (e.g. 1.141)
- Think about maintainability
- Check accuracy of inbuilt functions (and constants)
- Coding standards should be seen as encouragement but "be serious". That is: don't require conformance to coding standards when they are not suitable for the size/type of development.
- Comments: explanation but not over-the-top
- Read good programs for style and standards
- Procedures/functions should be well-encapsulated – no extraneous variables
- Re-factor when necessary – but not when not necessary (don't re-factor for the sake of it).
- Use meaningful variable names
- Use longer names for variables/functions with longer scope
- Keep module files and functions short (functions to fit on one printed page)
- Use libraries – increases assurance (and lets others do the work)
- Use standard (de facto) data structures, protocols, file formats
- Avoid goto
- (VB) Use naming conventions for objects/variables: txt, frm, etc.

B.1.2 LabVIEW

These comments apply specifically to visual languages such as LabView.

- Put rationale for code (the "why") in comments
- Defensive programming [Technique#7] – give feedback to user on unexpected conditions
- All wires should be visible
- All long wires should have labels at both ends
- Don't hide front panel items – no hidden data flows (use visible constant)

- Use colour as decoration to indicate substructures (for readability)
- Avoid global variables
- Declare array bounds: performance
- Keep each level on one screen
- Stop button should be easy to hit (clear margin around it)
- Error notification strongly associated with context (helps debugging)
- Use meaningful icons to convey purpose of sub-VIs
- Use versioning of programs and program components
- Arrange program flow *top-left* to *bottom-right* – helps readability

B.2 PHP

These comments arose from general observations from the reviews of PHP in iCal. The comments in the rest of this annex, on general (text-based) languages, apply in most cases to PHP. PHP at NPL is deployed mainly for web applications with databases; this is also true more generally of remote calibration systems. This means the PHP will construct HTML and SQL.

Review of PHP will require:

- Review of the PHP style and coding;
- Review of the HTML style and coding;
- Review of the SQL style and coding.

Coding of PHP will require:

- Special attention to layout of long lines of PHP and HTML
- Build HTML fragments as balanced collections of syntactic elements, and as $head.$body.$tail.
- Use long variable names for variables that are used infrequently but extensively, use short names for variables that are used frequently but only in a localised block of code;
- Long names of global variables (that are used lots throughout the code) should be aliased in local blocks and function calls.

B.3 General

This list is adapted from [21], with permission. The error types are at a slightly higher level of abstraction than the preceding lists and are not language specific – although the examples are from C.

Code Inspection Checklist – Error Types:

- Function size and complexity unreasonable
- Unclear expression of ideas in the code
- Poor encapsulation
- Function prototypes not correctly used
- Data types do not match
- Uninitialised variables at start of function
- Uninitialised variables going into loops
- Poor logic – [code] won't function as needed
- Poor commenting
- Error conditions not caught (e.g., return codes from malloc())

- Switch statement without default case
- Incorrect syntax (such as proper use of “==”/“=”, “&&”/“&”, etc.)
- Non-re-entrant code in dangerous places
- Slow code in an area where speed is important

C Validation of iCal software

This annex records the comments arising from the various code reviews of the iCal software system. The aim to show the kinds of issues that can be found by code review, and thereby suggest what is looked at in reviewing code.

These are the comments made at the time of the review, and no attempt has been made to reflect how the problems raised in the review have been dealt with in the subsequent development of the software. Many of the issues raised by these reviews will have been resolved, or become irrelevant, as the iCal system has developed.

C.1 Review of iCal/iVR server software

The code review of the software written for the project revealed little. An area of concern was integrity of the measurement data: there was potential for different representations of the data in the XMLRPC data, in the PHP code, and in the MySQL queries. The floating point measurements were represented in the XML and in the SQL as strings; but in PHP values can be stored as strings or numbers and strings that look like numbers will be converted to number is they are used in numerical expression. There was a danger that measurement data values in PHP would be converted unnecessarily from strings to numbers and back to strings. The code review showed that the measurement data values were not treated as numbers throughout the PHP code and so the values would remain the string representation of the measurement value as produced by the client.

The software for handling XMLRPC was not written by the project but was third-party open source software. This software library was also reviewed as part of the server software review. This quality of this software left much to be desired: there was little defensive programming to guard against the XMLRPC not having the expected structure and the way the structures were constructed and deconstructed were “flaky” (lacked robustness). But it appeared that the values being passed by iCal through XMLRPC were sufficiently plain that they would not trigger any of flaky behaviour.

C.2 Review of iCal/iVR client software

C.2.1 Background

iCal consists of software to control equipment over the internet.

The server is implemented in PHP, the client in VB 6 and the communication between them uses XMLRPC.

After some analysis of the software it was decided to concentrate on code review.

This document records our current findings of a code review of the client software.

C.2.2 Aim

The code review of the client software concentrated on:

- Measurement data validation: making sure that the software does not modify the measurement values as they are being transferred back to the server.
- Error handling.

