Convergence or divergence of Software Tools

R.E. Ghosh
Institut Laue Langevin, B.P. 156 F-38042 Grenoble, France

Abstract

At the first NoBugs, Grenoble 1996, attendees completed questionnaires summarising their current electronics, systems and software, with modest predictions on the potential changes envisaged over the following two years.

Today the variety of electronics and systems has been reduced by market forces. Intelligent data acquisition and data treatment are still far from being integrated. Sophisticated *ab initio* modelling computation is performed most easily by acquiring systems matching that of the developers; network access and client-server tools make these generally available.

There remains today the major problem of dealing with elderly but well-tested legacy programs, typically running under Unix (now including Linux and Macintosh OS-X), whilst the current generation of scientists, especially occasional users, would prefer a PC-Windows solution. Newer applications have taken to the path of using commercial packages, which offer system independence at a price; this constraint alone is sufficient to limit community activities due to the costs involved.

The NoBugs meeting is a fine forum to review this problem and discuss guidelines for directing software development towards a selection of recommended tools.

Introduction

In addition to providing the instruments for use by visiting scientists, large facilities also support the experiments by offering software to help planning and performing experiments, and analysing data.

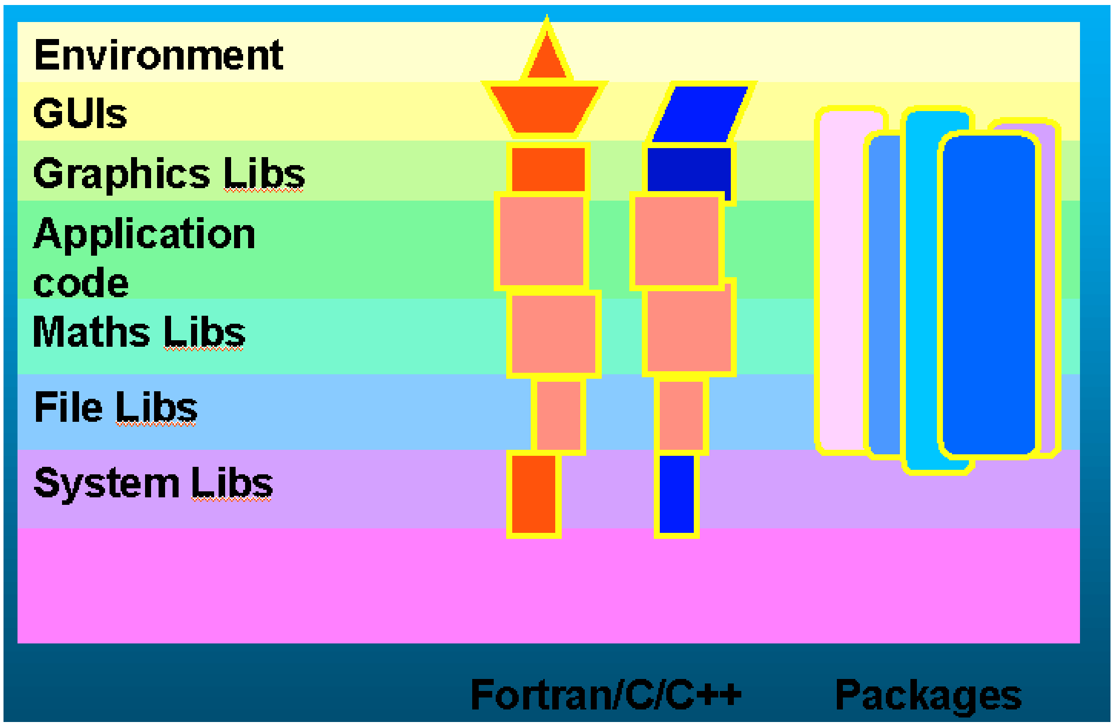
High performance computing allows sophisticated *ab initio* modelling in advance of experiments.. The next goal is to integrate some data analysis into the instrument control to allow optimum use of scheduled experiment time. One rider should be added here: the experiment is then reduced to a sequence of measurements, and when unexpected results are obtained, the data quality may be inadequate to allow for any alternative analysis and interpretation.

Reviewing the range of software tools used in this second phase will be the principal aim of this presentation. It is however useful to note briefly how high-performance software is developed and used. The basic hardware comprises fast processors, memory, and high-speed interconnections. Software is usually compiled using commercial optimising compilers and liked to well established mathematical libraries (Lapack, BLAS routines etc.) The scientific packages are typically the result of many man-years effort; both good physics and mathematical skills are required to make calculations practicable. The result of this effort is shared with the community, which can now afford to acquire similar hardware and use precompiled programs. Commercial software companies, e.g. Accelrys (Cerius, Materials Studio) have seen the value of a market in providing graphical tools for such packages, greatly reducing the setting up time for calculations, and providing tools helping the subsequent analysis. In general the continued development of the underlying calculations remains a specialist group activity.

