Robust Symbolic-Numeric-Graphic Software

Kevin A. Broughan
University of Waikato
Hamilton, New Zealand
kab@waikato.ac.nz

Abstract
Issues in the field of intelligent scientific computing are addressed from the point of view of the development of a problem solving environment. These issues include how symbolic, numeric and graphic code should be linked, how a system might be constructed so that it is robust and survives a changing environment, and whether language translators have a role in the retention of the algorithmic content of existing code. The need for a substantial numerical library in Lisp is underlined. Experience has led us to the conclusion that more homogeneity rather than less makes for stronger software.

Introduction
The question is asked: should we attempt to build frameworks that integrate existing tools built with "old" technologies or should we redesign the tools from the bottom up to achieve a new coherent environment?

At the University of Waikato Mathematical Software Project we have written a mathematical modelling system SENAC (Software Environment for Numeric and Algebraic Computation). Senac consists of five modules: Sencore - a symbolic manipulation computer algebra system, Senpack - a Lisp based numerical library, Sengraph - a graphics system targeted at PostScript, Numlink - an interface to the NAG Fortran Library, and Graflink - an interface to the NAG Graphics Library. Underpinning Senpack is a lisp numerical library, Lispack, and a Fortran to Lisp translator, f2cl. Underpinning Sengraph is the graphics system, CLPS.

The experience we have gained building these modules is the subject of this working paper. We started building on old technology but have been progressively pushed into redesigning tools from the bottom up to improve their robustness and compatibility.

Personally, I began looking at codes for the solution of systems of PDE's used in modelling fusion energy containment experiments. These codes were written in Fortran 66 and Fortran 77, had migrated from laboratory to laboratory and person to person and were heavily overlayed with changes and variations as new experiments required different models. It concerned me that these codes were based on a relatively small set of ideas and could have been generated automatically. The virtue of this approach would be that when variations were required, or comparisons with different models sought, a new code could be developed easily. From this notion developed the idea that the entire problem could be formulated and solved within a high-level environment that had symbolic manipulation and numerical features tightly integrated.

Integration or reformation?
We first wrote an interface between Macsyma and the NAG Library so that at least part of this idea could be tested. The interface was complete in that every aspect of calling a NAG routine from Macsyma which could be automated (such as having the interface write the so called user-written subroutines) was automated. This used the foreign function interface structures which gave rise to our first problem: the foreign function interface proved to be one of the most fragile parts of Lisp and many versions of the language (in particular those use to build Macsyma) were not distributed with a working interface.

It is worth noting why this fragility is in a sense natural. In the early Unix days there was a high level of compatibility between C and Fortran. With the demand for industrial strength compilers this compatibility has in many cases disappeared, so Fortran and C based code (Lisp for example) requires tricky tailoring to have the parts work correctly together. Versions which do so one month may stop working together the next. This becomes a crucial issue when a system generates new code at run time which is compiled by the then extant compiler.

These are not "grand ideas" but aspects of reality which spell the difference between survival of programs and their removal from active use. We could include also the problems experienced with dynamic linking. This needs to be provided by the operating system. In a high level environment with a large number of interacting entities the user does not need all of the code po-
using a foreign function interface rather than rewriting the body of tried and tested code found in numeric or symbolic code over pipes or through temporary files. It is surprising how widespread is the use of these primary executable program communicating with the same language. In our case this is Common Lisp. Where appropriate, foreign code should run as an independent executable program communicating with the symbolic code over pipes or through temporary files. It is surprising how widespread is the use of these primitive and restricted forms of communication.

Foreign function interfaces continued to be somewhat less than completely satisfactory, even after the advent of Common Lisp in the mid-eighties. They are a non-standard extension of the standard, lock one into using interfaced code in its given form, and are quite difficult to maintain, especially when C and Fortran architectures and library designs diverge. Increasingly we found we wanted to experiment with lexical scoping, closures, and parallelisms in numeric code. We found that Lisp code could be just as fast as C or Fortran. Once the algorithms are in Lisp then all of this flexibility becomes available. We see this as the path forward, although foreign function interfaces will continue to have a role - especially when foreign code does not require user supplied functions or call-backs to Lisp.

It is worth stressing this advantage of having numerical algorithms in Lisp. Numerical code can call (compiled or interpreted) symbolic code which can call numerical code in any order or depth and in any manner. Provided both symbolic and numeric components have been compiled with the same Lisp compiler they will always work harmoniously together, be they user defined or system defined functions.

All of this experience has led us to the conclusion that, wherever possible, code should be written in the same language. In our case this is Common Lisp. Where appropriate, foreign code should run as an independent executable program communicating with the symbolic code over pipes or through temporary files. It is surprising how widespread is the use of these primitive and restricted forms of communication.