WARNING: Review was done on a prototype version of the software so some of the comments will have been dealt with by the final release of the software.

C.2.3 Summary of findings

Overall client software has been designed to be very generic and should be usable with other instruments.

The main area of concern is error handling. This should be dealt with.

C.2.4 Approach taken

The approach taken was paper review of the code and use of the Microsoft Visual Studio.

C.2.5 Loading of software

There were missing controls (JPCWEBCAM.ocx, ADDFLOW3.ocx), which seemed to not allow the display of the main form. This caused some confusion over what some of the controls were. Some access was gained to the main form by looking at Mainform.frm but unfortunately this was misleading in that it replaced some of the controls with other ones!

Comments

On final delivery of the source code, we need to make sure it can be compiled.

C.2.6 Documentation

Documentation which we had access to was:

- A brief summary in file called Documentation.bas.
- A "to do" list in file called ToDo.bas.
- Commentary in the source files.
- A brief overview of the system, given by the project team.

Comments

Currently the documentation is too brief.

Will also need a description of the protocol used for controlling the calibration described (not the XMLRPC). This is both the generic case and its instantiation for the current case (iVR).

C.2.7 Code review

C.2.7.1 Measurement data

The flow of data is as follows:

- Read from the instrument (4950) as a string (`RunCmd`).
- Added to the ListView control as a string (`RunCmd`).
- Added as array of strings to a structure (`BuildFormStruct`) to be send via the XMLRPC (`btnNextClick`, `ExcecuteRPCMethod`).

Comments

Data is read as a string and returned to the server as a string. At no point in the client software does it get converted into a different type.

C.2.7.2 Error handling

- `Form_Load` has no error handling. Would argue that this currently is not necessary.
- `btnReset_Click` has got error handling (`On Error Resume Next`) but not of the usual form (`On Error Goto <Label>`). It only checks for errors at one point. It obviously has place holders for the usual form of error handling (although `GoTo Finish` should really be `Exit Sub`). We have tested this and it does work (removed server and error message came up).
- Lots of subroutines do not have error handling as yet e.g. `btnNext_Click`.
- `btnRunGPIB_Click` does have error handling (again `GoTo Finish` should really be `Exit Sub`) but `Looptimer_Timer` does not. The first time `Looptimer_Timer` executes any

errors (apart from GPIB ones) would be caught by `btnRunGPIB_Click` but after that `Looptimer_Timer` would be executed from the control Timer and these would not be caught; one could argue that the only errors likely are the GPIB ones that are actually caught, but these should really be passed back up rather than ending the program.

- As far as can be seen all GPIB commands are checked for errors. If an error is found this is displayed but the program then ends (`End` in `GPIBCleanUp`): this is not correct functionality. This has been tested by removing the instrument while readings are taking place (and after the first reading is complete).

Comments

Error handling needs to be implemented properly with a clear design in place.

C.2.7.3 Handling of events

`DoEvents,` in `btnRunGPIB_Click,` does not do very much: it should be placed in the loops inside `Looptimer_Timer` so that the GPIB could be stopped. This may not be what is wanted. The Timer control does allow handling of events between calls to `Looptimer_Timer`.

Comments

Some care and consideration needs to be used in the placing of `DoEvents`.

C.2.7.4 Generic

The following are areas, which may stop the generic nature of the client:

- Implement the Header and Footer for the GPIB commands as it was clearly designed for.
- Remove the "*IDN?" dependency.

Comments

It has been clearly designed to be generic and it would be good to make sure this is completed

C.2.7.5 Enhancements

- Need a quit button when running the GPIB commands.
- Would be good to make it clearer what a user does once the GPIB commands are loaded.

Comments

These may already have been dealt with in the next release.

C.2.7.6 Bug?

When looping to execute the GPIB commands it ends with the time delay after the last reading (pointed out by one of the developers). This is because check for number of loops is at the beginning of `Looptimer_Timer`.

Comments

This should be dealt with.

C.2.7.7 Miscellaneous

- Debug statements are still in the program: e.g., `Looptimer_Timer`.
- There are several ways of dealing with GPIB errors (`GpibErr`, `GPIBcleanup` and in line code) which perhaps could all be dealt with by one subroutine.

- It would be good to remove old style VB like `Dev%` etc.

Comments

None of these are that important.

C.2.8 Our understanding of how the code works

The following is only from a clients point of view and only lists what we think is of interest for our analysis:

- On the software being loaded the subroutine `Form_load` is called. This initialises a few things including where the server is. The main form should then appear.
- The user should then do a reset via a button, which calls the subroutine `btnResetClick`. This sends the method "LoginPage" to the server, collects the response and calls `DisplayResult`. `DisplayResult` does the following if requested by server:
  - Gets session and user ids.
  - Asks user to set up camera.
  - Loads GPIB data.
  - Requests any measurement data.
  - Sets up next (and previous) methods. Seems to always do this.
- The user can execute the next step through the subroutine `btnNext_Click` (via the appropriate button). `btnNext_Click` tells the server to execute the next method.

