Report on current software and practices (task 1)

NMI-3 Workpackage 6 FP7/NMI3-II project number 283883

March 22nd, 2013 - R. Leal and E. Farhi (with input from members of the workpackage)

Abstract

In this report, we have reviewed a selection of data treatment software for neutron scattering experiments. The practices used to develop and maintain the software are also analysed in order to define a set of recommendations to be used in further projects, including the development and evaluation of European prototype software which is the main task (3) of this workpackage . This report fulfils Task 1 of the work-package and aspects of Task 2.

The criteria used for the software review are Deployment / Installation, Usability, Functionality, Maintenance and Expandability. The criteria used for the software practices are related to version control, points of failure, testing, documentation, and code duplication.

A Software for neutron data analysis

The neutron scattering software selected for this analysis was those distributed in the ready to run LiveDVD [http://nmi3.eu/about-nmi3/other-collaborations/data-analysis-standards.html], as well as the qtiKWS SANS package. See Table 1 for the list of software reviewed.

We have chosen to group the scientific software in the following categories :

1. Structure
    1. Powder
    2. Large scale structures (SANS)
    3. Single-crystal
2. Spectroscopy
    1. Time-of-flight
    2. Triple-axis spectrometer
    3. Muon
3. Reflectometry
4. Backscattering
5. Spin-echo

Table 2 classifies the software according to the categories above.

In addition to grouping the reviewed software according to the type of beamlines/science it has been applied to, the software may also be categorized according to whether it has been applied to simulation, reduction or analysis. However, scientists in different areas of science and, even within the same area of science, associate different meanings to such categories. Here is what we mean by these, in relation to software for neutron scattering and muon spectroscopy:

- Simulation: Software which simulates data from e.g. a virtual instrument or a material model (e.g. molecular dynamics).
- Reduction: Software that takes raw data collected on a beamline and removes instrument/detector specific artefacts from the data, including detector efficiencies variations over a detector bank, sample can scattering etc. This category mainly includes mathematical operations and variable changes e.g. from scattering angle to momentum transfer.
- Analysis: Software which takes data collected at a beamline, raw or processed, and infer some physical quantities from the data which are dependent on the sample used for the experiment. This category often implies a fit of a model onto the data set, and integrated quantities.

A.1 Software development status

The Table 3A and table 3B illustrate the overall status of the tested software. A sizeable fraction of the projects is not active any longer, others are merely active for bug-fixes and just a few appear to be actively developed (e.g. Mantid, Sasfit/Sasview, McStas, iFit, LAMP, FullProf suite, Vitess). Mantid is to our knowledge the only one that involves a large team of developers of about fifteen full-time active software engineers and programmers.

A.2 Software OS and Installation

All the software reviewed in this report were tested under Linux Ubuntu 12.04 operating system – the same operating system present on the live CD available on the NMI3 website. According to the

web sites where those applications can be downloaded, all the software can run in the three principal operating systems (Windows, Mac OS X and Linux). However, for Linux and according to our tests, not all binary files can be executable out of the box. Normally due to back compatibility issues, the already compiled software written in C++ or Fortran may have issues with binary shared libraries (i.e. libstdc++ and libgfortran).

This problem is actually present in any modern Linux version. It arises when users try to run a legacy program that was compiled against an old shared library expected to be part of the operating system (in an older Linux version), usually, libstdc++ or libgfortran. This can usually be fixed by recompiling the software from the source (when it is available). Having to compile from source makes the deployment of software harder, particularly for non-expert users.
Another issue, that the inexperienced users may face under Linux and Mac OS X, is the amount of extra libraries required for some programs. Windows packages also suffer from this dependency issue, to a lesser extent, by relying on DLL files which may be system dependent. The only practical solution for developers to overcome this issue is either to reduce the number of dependencies or bundle the software with all the necessary dependencies. The latter may be impracticable due to the size of some external libraries.