Interactive Data Treatment and Analysis

In contrast to the long-running, batch oriented, simulations of systems and instruments, instrument control and analysis is expected to offer greater interactivity to the experimenter. Today there are essentially only two types of system. PC-Windows, and Unix/Linux/Macintosh OS-X. For graphics it is possible in all cases to run X-window software, however native implementations of graphics using Windows libraries, or, in the case of Macintosh, the Aqua interface, can offer advantages, especially in simplifying cut and paste procedures for documents. Although once guidelines existed on GUI layout and conventions (e.g. the OSF-Motif Style Guide), today GUI implementations present a jungle of information and tools to the user and absorb much if not most of current scientific programming effort.

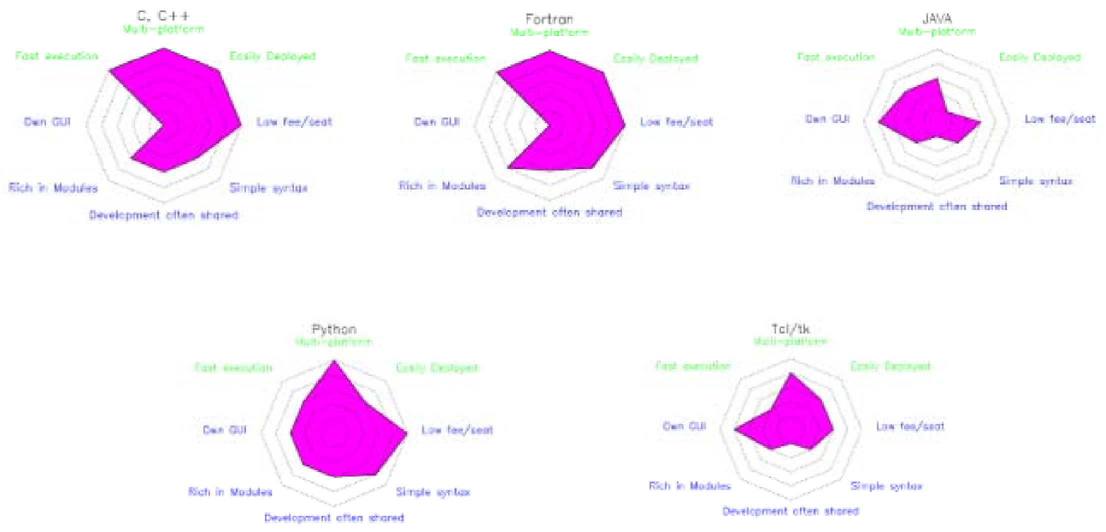
Modules and Interfaces



Clearly defined interfaces and layered functionality simplify the task of the programmer who only has to understand his own application layer. Conventional programming languages may be used in any layer. In the past some order was introduced with the availability of first system, then mathematical and graphical libraries. These showed the value of re-usable code, exemplified by memory resident shareable libraries. Languages such as Fortran, C C++ can inter-operate smoothly and share libraries. The GNU suite of compilers which are available on all platforms demonstrates this by using a common low-level code generator after pre-processing by a language specific layer used to parse the Fortran or C code. There is a wealth of scientific program and library code for these languages.

Most other languages today have been constructed using C and C++. Amongst other Open Source languages are Python and Java, which offer wrappers to re-use Fortran and C routines to improve performance in computationally intensive work. This does require the additional work in implementing variants of an application for each distinct platform. The basic elements of these languages can be learnt rapidly. GUI programming using either PC-Windows or X-window can also be performed using any of these languages. Some version dependence of modules built for Python pose problems of deploying binaries directly. The characteristics of these source-code languages can be compared schematically in the diagram below, from the users' viewpoint in green, and the developers' in blue.