At some point in the above process some GPIB commands would have been loaded, the user would select a device to use and then call the subroutine `btnRunGPIB_Click`. This subroutine does the following:

- Calls the NI GPIB commands `ildev` and `ilclr` in turn.
- If successful calls the subroutine `LoopTimer` at fixed intervals (`loopWait`) for a fixed number times (`numLoops`) using the Timer control. `LoopTimer` is called before the timer is enabled. `LoopTimer` does the following:
  - Executes each GPIB command in turn through the subroutine `RunCmd`. `Runcmd` does the following:
    - Either writes or reads from the device.
    - On reading it adds readings to the ListView control. Note that it assumes all readings start after the command "*IDN?" has been sent.
  - Updates progress bar.

C.3 Review of iCal/iOTDR server software

The iVR system was extended and adapted for the implementation of iOTDR demonstration system. The review of the iCal software continued with a review of this iCal/iOTDR software. The server software had changed considerably, as the calibration process for iOTDR was considerably different from iVR and there was considerably more to be done in the procedures that made up the calibration process.

The review is broken down by the different PHP files and the functions within those files. The review comments include typos, inconsistencies in coding style, departures from the (reviewer's) coding standards, and potential and real bugs in the code.

At this stage in the development, the PHP files did not describe their purpose, and there was no change-log or version control (versions numbers are all 1).

C.3.1 statslib.php

Project should test the mean and standard deviation functions using the existing SS*f*M reference data sets. Review the implementation of the linear regression function against algorithm testing and numerical analysis project recommendations, and test this function with SS*f*M reference data sets. See section D.

The strange functionality of the inverse student-t function, which takes a parameter for probability and then assumes it is 0.95, should be changed. Best would be to remove the parameter and rename the function with "_95", alternatively the parameter should be tested and an exception thrown if the value is not the expected/assumed values.

C.3.2 html.php

PHP has an ".=" operator to concatenate to a variable ($a .= $b is equivalent to $a = $a.$b). It should be used to simplify the various statements accumulating to $functionhtml.

At various places the while loops in the code are indented to the level to the HTML in the code being constructed not to the level of the PHP code constructing the HTML. Need to maintain independent indentation for PHP and HTML, and should aim to respect the HTML indentation in any multi-line constructed strings.

Does this mean it is XHTML that is being constructed? The headers in xmlrpc/xmlrpc.server.php define it to be so. This means that various conventions should be observed in the constructed (X)HTML:

- attributes need values, e.g. select, noshade
- attribute values need quoting (always, even if numbers?): e.g. " value=\"$ODTRport\" "
- standalone elements need to be marked as closed: <hr/>,
- bracketing elements must be closed: …

The function equipment_html is long and impenetrable.

HTML style: <th>…</th> instead of <td>…</td>

Why <form onsubmit="javascript: return false"> in forms – is this necessary? (Indeed, why doesn't it stop the form from being submitted.)

Should test the resulting HTML (or fragments thereof) using W3C compliance tools.

The HTML table row for THERMport has no closing </tr>

Can there be common code to build sections from the MySQL result (or query) – and so reduce length of equipment_html?

Missing <td/><td/> in DSC table row corresponding to port number columns.

The initial <tr> after $functionhtml should be lined up – and the closing </tr> as well.

No need to assign return values to variables if they are only used in an immediately succeeding return statement.

C.3.3 library.php

C.3.3.1 Functions raise_power through Rolling_window

What range of values is expected for raise_power? Why not use the built in pow?

correct says it will be called on the last trace although there is no next trace to use in the calculation – what happens in this case? – should be documented. What are the parameters?

There is an explicit formula for 10^P – why not use raise_power?

When return values are passed by reference (&$foo) it is more intuitive if these variables occur at the beginning , or at the end, of the argument lists.

Funny indentation at the end of Rolling_window and fwrite()s.

C.3.3.2 read_script

No checking of return values of input/output in read_script and throughout – or are there exceptions caught (somewhere else)?

Do we need explicit XML character escape translation – isn't there something in xmlrpc library?

(Are we sure code will cope with overlapping translations? – check definition of strtr looks OK.)

Since all the scripts are in one directory, this could be assumed by read_script and the calls to the function are simplified by not giving the directory name.

C.3.3.3 unpack_client

Why are these values from the client delimited in this adhoc way – why not use XML instead of %% and @@ separators.

Variable names: $ve meaningless; $controlsa…aux too long. Length of names should be proportional to scope (i,e. the area of code in which it is used and have meaning).

Variable references can be used as variables in body – so, use $num_params instead of $n.

Do not need the $key variable in the foreach()s? – so, just foreach ($array as $value).

Where we assume a list has only one element (or we use only one element) should check length of list.