A few software distribute the Linux version as RPM or/and DEB packages. Those packages are usually capable of calculating dependencies and fetch transparently the necessary libraries from the internet before installation. However, these packages are often Linux version dependent, and not all versions are supported by the developers.

Bundled software, such as LAMP, Grasp or iFit, are distributed in a single package including all external dependencies. LAMP for example, has a live update feature, which fetches the last version from the internet and updates the program transparently for the users.

A.3 Software programming features

Several programming languages and libraries are present in this study. As expected the majority of the legacy programs are written in procedural languages such as Fortran. Not only in the context of this study, but in general, software that started to be developed in the 70-80s, are mostly Fortran based. Some active software packages are still developed in Fortran (e.g. CrysFML library and FullProf Suite).

Some of the reviewed software is still coded in procedural programming (PP) languages such as C and Fortran. Object Oriented Programming (OOP) can be advantageous over PP as both data and functions are wrapped into clear modular entities (objects). Moreover, properly structured OOP code is easy to maintain and modify as new functionalities can be created (extended) with small differences to existing ones. The benefits of OOP can be summarised as: abstraction, encapsulation, modularity, polymorphism, and inheritance. We should however point out that an equally properly structure PP software can be as effective as its OOP counterpart, and can also be made extendible. It all depends on the developer practices.

A great portion of the reviewed code started to be developed a few decades ago, to facilitate and automate certain simple tasks. Due to the continuous requirements for new features, these programs have grown on the same basis. Although the presence of this legacy code (robust code validated by decades of usage and debugging) does not represent a problem, still developing new functionalities upon this paradigm at present can be seen as unnecessary effort and an increased additional complexity. It may then be much more effective to use the existing legacy software as external commands or libraries within modern environments if it is indeed possible to decompose the legacy software into usable pieces in this way.

There are still ongoing development in procedural languages (e.g. Sasfit, McStas, FullProf Suite, LAMP, Grasp, Vitess), which shows that a procedural based project is not incompatible with a long life-time, when properly structured. The developers of these packages often organise the software in folders to keep the functionalities sorted.

Proprietary frameworks are also present in this report. IDL was a platform of choice in the 90s for scientific development. LAMP and DAVE are two programs that use that language. The Matlab language is also used, as two recent projects are based on this platform (iFit and Grasp). These software can be run without purchasing a Matlab/IDL licence, but further development requires a licence to be purchased. On one hand, rather high inherent costs can prevent some software developers from contributing to the code, but on the other hand the effective development cost can be significantly reduced by using such a high level languages. In this context, LAMP has a community of user-developers both within ILL and at other facilities (e.g. HZB-Berlin and ANSTO Australia). The use of iFit is growing among more experienced users accustomed to the Matlab platform.

Java code is less commonly used in the neutron scattering and muon spectroscopy scientific environment, whereas it is actually one of the most used languages along with JavaScript, Ruby, Python, PHP, C, C++ (source: https://github.com/languages). In this study, only the ISAW platform and the triple-axis instrument simulator vTAS were implemented in Java. The ISAW developers added the Jython scripting language support to facilitate the customisation of ISAW at other laboratories. Jython has the same syntax as python, however it does not provide support for other python packages, such as the popular Numpy or Scipy.

Python tends to be indeed the favourite programming language among scientists for recent software (e.g. GSAS-II, GenX and many others not covered in this report). The simplicity of Python programming allied to scientific packages (e.g. Scipy, Matplotlib, which mimic many features of proprietary packages such as Matlab) has made python scripting very popular. A great effort is being devoted to port scientific packages to Python (RPy for the R Project for Statistical Computing, SymPy for symbolic mathematics, Biopython for biological computation, etc.).