Qualitative comparison of scripted languages

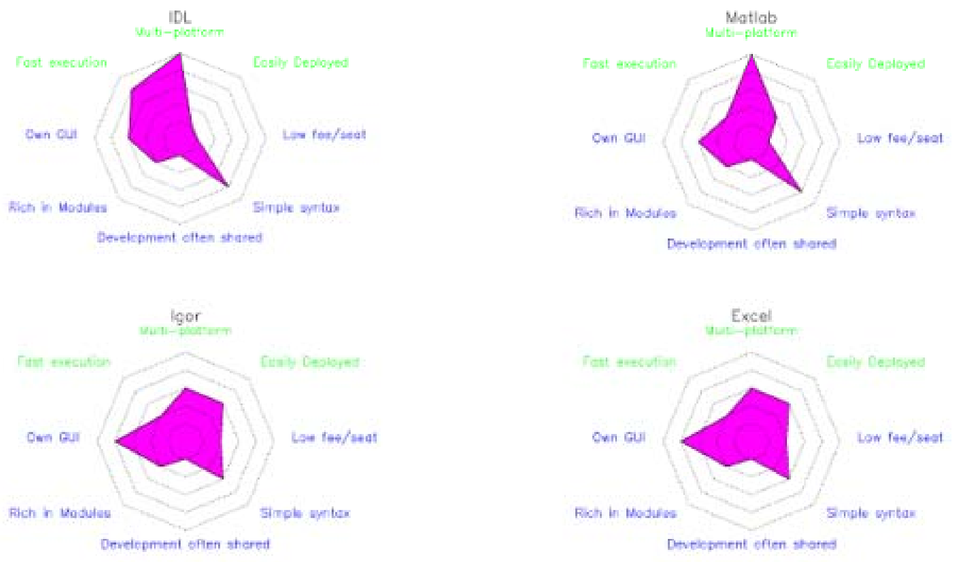


Both Java and Python have common GUI extensions. For comparison C or Fortran may be launched via a controlling Tk/Tcl script which adds an easily programmable GUI level to the conventional program code.

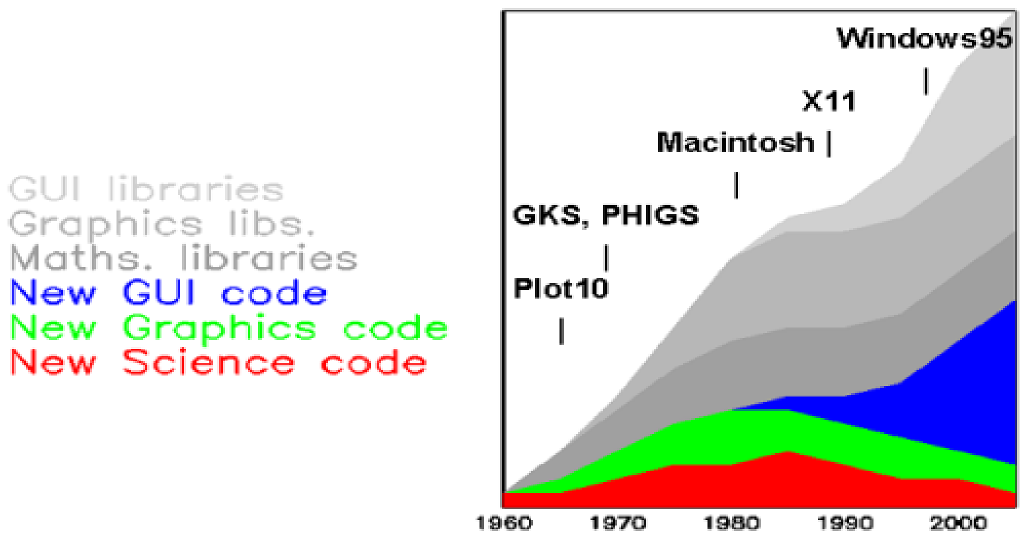
Packages

Integrated graphical and calculation packages became available as workstations evolved especially once the X-window standard was established. These have now been adapted to PC-Windows, and provide a common platform-independent programming interface. Scilab and Octave are in the public domain and use a syntax close to that of Matlab. Scilab includes a large volume of mathematical routines from well established open source libraries. The commercial offerings Matlab, Igor, Origin, IDL, PVwave etc. are scripting languages incorporating GUIs. In some cases, for example IDL, the code is automatically compiled on execution, hence executes rapidly in repetitive calculations. Others depend on internal routines written in C to perform numerically intensive tasks. The performance issue has been sufficiently severe in some cases that, after prototyping a project in Matlab, it has been necessary to rewrite the calculation in C, and integrate this into the Matlab GUI of the prototype for production work. The cost per user seat is an important factor which restricts the building up of a user group where all have the capacity to contribute. The high initial and ongoing maintenance costs limit the number of copies to specific machines bearing the licence; run-time executables can be distributes, sometimes at extra cost, but these remain frozen copies, and cannot benefit from further development. Commercial pressures also lead to incompatibilities between software updates. Wrapper code is used to transfer variables in and out from users' routines written in C or Fortran; this re-introduces platform dependence, and may require using specific commercial compilers.