C.3.3.4 unpack_client_trace

This involves repeated calls to unpack_client_single, so repeated exploding of $object – it would be better to use unpack_client.

C.3.3.5 expired

Should "factor" the code with one function mysql_query_array_0, to replace repeated code, and check the length of the array. Also mysql_query_array_calibration. Can also factory expiry_date ?

C.3.3.6 use_colour

Hard coded values (especially, with one value repeated) is not very robust.

How about more robust/extensible:

```
$colour = array("#00FF00", "#CCFFCC");
function use_colour {
    static $index;
    $index = ($index + 1) mod count($colour);
    return $colour[$index]
}
```

C.3.4 xmlrpc.php

Why '@' prefix on include_once and @$obj =& new ? – what errors are we expecting?

The variable $test is assigned but not tested?

Factor all the write to log.txt, as:

```
function write_log ($message)
{ $fp = fopen('log.txt','a'); fwrite($fp, $message), fclose($fp) }
```

Here, and everywhere, results of input/output should be checked – certainly fopen.

Where there are multiple lines to be written, the log messages could be concatenated. Alternatively, reimplement the functions as:

```
function write_log_messages ($messages)
{    $fp = fopen('log.txt','a');
     foreach (get_func_args as $message) { fwrite($fp, $message) }
     fclose($fp)
}
```

To be called as

```
write_log_messages(
      "\nDelta SL table",
      "\n$ref_length[1]  $ref_length[2]  $ref_length[3]
          $ref_length[4]",
      "\n$OTDR_Meas[1]  $OTDR_Meas[2]  $OTDR_Meas[3]
          $OTDR_Meas[4]  $OTDR_Meas[5]",
      "\n$Temp_unc[1]  $Temp_unc[2]  $Temp_unc[3]  $Temp_unc[4]",
      "\n$Wave_unc[1]  $Wave_unc[2]  $Wave_unc[3]  $Wave_unc[4]",
      "\n$comb_unc[4]  $comb_unc[5]",
      "\n$exp_unc[2]  $exp_unc[4]  $exp_unc[5]\n"
)
```

But that's a bit odd anyway (will the lead "\n"s), probably best as one multi-line string.

C.3.5 server.php

C.3.5.1 Login

How about:

```
$passwordcheck = $myrow and ($myrow["password"] == $password);
```

C.3.5.2 SelectService

Do we need: `$ret = ...; return $ret;` ? rather than just `return ...;`

Should introduce some functions to make the xmlrpc struct values, avoid repeating all the wrapping and escaping functions:

- xmlrpc_struct_prev($object,$html,$prev,$next);
- xmlrpc_struct_user($object,$html,$next,$user);
- xmlrpc_struct($object,$html,$prev,$next,$user);
- xmlrpc_struct_session($object,$html,$prev,$next,$user,$sessionid);
- xmlrpc_struct_control($object,$html,$prev,$next,$control,$user,$sessionid);
- xmlrpc_struct_all($object,$html,$prev,$next,$control,$user,$sessionid,$disconnect);
- xmlrpc_struct_disconnect($object,$html,$prev,$next,$user,$sessionid,$disconnect);

Reformat boolean expression to clarity (based on bracketting/precedence).

$functionthml2 is not a nice name: use $functionhtml and replace later $functionhtml and $functionhtml3 by $functionhead and $functiontail.

Inconsistency about use of quotes in array references: $myrow["password"] but $myrow[expiry].

(But different usage in interpolated strings: "$myword[sessionid] …" or "{$myword[sessionid]} …")

Need (escaped) quotes in size = "\$active\" . Is <th> better than <td><b> ?

Factor the blocks of code that do $result = …; $myrow = …; (and test $result?)

Why are the values in xmlrpcvalues wrapped in extra brackets, e.g. ($nextmethod) ?

Why assign to $result if not tested, would be good to test return values, otherwise omit assignment.

C.3.5.3 OTDRactivity()

$activity=**review**: missing <ul> or <ol> round s, and missing s. Could be better as

```
$functionhtml = "<h2> Session details:</h2>
                  <dl>  <dt>id:</dt>  <dd>$sessionid</dd>
                        <dt>expiry:</dt>  <dd>$expiry</dd>
                  </dl>";
```

Why no previousmethod in this `xmlrpcvalue`? Does the iCal server/client software actually use previousmethod: seem to recall that it was meant to allow the user to step back through the process – but it was never implemented and abandoned as a bad idea.

$activity=**wavelength**: "<H2>" – with capital H? Missing <ul> or <ol>.

Inconsistency "And"/"and".

$activity=**distance**: "<H2>" again. "

" !?

Repetitious code between this and $activity=wavelength

$activity=**linearity**: "<H2>", "

" again; code repetition.