However, Python as an interpreted language usually performs slower than compiled languages (e.g. Fortran, C or C++). It is generally accepted that Python however performs faster for prototyping. Some packages compiled in, usually, C++ and C (e.g. numpy) and wrapped in python, can help improve performance when used to store and manipulate large datasets. In the neutron scattering and muon spectroscopy community, Sasview, PDFfit and Mantid have followed this approach, that is develop the core infrastructure in C++ and implement Python bindings to allow users/scientists to contribute and write their own scripts in Python. In Mantid, some of the components, such as GUIs and algorithms, are indeed written in Python.

The Frida software has been developed in C++ and has recently migrated to the most recent version of the standard of the C++ programming language (C++11). Although this might create some issues with old version of GCC, C++11 introduces new features to facilitate the software development. We believe that the C++/Python combination might be wide spread in the coming years.

Mantid follows an object-oriented design. The Mantid initial design was inspired by the GAUDI (http://proj-gaudi.web.cern.ch/proj-gaudi/) platform at the LHC, Geneva. Mantid relies on a number of dependencies within which Boost, POCO, and OpenCascade. Several “Design Patterns” are implemented (Abstract Factory, Proxy, Command), following the GOF book “Design Patterns: Elements of Reusable Object-Oriented Software”. The Template definition and specialisation (Abstract Factories and Singletons) is also observable within the main components. Even though the

Mantid core design is fixed (Algorithms, Geometry, Workspaces, Data Services), the implementation is continuously evolving after reviews and new use cases requests. Actually, this leads to a complex and rich object hierarchy heavily relying on inheritance. Composition may have been a better option in some places to reduce the object hierarchy rigidity as suggested in the GOF book: "Favor object composition over class inheritance".

These considerations are in fact highly related to computer technology history. In the end, the initial design of a package determines most of the usability and life-time of the project (that is maintenance costs). The use of object-oriented languages can help in this essential step, but a clever procedural language design can be as effective. However, simple solutions, which reduce the role of object classes and therefore dependencies, should always be preferred. To determine a proper design choice, it may be appropriate to code a few prototypes before starting a larger implementation, especially as the core design of a project can hardly be changed during the project life-time (except in the early stages).

A.4 Usability and Graphical User Interfaces

The usability of a software is its *raison d'être*. For neutron and muon facilities, this resides in the scientific content. A rich-featured software which cannot be used immediately by an untrained non-expert user can be seen as a failure, whatever be its internal coding architecture. It is thus an absolute requirement to clearly expose software functionalities in terms of science, with details of the data processing steps in order to convince scientists. One common pitfall found in some projects is the assumption that most users share the same degree of knowledge as the developer for whom *everything is simple*, especially when the development team is too heavily focused on internal programming details. To avoid it, software users must be involved at all stages of the software design and coding.

A great part of the tested software provide a graphical user interface (GUI). Exceptions are Frida, PDFfit, iFit and GSAS. The latter has no GUI, but a graphical user interface (EXPGUI) is available as a separate program. LAMP presents several graphical user interfaces providing normal and expert modes, as well as a command line. Mantid also provides a set of dedicated simplified GUI's, in addition to a more complex user interface, and an underlying framework, which is partly exposed through a Python command line, allowing users to load and process data through a Python console, useful for expert-users. iFit provides a command prompt, as well as a plotting infrastructure, but no muon/neutron scattering dedicated user interface.

Java programs use the Java native Swing library for interface development. The programs built under proprietary software use the native GUI system (Matlab and IDL use Java widgets). Some older interfaces are coded in TK (either through TCL or Perl). The FullProf suite uses a commercial platform called Winteracter. Newer GUIs have been implemented in the Python library wxPython (Sasview, PDFGui, GenX, GSAS-II).

Despite the popularity of Qt in the IT community, only Mantid and QtiKWS feature a GUI based on this library. The Qt toolkit is a cross-platform application framework mainly for graphical user interface. It is natively built in C++ but provides bindings for other languages, including Python. To the authors knowledge, this library is very powerful but has a steep learning curve, making wxPython an attractive alternative for scientific software developers.