**Fortran to Lisp translation - the new game**

The body of tried and tested code found in numeric subroutine libraries is usually given as the reason for using a foreign function interface rather than rewriting the algorithms. However one should note the current popularity of the translator \$f2c\$. We wrote a translator \$f2cl\$ - this really breaks the back of the translating process. We used an early form of \$f2cl\$ to bring up Numerical Recipes in Lisp and then interfaced that code for Senac. This work is a prelude to the production of a full numeric library in Lisp (Lispack), for which we are gouging interest and seeking support.

I believe this was tried at MIT in the seventies. Lisp compiler technology has improved and the declarations available in Common Lisp imply that now is a much better time for this development. We performed experiments which showed that the compiled Lisp numeric code was not at all slow when declarations were used, but ran at speeds comparable or better than Fortran in environments with plenty of physical memory.

The translator is at present really a rapid prototyping tool. There are probably about 15 bugs to be eradicated and a Fortran standard verifier to be obtained and run as a first pass. The translator handles common blocks but we don't attempt to handle those which are named and then defined in different ways in different subprograms. We don't handle equivalences. In our experience it is only the worst Fortran code which contains these features in any case. We verified the process by translating the very popular set of programs Numerical Recipes into Common Lisp and then interfacing the Lisp code with Senac. Along the way we used Lisp features such as lexical scoping to advantage. Unlike the \$f2c\$ translator, where the goal is to have C code which would run in a Fortran like runtime environment, we want Lisp code which will run in a Lisp environment. Thus no attempt is made to respect the call by reference mechanism of Fortran.

One way to verify the effectiveness of \$f2cl\$ would be to translate a system like LAPACK from Fortran to Lisp. This is one of the ideas we have for the further development of Lispack.

**The use of AI tools**

What AI tools have been found useful in automating phases of the scientific enquiry process? Of course this question leaves open the matter of when something is an AI tool and when it is not. We interpret it to mean the use of a greater degree of automation than normal in scientific computation.

In addition to the Fortran to Lisp translator, language translation and program generation have been used extensively both in the development of the original code and are provided for the user. Numlink was developed using a Lisp based rapid prototyping tool which produced most of each interface routine from the Fortran subroutine header, including the variable declarations. Sencore contains a language translator from Senac internal forms to Fortran Forms. Interface programs contain program generators for creating, at run time, the user written subroutines.
We chose Lisp for software development because of the range of features and data types, the history and current impetus of the language, the evolution of standards, the number of strong vendors and the fact that the language itself was vendor independent. The development of Lisp compiler technology has been strong. Features such as closures, lexical scoping, macros, continuations, the object oriented system, and the condition system become increasingly important and very difficult for single vendor languages to match. We now have had in-depth experience with Franz Lisp, XLisp, Scheme, AKCL, VAX LISP and Allegro CL. Serious users can now write Lisp code, which will run with systems like Senac, knowing that the code has more professional support, canonical language features and compiler support than any language produced by a single vendor.

A deficiency with current Lisp systems: real-life problems are large. Lisp tends to manage its own storage requirements until these are exhausted wherein the system, in general, dies. Building a new and bigger Lisp is the normal exhortation of user manuals. Keeping the size not too large is important because incorrectly structured recursive calls frequently are the cause of resource exhaustion. Automating the growth of the system (i.e. making it dynamic) but catching incorrect recursive calls would be a valuable improvement. Users should not be expected to have to rebuild Lisp. Developers of Lisp based systems should not need to distribute the facilities for rebuilding Lisp and then rebuilding their application.

Two questions related to code generation: How can science users become more comfortable and more trusting of automatically generated code? How can increased confidence be facilitated by the design of code generation systems?

Our approach in the past has been to regard the translation and generation of code as an internal step so that the user does not need to ever view the system produced code. Users trust compilers by considering their results. With f2cl, however, we attempt to make the Lisp code reflect the logic, variable names and sequencing of the Fortran code. Maybe this will instill more confidence in potential users.

A framework for scientific computing

Can we understand the complexity of scientific software, design a better framework for scientific programming, and design software to fit into that framework so that we can integrate new technologies into scientific environments?

A key sub-issue here is that an environment needs to be extensible by a user. Many layers of user extensibility are possible but one primary need is perceived. When a user defines a family of functions these should be able to be compiled. Insufficient speed remains a good reason for why scientists with serious problems refrain from trying higher level techniques.

Real-world problems

How can symbolic computing be integrated with numeric computation in solving real-world complex problems? What advances do we know we need in symbolic algebra systems (e.g. parallel methods)?

Much of the NAG Library is targeted at rather small problems. Take the optimization routines for example. In the development of Numlink we were somewhat led astray by this “smallness” approach and established a matrix data type using Lisp s-expressions as the normal form of input for interfaced routines requiring matrix data. This is inadequate for large problems. Special data types for sparse matrices go some way towards repairing this deficiency but arrays at the Symbolic language level are the natural solution.