$activity=**equipment**: lots of code repetition, check $p< 0

C.3.5.4 ChooseWavelength

Instead of `$Temp = $parameters[1];` and then testing `$Temp=="" and $n<2.` should do

```
if($n < 2) { $Temp = "" } else { $Temp = $parameter[1] }
```

and then just test `$Temp==""`

Should be checking $results of MySQL updates?

C.3.5.5 get_temperature

Add a comment: `//THERM port = 2, by default`

Inconsistent style in creating elements of simplecontrol array, do not need to use $controlvalue.

C.3.5.6 LinearCal

The code to instantiate values in the scripts ($script=strtr($xscript,array('$Foo' => $Foo))) is not very robust, for instance what would happen if there are longer variable with the same start. Suggestions:

- delimit the strings to be translated in the scripts 'Foo'
- we can simply evaluate the script value to have the PHP value interpolated!?

  ```
  script = eval("\"$xscript\"");
  ```

- change the variables in the script to VBscript variables and construct an initialisation block for the start of the script.

Any solution should be part of an extended read_script function, for instance the translation array could be a parameter to read_script and the instantiated script string could be returned.

Also "Lead in" should be hyphenated in log messages (spelt "lead-in" elsewhere).

C.3.5.7 OTDRLin0

Factor out the HTML " Please clean … and try again ", and add at end of if-elseif-else.

Some comment on $GPIBatt = 1 (is this a default?)

Sometimes $myrow is checked, sometimes it isn't. Be consistent or explain inconsistency.

There are two expressions ending +1000 and +3000: the code reads as if they should be the same. Either:

```
$min_range = $fibrelength+$leadin+1000; // or 3000
if( $end_range < $min_range ) { $end_range = $min_range }
```

or explain why the two expressions are different.

C.3.5.8 CalLin

Repeated unpack_client_single should be replaced by unpack_client, as above.

Capital “P” in “<P>”.

C.3.5.9 linearconfirm

Why $yes = “YES” ?

C.3.5.10 linearanalyse

Spelling: “lead in” again.

There is odd indentation of nested if/for block and of calls to fwrite.

This function is very long – there should be more functions to separate out the code, and more functions should be moved to separate library files.

D Testing of PHP software components

D.1 Testing *sample mean* and *sample standard deviation*

The testing targeted mean and stdev functions for iOTDR in the stats library html/php/statslib.php. We wrote a script that loaded the library and read test data from a test data set file loaded from www.eurometros.org/reference_data_sets. The first test used file33 (mean=SD=100) and the answers were obviously wrong (e.g. mean=98.7). The script was extended to read in the reference answers file (.aux) for comparison and to calculate the performance measures. The performance measures were greater than 10, meaning at least 10 figures of accuracy were being lost compared to the reference software.

To try and understand the problem, the test script was converted to Perl, including straightforward implementation of *sample mean* and *sample standard deviation*. This gave performance measures of less than 1, meaning at most 1 figure of accuracy lost. So the problem was not an inherent problem in scripting languages. Inspecting the implementations of mean and stdev, we felt sure there was nothing wrong; then we realised that the code was based on 1-based arrays (data with keys 1 to $n) while the test script put the data in a 0-based array (data with keys 0 to $n-1 – the way PHP arrays are).

We re-implemented the test script but the problem did not go away, so we replaced the functions from statslib.php with our own implementation of *sample mean* and *sample standard deviation*. (mimicking the Perl implementations) and the problem went away. The difference between the two implementations was that the statslib functions took two arguments, the number of data values and the array of data values, and the naive implementations took one argument (a zero-based array of data values) and deduced the number of data values from the length of the array. It was the two argument functions that were failing, and they were failing because the first argument was being passed the wrong value.

We rewrote the test scripts with the right variables passed to the statslib functions, and with an offset to use when populating the array of data values. With an offset of zero, the same bad performance measures were obtained; but with an offset of one, the performance measures were very good (less than one, as had been obtained for Perl).

From this testing, we learnt that test scripts can suffer all the same errors as other software. In particular:

- Don't re-use variables in different parts of the code;
- Make sure the test script uses the interface to the function under test;
- When writing code in one language to mimic code from another language, try to be as faithful as possible.

When tested with the correct script, using the right interface to the statslib library functions, the functions under test were shown to be performing as well as possible. This not only validates the implementation of these functions but goes some way to validating the use of PHP in applications with significant numerical calculation.