Almost all of the tested software possesses plotting facilities. Those based in Python often use matplotlib (GenX, Sasview, GSAS-II). Frida for example uses Gnuplot. Sasfit uses TCL/TK blt tool kit. Java, Matlab and IDL-based software (Grasp, Mfit, iFit) use Java native plotting libraries. MantidPlot and QtiKWS were built as part of QtiPlot and use its integrated library for plotting.

Mantid also links to Vates (a customised version of Paraview) for 3D visualisation. McStas supports a variety of plotters and user interfaces, using e.g. Python matplotlib and chaco, perl/PGPLOT, Matlab, Gnuplot, VRML/X3D. This solution ensures that whatever be the installation configuration, at least one interface/plotter is ensured to be functional.

It is worth noting that LAMP has a server side application running at the http://barns.ill.fr website. It exposes remotely the main functionalities of the software through an HTML interface. To our knowledge, in addition to LAMP, only McStas and vTAS provide a web interface. The Mantid team also starts to consider a rich internet application for the near future. This interface will be more limited than the current one (for security issues, no python scripting interface should be available).

A.5 Data formats

All reviewed software start by importing data sets from neutron and muon facilities. However, as there is no standard format for storing data, each software has implemented loaders for the formats related to the facility where the software is developed, as well as an additional list of formats related to the software functionalities. The most simple data format, which is supported by all software, is the column-based text format.

The increasing adoption of the NeXus format helps but does not solve the sparsity and incoherence of data formats, as it is intrinsically a specification onto an HDF container. There is no guaranty that two NeXus files for similar instruments in different facilities (or even in the same) will have the same structure.

A sensible solution is to make sure that the interpretation of the loaded data set within codes is tolerant enough to adapt to variations around given templates. The NeXus file specification helps by taking a significant step towards self-defining data files.

The number of exported data formats is usually smaller than the imported ones, and software often define their own specific formats which extend the already long list.

B Practices of the software developers

The science we acquire today is a continuation of what has been measured and processed in the last 50 years, with much simpler means in computing than today. Only the size of the data sets, not their signification, has changed as instruments have gained new technologies: we still measure materials structure and dynamics, in real and reciprocal space, time, energy, … Only the methodology and development tools have changed. As opposed to the software built by software engineers, scientific software is simply a means to an end rather than the ultimate aim. Such software is used as a tool to progress in research. Also, as stated by Killcoyne and Boyle [*Comput. Sci. Eng.* **11** (2009) 20] the scientific software is usually very specialised for a particular topic and is rarely extensible or interoperable.

For scientists, requirements are emerging and constantly evolving. It may then be difficult for software engineers to continuously capture requirements and design a robust software solution to be used by scientists. As a consequence, scientific software often lack clear requirements, architecture design and documentation, which is mandatory in any commercial IT product. However, building scientific software based on standards and common enterprise architectures does not guarantee alone a successful outcome. For instance the DANSE project produced valuable software (Sasview, PDFgui) but did not fully complete the initial design goals.

Large scientific software projects typically take too long to develop and suffer from poor adoption. Regardless of the size of a project, producing useful software at an early stage of the project is

important. Large scientific software projects necessarily take longer to develop and will suffer from poor adoption if functionality cannot be released periodically and from an early stage in the project.

Mantid appears to be the closest to an enterprise software solution. The project has gained from the expertise of a specialised scientific software consultancy company both at a project management level and coding level. The coding team is based in the UK and the US and new features are released on a regular basis, according to Agile development principles.

Scientific software of any size can have maintenance issues, which are a consequence of the conceptual design and project size. Unit tests do not reduce the maintenance but only allow to measure its volume.

B.1 Coding and Hosting

Good practices start to emerge. The great majority of the software analysed is hosted by code repositories (SVN, Mercury or GIT protocols) with commit tracking features (see table 3B) .