Qualitative comparison of Packages

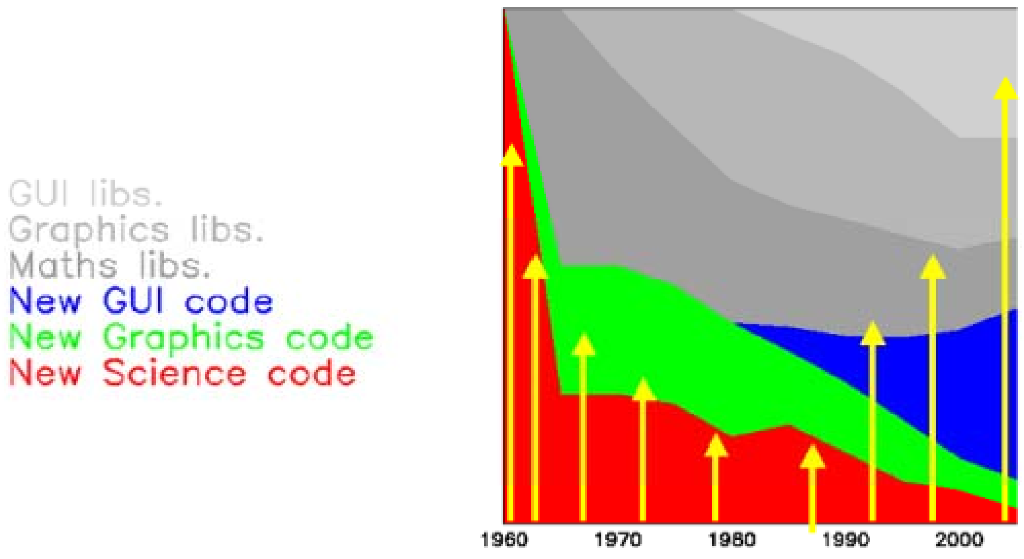


Code Complexity
The early years of Fortran programming lead to the development of spaghetti code which was progressively cleaned when first mathematical routines, and later terminal graphics were introduced as libraries.



GUI programming was well stylised by the excellent Macintosh toolkit which imposed a consistency of design. The multifarious features of X-window and PC-Windows has left the scientific programmer with a complex toolset, though Tcl/Tk remains an option as a bolt-on scripting GUI.
The integrated package traverses all levels of software; this has lead again to a rise of monolithic programming, where much effort is invested in the GUI, and little on new science. Uniform access to all levels, without notions of application layers has lead to spaghetti programming within these monolithic programs which effectively resist sharing future development activities.

The resurgence of Spaghetti code



Federating Experimental Data Treatment and Feedback
These activities used to be performed separately, often using disk files to transfer information. Packages like Labview showed how it was possible to offer a user interface which included a flowpath for data evaluation and possible feedback. The separate packages however remain; users in general do not have access to Labview on their personal computers, and hence duplication of program effort is necessary. The result is a general increase of the number of source code incompatible languages used for data reduction. In general it remains difficult today for an experimenter to include his personal analysis scheme directly into the sequence of measurements.

Shared Data between Control and Treatment
At the ILL, since 1979 data could be treated from the current measurement typically by sharing the data files. These were updated periodically by the control program throughout the measurement. By 1985 this had evolved to sharing data and control parameters which were held in memory on a single system. With no input or output to program the treatment could be substantially simplified. With the addition of workstations at the end of the 80's this concept was extended with simple network daemons updating memory segments in the distributed workstations. Today, despite the advances in network technology there lack easy data interchange methods

between, for example, an Excel spreadsheet and a typical control program, except by file-sharing. Device independent graphics have demonstrated the advantages of layering applications, enabling newer technologies to be introduced painlessly. The client-server model would be an elegant way to federate data movement between intelligent control and treatment. There are not so many packages in general use which would require an appropriate interface routine. Such a routine could be extended to include accessing data in archives on remote servers.

Conclusions

Those still programming with traditional techniques re-use extensively libraries of scientific code and are motivated to adopt modular programming methods.

Packages are proliferating, with no obvious best choice. Inherent limitations often require extensions typically in C or C++.

The community of facility users is expanding and are likely to use tools drawn from an even wider range, leading to less sharing of developments and programs. One way of contributing to minimising this problem is to facilitate data exchange through a standardised interface capable of functioning in a distributed environment.