Test script for *sample mean* and *sample standard deviation*

The script follows faithfully the document "Procedure for the Calculation of Sample Mean and Sample Standard Deviation" that accompanies the reference data sets on the EUROMETROS website[2]. The steps in the procedure are marked by comments in the code.

```
<?
      // library to test: implementing mean, stdev
      include_once("html/php/statslib.php");

      $test = $_ENV[PHP_TEST_FILE];
      if (!$test) { $test = "file33"; }

      print "\n";
```

[2] www.eurometros.org

```
print "FILE:\n";
printf("%-12s = %16s\n", "test file", $test);

// step 1
$eta = 1;
while( 1.0 + $eta > 1.0 ) { $eta /= 10.0; }
while( !(1.0 + $eta > 1.0) ) { $eta *= 2.0; }
print "PREC:\n";
printf("%-12s = %16e\n", "eta", $eta);

$eta = max($eta, 1e-16);

//step 2
$datp = fopen($test.'.dat', 'r') or die;

$base=1;
  $num = 0;
  while(!feof($datp)) {
    $line = chop(fgets($datp,80));
    if(strlen($line)>0) { $data[$base+$num] = $line; $num++; }
    else { break; }
}
fclose($datp);

if( !($num == 100) ) { die("Expected 100 data values\n"); }

//step 3
$auxp = fopen($test.'.aux', 'r') or die;
$naux = 0;
  while(!feof($auxp)) {
    $line = chop(fgets($auxp,80));
    if(strlen($line)>0) { $aux[$naux] = $line; $naux++; }
    else { break; }
}
fclose($auxp);
if( !($naux == 4) ) { die("Expected 4 aux data values\n"); }
list($mean_ref, $sd_ref, $K_mean, $K_sd) = $aux;

print "REF:\n";
printf("%-12s = %16e\n", "mean ref",$mean_ref);
printf("%-12s = %16e\n", "sd ref",  $sd_ref);
printf("%-12s = %16e\n", "K(mean)", $K_mean);
printf("%-12s = %16e\n", "K(sd)",   $K_sd);

// step 4
$mean = mean($num,$data);
$stdev = stdev($num,$data);

print "TEST:\n";
printf("%-12s = %16d\n", "number",  $num);
printf("%-12s = %16e\n", "mean",    $mean);
printf("%-12s = %16e\n", "sd",      $stdev);

// step 5
$d_mean = abs($mean - $mean_ref) / abs($mean_ref);
$d_sd = abs($stdev - $sd_ref) / abs($sd_ref);

// step 6
$perf_mean = log10(1 + ($d_mean/($K_mean * $eta)));
$perf_sd   = log10(1 + ($d_sd/($K_sd * $eta)));
```

```
        print "PERF:\n";
        printf("%-12s = %16e\n", "diff(mean)",    $d_mean);
        printf("%-12s = %16e\n", "diff(sd)",      $d_sd);
        printf("%-12s = %16e\n", "perf(mean)",    $perf_mean);
        printf("%-12s = %16e\n", "perf(sd)",      $perf_sd);

        print "\n";
?>
```

D.2 Testing *ordinary straight-line regression*

We tested the iOTDR implementation of linear_regression against the old (SS*f*M-1) static dataset and against datasets generated from the (SS*f*M-2) dynamic dataset generator. The PHP script linregall.php reads from a copy of the SS*f*M-1 reference datasets (by default) and produces HTML that details the test results and produces a summary table of the performance measures for each data set.

To generate the dynamic datasets from the command line, we added some new methods to the JAVA class in OrdinaryStraightLineRegression.java to capture the output from the DataGenerator functions. There is also a simple front end to this class in TestData.java that converts from the old parameters for the datasets to the new parameters and prints out the dataset and the extra parameters.

The old datasets are described in terms of b1(intercept), b2(slope), m (number of points), sigma (noise) and x (median of x-values). The new data sets are described by x, y, alpha (angle of line), m, L (range/length of interval of x-values), sigma and S (random seed). We derived y, alpha from b1, b2, x by

$$y = b1 + x * b2; \quad alpha = \arctan b2.$$

The range of x-values (L) and the random seed did not correspond to anything in the old dataset description, so they were simply added to the command line arguments for TestData. So the full set of command line arguments for TestData were b1, b2, m, sigma, x, L, S. (TestData.java contains a bit more detail of this specification.) The input/output for creating the new datasets was done by wrapping TestData in a perl script filetestdata.perl. The first five parameters for TestData were taken from the .aux file of the old datasets and the values for L and S (which do not correspond to anything in the old dataset description) can be set on the command line – with defaults L=5, S=0. The new datasets are written to a different directory and use the suffixes .data and .test. To generate the test results for these datasets uses the same linregall.php script as above.

The result of all this testing is that, especially for smaller intervals, the performance does degrade as the data and the noise get larger. But to make sense of the result you have to know what you are testing and what is the output of the calculation or what is the output used for. For the range of values that will be used in the iOTDR system, the performance was more than adequate and the overall results of the testing show that the implementation of *ordinary straight-line regression* and the numerics of PHP itself are fit for purpose for a primary calibration system.