Few exceptions arise for code that is not freely available: GSAS, vTAS, and part of the Fullprof Suite are not available for download. Some source code, although available for download, can be restricted in use by their licensing scheme (which is mostly GPL-like).

Despite the development of some software on proprietary development frameworks (IDL, MatLab, IGOR and PV-wave), the code is usually available for download. Although the development on such commercial platforms typically implies the payment of licence fees, the learning curve is usually fast and the scientific tools provided are often seen as a great advantage. A way to circumvent the purchase cost is to distribute compiled versions, as for LAMP, Grasp, MFit and iFit. The level of these commercial packages compares with that of NumPy in terms of compactness, and functionalities, thus reducing the amount of coding to achieve a given task, and finally lowering the total maintenance as a result.

Obviously, the computational performance of interpreted codes (Python, Matlab, IDL, ...) is lower than that of lower-level language such as Fortran, C, C++ or to a lesser extent Java. To improve the performance of the interpreted languages, all of them allow wrapping low level libraries/programs. A sensible solution is thus to mix a high level language such as Python, with lower level codes such as Fortran, C and C++ so that performance is improved locally.

In total, there is no obvious coding solution, and any software is a complex equilibrium between coding, maintenance and performance costs. It all depends on the initial design, and the programmers ability to keep it simple yet efficient.

If the main criteria for software production is set to be an open access environment, then the current trend is to prefer a Python user layer, with underlying C/C++ routines for heavier computational tasks. The low-level C/C++ layer can be derived from previous libraries such as the CERN ROOT framework to benefit from decades of development, or implement a new framework, as adopted by the Mantid project.
Commercial solutions offer compact, fully functional infrastructures for the developer and the user, and can be distributed without license requirement. This saves development and maintenance efforts at the cost of license fees.

B.2 Testing

Almost none of the software reviewed possesses a unit test or system test platform, where unit tests refer to for example testing of methods in classes, and system tests check a sequence of steps done by the software to produce an expected result. The Mantid project is currently the only one that fully makes use of a testing suite (Google C++ Testing Framework, including Mock tests). iFit, while not implementing any specialised unit test platform, has a set of test routines. The same is true for GSAS II and McStas. Other software provide example script files that can be viewed as tests.

Mantid possesses two Jenkins (http://jenkins-ci.org/) Continuous Integration Servers (performing builds on Windows 32 & 64 bit, Linux & Mac OS) that perform automated builds of the Mantid Framework, MantidPlot and the install packages following each check-in to the Mantid Github repository. A developer is notified if he or she breaks the build or if tests fails. McStas uses a simpler home-made solution for developers notification, which *de facto* provides the same functionality.

B.3 Documentation

User manuals are often available. Some of them are occasional guides rather than exhaustive step by step guides. This reflects the fact that scientific software developers have traditionally been the (first) users of the software and therefore not needed complete documentation. In addition, documentation is the easiest task to postpone when early delivery of software and functionality puts excessive demands on limited development resources. iFit, developed over the last few years, provides good documentation for both beginners and advanced users, including code documentation. GSAS-II provides good tutorials for less experienced users.

Some of the code is not intuitive and lacks documentation both in the code and technical documentation that describe the source code. Comments in the code are generally sparse when they exist. Some of the informative comments stored when pushing a code change to the source repository are not very informative either. In practice, automatic documentation systems, such as Doxygen (http://doxygen.mantidproject.org/), generate only a technical description of a project, and can not replace a proper human-written documentation with tutorials and examples. It is clear that much has to be done in these areas.

For Mantid there is roughly two sets of documentations. One is for the benefit of the developers, which include Doxygen documentation and wiki pages. The other is aimed at users, which is partially written by developers and partly by users of the software. For example, introductory documentation for using the non-expert Muon interface within MantidPlot is entirely maintained by the Muon scientists. The quality of the user documentation created by the developers, is largely determined by the stakeholders and users of the project, who determine the overall task priory list of the project. Hence the user documentation of the Mantid project has seen improvements based on the demand of it users and is therefore tailored to them at SNS and ISIS. However, recent interest in Mantid outside SNS/ISIS has highlighted some lack of documentation in a number of areas.