An illustration of this idea from linear programming where the number of variables in real-world problems is often truly large. In an integrated symbolic numeric environment the user should be able to introduce each constraint, or indexed family of constraints, using symbolic arrays and symbolic inequalities looking like mathematical expressions. I believe there are PC spreadsheet type programs which allow this. We hope to have verified that this approach is possible and useful in the Senpack setting by the time of the symposium. Based on our experience with hydro system modelling (see below) the difficulty of inputting complex constraints into real world LP type programs is very significant. When the model has a significant number of integer variables then researching effective methods of guiding the search for a solution is a very good AI problem I feel.

I believe one has still to include Linear Programming in the “Grand Challenges” bag. Wherever one goes, problems which are most easily modelled using mixed integer/real linear programming appear and are being used for the solution of serious (economically important) problems. We are building a model for the nearby Waikato River system hydro electric stations. There are 8 stations in all and we aim to model demand over a two day period in one half hour time steps. Solution time is currently 2 hours - this must be reduced to about 20 minutes on “stock” hardware. There are about 1000 integer variables.

Another aspect of the real-world modelling problem is the difference between interactive and batch processing. We would like to be able to set up the environment for solving a particular problem interactively and then send off the problem to be solved on another, larger, processor while still continuing to work in the same or an adjusted environment. The detached process, when complete, should be able to be inspected interactively. This is a simplified aspect of a full multiprocessing setup which would include parallel processing. We are making first steps and hope that others will go the full distance in making this distributed processing practi-
Parallel versions of XLisp and Scheme have been written within the Mathematical Software Project for a family of Inmos transputers. Load balancing, process killing, and expression propagation all offer significant challenges. A more natural environment than the transputers would offer a mixture of coarse and fine grained parallelism and we believe that such networks are now becoming available at reasonable cost.

The parallel XLisp was used to implement a parallel probabilistic greatest common divisor algorithm for multinomials. The parallel Scheme will be used for the factorisation of large integers. This latter arrangement will have different processors computing different but communicating algorithms.

**Symbolic systems do help**

How can symbolic mathematical systems push an applied computing frontier forward, whether in a direction different from the conventional one or further than faster numerical computing could?

Symbolic systems can assist very significantly with the formulation of problems and the processing of the output. Within a symbolic manipulation interpreter the values returned from an evaluation are data types and not just printed expressions. Thus they are available for further processing which could be performed by a program. This is a simple but very significant idea, considering the time that is spent writing code to format output or read formatted data. As well we have the symbolic computation of derivatives and many other well known ways in which symbolic systems can assist.

Experiments with Numlink showed that for some routines (especially those requiring user-written subprograms) the code development time saved using the symbolic language and an interface was more than two orders of magnitude. With more time to spend thinking about a problem, and trying variations, the frontier must go forward faster.

The area of partial differential equations requires special attention. Subroutine libraries have data structures which are not sufficiently flexible to model aspects such as domains at all well. (Fortran 90 might change this however). The discretization of the system could well be performed by a symbolic system. Change of variables is already routinely performed. Ellpack (XEllpack) is an interesting example of high order input and it does use Macsyma. Dealing with the complexity of the input data for these problems and striking the balance between very low level facilities (e.g. the current situation using a subroutine library for the linear algebra and writing the rest of the solution code in C or Fortran) or very high level specialized environments (e.g. Ellpack) are key issues. A positive note - the PDE area will use intensively most of the aspects we are trying to foster: numeric, symbolic, visualisation, applied AI.

Regarding our own work: Specifying domains is in an early state of evolution. We have a set of algorithms for computational geometry (convex hulls, triangulations of non-convex regions with non-convex holes) but this is clearly an area with a lot of potential: especially in three or more dimensions. Again we have found Lisp to be very flexible for algorithm design.

Some library and system programs (optimisation, elliptic PDE solvers, linear programming packages) have numerous tuning parameters with non-trivial relations between these. We have found that the resolution algorithm (written in Lisp) provides a sensible and easy to implement approach to checking these parameters and recognising when they are valid. This applies for instance to an experimental interface to Ellpack for Senac that we have worked on. These parameter sets should be able to be altered interactively, which is very natural of course in a Lisp like environment: one need only make the parameters global variables in appropriate packages or use scoping mechanisms.

**Conclusion**

If we can reach even a partial consensus on the ingredients and nature of a programming development environment for Intelligent Scientific Computation then the field will take a leap forward. Take for example the advent of numerical subroutine libraries and their influence in the seventies and eighties. The programming environment should enable different scientists from different perspectives to make contributions to a larger whole, knowing that the code they write will be vendor independent. The conclusion we have reached will be obvious to the reader - Lisp is the natural environment. The development of a variety of different types of function library in Lisp, to support Intelligent Scientific Computation is, I believe, a way forward with significant potential.

**Acknowledgements**

The author, Terry Robb, Diane Koorey Willcock and Wayne Schou have been the principal authors of the Waikato software. The support given by the University of Waikato Research Committee is gratefully acknowledged.

**References**