Test script for *ordinary straight-line regression*

The script follows faithfully the document “Procedure for Straight-Line Linear Regression” that accompanies the reference data sets available from the EUROMETROS website. The steps in the procedure are marked by comments in the code. This script produces an HTML page with all the results for each test set and a summary of the results in a table at the end.

```
<?
        // library to test: implementing leastsquares
        include_once("html/php/statslib.php");

        // functions for l2 norm and distance on R^2
        function sq($b) { return $b*$b; }

        function sumsq($m,$b) {
                $sum = 0;
                for ($i=1; $i<=$m; $i++) { $sum += sq($b[$i]); }
                return $sum;
        }
```

```
      function norm($m,$b) { return sqrt(sumsq($m,$b)); }

      function distance($m,$b1,$b2) {
            $b = array();
            for ($i=1; $i<=$m; $i++) { $b[$i] = $b1[$i] - $b2[$i]; }
            return norm($m, $b);
      }
?>

<?  function precision() {
      // step 1
      $small = 1;
      while( 1.0 + $small > 1.0 ) { $small /= 10.0; }
      $eta_test = $small;
      do { $eta_test += 0.1 * $small; }
            while( !(1.0 + $eta_test > 1.0));
      // $foo= array(0,1,2,3); $bar = array(3,2,1,0);
      // $s=distance(3,$foo,$bar); $t = norm(3,$foo);
      // print "$s - $t\n";
      print "<h3>MACHINE PRECISION</h3>\n";
      printf("<p>%s = %.2e</p>\n", "eta<sub>test</sub>", $eta_test);

      $eta_ref=1e-16;
      return max($eta_ref, $eta_test);
    }

    function test_linreg($eta,$file,&$perf,&$file_ref,$dir,
                         $data_suffix,$test_suffix) {
      $file = substr($file,0,-5);
      print "<hr/>\n";
      print "<p><a name=\"$file\"/>file = $dir/$file</p>\n";

      //step 2
      $datp = fopen($file.'.'.$data_suffix, 'r') or die;

      $base = 1;  // offset for data in array passed to function
        $numx = 0;
        while(!feof($datp)) {
          $line = chop(fgets($datp,80));
          if(strlen($line)>0) {
                list($x,$y) = explode(",",$line);
                $xdata[$base+$numx] = $x;
                $ydata[$base+$numx] = $y;
                $numx++;
          }
          else { break; }
      }

      //step 3
      $auxp = fopen($file.'.'.$test_suffix, 'r') or die;
      $naux = 0;
        while(!feof($auxp)) {
          $line = chop(fgets($auxp,80));
          if(strlen($line)>0) { $aux[$naux] = $line; $naux++; }
          else { break; }
      }
      fclose($auxp);

      $aux_data = 9;
      if( $naux < $aux_data ) {
            die("Expected at least $aux_data aux data values\n");
```

```
}

list($java_ver, $xc, $yc, $alpha, $num, $L, $noise, $S, $K) =
      $aux;
if( !($num + $aux_data == $naux) ) {
      $aux_data += $num;
      die("Expected $aux_data aux data values, read $naux\n");
}
if( !($num == $numx) )
      { die("Expected $num data, read $numx\n"); }
for ($n = 0; $n < $num; $n++) {
      $d_ref[$base+$n] = $aux[$aux_data+$n];
}

$log_noise = floor(0.5-log10($noise));
$logx = floor(log10($xc)+0.5);
if( isset($perf["$num"][$logx][$log_noise]) )
      { die("Repeated data set\n"); }

print "<h3>REFERENCE VALUES</h3>\n";
print "<table>\n";
print "<tr>";
printf("<td>%s</td><td>=</td><td align=\"right\">%s</td>",
      "Java", $java_ver);
print "<td width=\"20%\"> </td>";
printf("<td>%s</td><td>=</td><td align=\"right\">%d</td>",
      "number", $num);
print "</tr>\n";
print "<tr>";
printf("<td>%s</td><td>=</td><td align=\"right\">%f</td>",
      "x<sub>c</sub>", $xc);
print "<td width=\"20%\"> </td>";
printf("<td>%s</td><td>=</td><td align=\"right\">%f</td>",
      "range",    $L);
print "</tr>\n";
print "<tr>";
printf("<td>%s</td><td>=</td><td align=\"right\">%f</td>",
      "y<sub>c</sub>", $yc);
print "<td width=\"20%\"> </td>";
printf("<td>%s</td><td>=</td><td align=\"right\">%f</td>",
      "noise", $noise);
print "</tr>\n";
print "<tr>";
printf("<td>%s</td><td>=</td><td align=\"right\">%f</td>",
      "alpha", $alpha);
print "<td width=\"20%\"> </td>";
printf("<td>%s</td><td>=</td><td align=\"right\">%d</td>",
      "seed", $S);
print "</tr>\n";
print "<tr>";
printf("<td></td><td></td><td align=\"right\"></td>");
print "<td width=\"20%\"> </td>";
printf("<td>%s</td><td>=</td><td align=\"right\">%f</td>",
      "K(<b>d</b>)", $K);
print "</tr>\n";
print "</table>\n";

// step 4