Some software (including Mantid and Sasfit, Frida) use auxiliary software to generate browsable code documentation. Although useful to navigate through the code and the class dependency figures (where they exist), if the code is not properly commented, the value of this solution is very limited. For Mantid this kind of documentation is mainly useful for developers and a few expert users.

Mantid appears to be the only software presented here that had a software architecture planned. However the situation to date is rather different from the initial plan and documentation about the

current architecture is missing. The last documentation available about design dates from 2009. McStas also started with a careful architecture design, which has been kept robust since then. The iFit software was also designed with an initial phase during which two prototypes where implemented. The Frida software is derived from previous projects such as IDA.
Generally speaking, many projects, even though not having written architecture principles from the outset, start from well thought-out plans and are flexible enough to evolve with software iterations towards stable architectures.

B.4 Code reuse and duplication

Re-factoring and reusing existing code is a quite general concept nowadays. For the case of the recently developed software, reviewed in this study, two techniques were widely used: 1. the complete recode of old applications in a new programming language and 2. a “facelift” of the user interface and introduction of new features keeping the main core of the application (legacy code) unchanged.

The ”best” software available to date are either based on the legacy source code with new interfaces (e.g. EXPGUI interface for GSAS) or full recode of the old application. In GSAS II, for example, only 5% of the legacy Fortran code was kept. For PDFfit2 the decision was to completely rewrite the old Fortran-77 PDFfit engine in C++, and create python bindings to facilitate the production of specific routines and bindings. The MantidPlot interface of Mantid is built as a fork of the QtiPlot application, and the fit engine in Mantid uses the Gnu Scientific Library (GSL). The iFit package includes a significant portion of contributed code, including BLAS/LAPACK/BOOST, as well as hooks to CrysFML and McStas. Using external libraries is a way a reusing code that is handled to a large extent in high-level languages like IDL and Matlab by the packaging of e.g. graphics and maths libraries.

Our experience with recent software supports the opinion that new software may not perform better than old software with 20 or 30 years of testing and fixing. It may thus be effective to directly integrate old codes rather than re-coding them, especially when development resources are limited, even though it does not look very appealing in terms of software architecture. This approach is limited to the extent to which legacy code has to be used as is, in which case the only improvement for the user may be a GUI and the possibility to create workflows with several such pieces of software. Indeed, given the maturity of many experimental methods, the main gain to be provided by new software may be in terms of user-friendliness for new, non-expert users, workflows and eventually automated data treatment.

Code duplication and replication is evident throughout this review, even within a single project, but also when comparing topically-close software (e.g. for small angle scattering and diffraction). Duplication of features appears to increase with the number of developers involved and project size. Mantid for example uses a tool called PMD/CPD <http://sourceforge.net/projects/pmd> in its integration server to probe for code duplication. The result of running this tool shows a non-negligible amount of duplicated code. Similar tools (e.g. simpler Duplo <http://sourceforge.net/projects/duplo/>) can be employed to evaluate and limit code duplication. All software projects should be reviewed periodically and have the resources to address issues like code duplication.

The Mantid Framework is designed around the concept of an Algorithm which takes input data workspaces and return output ones. This plug-in architecture is very convenient for the extensibility of the platform. A number of algorithms constitute a large parts of code which are, in many cases, directly incorporated in the main class. This makes code re-use between algorithms a difficult task, which is resolved by simply duplicating snippets. A possible option would be to keep the code in

the algorithms minimal and implement its functionality in separate independent packages. The concept of workspace and algorithms was first used in LAMP, and is also central in iFit.