if( !leastsquares($num, $xdata, $ydata,
                  $b2, $u2, $b1, $u1, $df) ) {
      die("leastsquares failed\n");
```

```
      }

      print "<h3>TEST RESULTS</h3>\n<table>\n";

      print "<tr>";
      printf("<td>%s</td><td>(%s)</td><td>=</td>",
                  "<td align=\"right\">%f</td>",
                  "b1", "intercept", $b1);
      print "<td width=\"20%\"> </td>";
      printf("<td>%s</td><td>=</td><td align=\"right\">%e</td>",
                                    "intercept uncertainty", $u1);
      print "</tr>\n";
      print "<tr>";
      printf("<td>%s</td><td>(%s)</td><td>=</td>",
                  "<td align=\"right\">%f</td>",
                  "b2", "slope",    $b2);
      print "<td width=\"20%\"> </td>";
      printf("<td>%s</td><td>=</td><td align=\"right\">%e</td>",
                                    "slope uncertainty",    $u2);
      print "</tr>\n";
      print "<tr>";
      print "<td colspan=\"4\"> </td>";
      print "<td width=\"20%\"> </td>";
      printf("<td>%s</td><td>=</td><td align=\"right\">%d</td>",
                                    "degress of freedom",   $df);
      print "</tr>\n";
      print "</table>\n";

      // step 5
      // print "<h3>RESIDUALS</h3>\n<table>\n";
      // print "<tr><th>d<sub>ref</sub></th><th>d</th></tr>\n";
      for ($n=0; $n<$num; $n++) {
            $d[$base+$n] = $ydata[$base+$n] –
                              ($b1 + $b2 * $xdata[$base+$n]);
      //    printf("<tr><td>%e</td><td>%e</td></tr>\n",
      //          $d_ref[$base+$n], $d[$base+$n]);
      }
      // print "</table>\n";
      $d_d = distance($num,$d,$d_ref) / norm($num,$d_ref);

      // step 6
      $perf_d = log10(1 + ($d_d/($K * $eta)));

      print "<h3>PERFORMANCE MEASURE</h3>\n<table>\n";

      print "<tr>";
      printf("<td>%s</td><td>=</td><td align=\"right\">%e</td>",
            "diff(<b>d</b>)", $d_d);
      print "<td width=\"20%\"> </td>";
      printf("<td>%s</td><td>=</td><td align=\"right\">%f</td>",
            "perf(<b>d</b>)", $perf_d);
      print "</tr>\n";
      print "</table>\n";
      $perf["$num"][$logx][$log_noise] = $perf_d;
      $file_ref["$num"][$logx][$log_noise] = $file;
    }
?>


<html>
<head><title>Testing least-squares linear regression</title>
```

```
<!--
      produced
      by    php -f linregall.php > <OS>_linreg.html

      or    set PHP_TEST_DIR new
            php -f lenregall.php > <OS>_new_linreg.html
-->
</head>
<body>
<h1>Testing least-squares linear regression</h1>
<p>Each section is the results from one test file, with a
<a href="#summary">summary section at the end</a>.</p>
<?
      $dir = getenv("PHP_TEST_DIR");
      if( !$dir ) { $dir = 'new'; }
      $dir = "reference_data_sets/linear_regression/test/$dir";
      chdir($dir) or die;
      $data = glob("*.data");

      // use each test file
      $eta = precision();
      $perf = array();
      $file = array();
      foreach ($data as $f) {
            test_linreg($eta,$f,$perf,$file,$dir,'data','test');
      }
?>

<h1>Performance values</h1>
<ul>
<li>The values in the tables are perf(<b>d</b>), calculated above.
</li>
<li>Each table corresponds to data sets with 5, 50, 100 values.</li>
<li>The columns are data sets with given value of x<sub>c</sub>.</li>
<li>The rows are data sets with given <emph>noise</emph> value.</li>
<li>Each value is a link back to the corresponding test file section.
</li>
</ul>

<?    // table of performance values
      foreach ($perf as $num => $perf_num ) {
          print "<a name=\"summary\"/><p>",
                  "<table border=\"1\" width=\"100%\">\n";
          printf("<tr><th>Number=%d</th>", $num);
          for( $i = 0; $i < 5; $i++ )
                  { printf("<th>x~%d</th>", pow(10,$i)); }
          print "</tr>\n";
          for ($j = 1; $j <= 5; $j++ ) {
            printf("<tr><th>noise~%.2e</th>", pow(10,-$j));
                  for ($i = 0; $i < 5; $i++ ) {
              print("<td align=\"right\">");
              if( isset($perf["$num"][$i][$j]) ) {
                printf("<a href=\"#%s\">%.4f</a>",
                      $file["$num"][$i][$j],
                      $perf["$num"][$i][$j]);
              }
              else { print(" "); }
              print("</td>");
            }
            print "</tr>\n";
          }
```

```
            print "</table></p>\n";
        }
?>
<pre>
<?    phpinfo(INFO_GENERAL);    ?>
</pre>
</body>
</html>
```