It is no surprise to find common, and thus overlapped, functionalities in different software. These functionalities are often rewritten in different styles and languages. Unfortunately few software make use of the other software knowledge by inclusion. In some cases, a full project or large parts of it can be derived from an existing software (e.g. Sassena at the SNS, an nMoldyn fork). Such a strategy does not lower the maintenance for our developer community, but rather doubles it. One may thus argue that collaboration and contribution to a solid software package would be a better solution.

Our recommendation is that collaboration among groups must be strengthened to avoid code duplication. One could envisage listing all current software functionalities so that new software could directly choose these as libraries. Such a catalogue could list software characteristics like models, algorithms, I/O routines, interface design templates. Then, it would be easier to identify overlapping functionalities, and potentially develop a unified set of libraries, simple and well documented. Such a common infrastructure could inspire from the CERN ROOT framework, once data structures have been agreed on. The FP7 project PaNdata (http://pan-data.eu/PaNdataODI) should produce such a software catalogue.

C Summary of recommendations and Conclusion

We list below a set of items that should be considered when starting and managing a project.

Project infrastructure:

- svn/git repository and test suite
- daily package build with automatic reports on the test suite
- bug tracking system (TRAC), but in practice only developers add and browse tickets
- project email lists for all users and for the developer team
- documentation, both for users (as tutorials), and developers (technical)

Packaging/installation:

- Build packages for Windows, Linux, MacOSX
- Deliver releases regularly
- Installation should be achieved with a single mouse click, dependencies must be installed automatically or included in packages
- Distribute sources under e.g. GPL or EUPL

Project design:

- Always prefer the simplest solution (objects with single responsibility/ few libraries)
- Define a simple and extendible data structure to be kept unchanged during the whole project
- Clearly separate computational level and GUI
- Computational level should be callable as libraries and as external executable commands
- Prefer past projects re-use/wrapping that recoding (at least as 1st implementation)
- Minimize dependencies w.r.t. 3rd party libraries (or fully include them)
- The main criteria for developers should be maintenance (that is future cost)
- The main criteria for users should be usability (script/GUI/documentation)
- Bound total project size (max 100 kLOC/developer)
- Pure coding technologies and IT aspects, including GUI's, should represent only a limited fraction of the project
- Science should represent most of the project content and development effort

- The balance of scientific and technical code or GUI in a software project depends on whether the software aims principally to implement new methods for emerging experimental techniques or to enhance user-friendliness and efficient workflows for non-experts
- A Python/C/C++ coding appears as the current preferable non-commercial solution
- Data storage should prefer NeXus/HDF5

Finally, we quote a number of thoughts discussed in the crystallography community (from the IUCR Computing Commission 2008 report about *Age Concern*, <http://www.iucr.org/__data/assets/pdf_file/0010/10531/iucrcompcomm_oct2008.pdf>)

- Users would prefer to press buttons rather than think about a problem – the Microsoft syndrome.
- Users do not want complicated environments – you have to hide your cleverness from them.
- Users will only read a manual as a last resort – but a full reference manual must exist if only for your own sake.
- Users do not understand the programmers problems – don't expect sympathy when things go wrong.
- Programmers do not understand the users problems – but users probably don't understand them either.
- Programmers relish complexity and compactness – it is a symbol of how clever they are.
- What is clear today will become obscure tomorrow – write code that you will never have to re-visit, but just in case you do, comment it.
- Do not re-invent but do improve the wheel – stand on the shoulders of giants.
- Remember that you are not the only expert in the world – listen carefully to your colleagues, critics and users.
- Ensure your sponsors will let you share the source code – otherwise it will certainly die.
- The internal data structure must be well defined and rigid – ad hoc data definitions lead to duplication and confusion.
- Avoid near-duplication of procedural functionality – spend a little more time on generalising one function.

Going further: Mantid evaluation

Given the findings of this review and that Mantid is the most recent and best-resourced software project in the neutron and muon community, the main task (3) in the software workpackage will evaluate the extent to which Mantid can be the basis of a European-wide, collaborative data treatment effort.