European Society of Computational Methods in Sciences and Engineering (ESCMSE)

Journal of Numerical Analysis, Industrial and Applied Mathematics (JNAIAM)
vol. 3, no. 1-2, 2008, pp. 17-26
ISSN 1790–8140

Embedding a General-Purpose Numerical Library in an Interactive Environment[1]

Mike Dewar

The Numerical Algorithms Group Ltd
Wilkinson House
Jordan Hill Rd
Oxford
OX2 8DR

Received 23 October, 2007; accepted in revised form 26 January, 2008

Abstract:
The NAG Library is a collection of 1533 numerical subroutines, comprising almost a million lines of code which has undergone continued evolution, and been exercised by a large user community for over 36 years. It represents a vast knowledge base, whose re-invention is out of the question. Rather, the question is how to make this functionality available in 21st Century environments. To this end, NAG has developed mechanisms to allow its library to be called from both Maple and MATLAB. This paper addresses the general design, software engineering and documentation issues which arise when trying to integrate a large general-purpose C or Fortran Library such as NAG into an interactive environment.

Keywords: NAG, Maple, MATLAB

Mathematics Subject Classification: 33-04, 34-04, 35-04, 49-04, 62-04, 65-04

1 Introduction

The NAG Library [1] began life in 1971 as a collection of 98 user-callable numerical and statistical subroutines implemented on the ICL 1906A/S. Today the Library has grown to 1533 user-callable routines and is available on 64 different platforms (where a platform is defined as a combination of hardware, operating system, and compiler). It includes routines for solving differential equations, non-linear optimisation, linear algebra, statistics, quadrature, curve and surface fitting, evaluating special functions and much more. The library is available in both Fortran and C versions (Mark 1 was available in Fortran and Algol 60) with special implementations for SMP and cluster architectures.

The NAG Library is popular because the algorithms it contains are accurate and robust, as well as efficient, and because of the wide range of platforms that it runs on. Routines are never withdrawn without notice and replacements, as far as possible, maintain the interfaces of the

[1]Published electronically March 31, 2008

original. This makes it a solid basis on which to develop software which is expected to have a lifetime measured in years or even decades. Users come from disciplines that span many areas of science, engineering and finance, in both industrial and academic settings. However modern users are less keen on programming (at least in the traditional sense) than their predecessors, not least because they have access to excellent interactive tools such as MATLAB [2] and Maple [3] which, combined with modern hardware, allow them to formulate and assemble a solution for problems much more quickly than by writing a Fortran program. Indeed many NAG users actually start out in such packages and develop a prototype of their program there before re-coding it in C or Fortran.

Both MATLAB and Maple provide mechanisms for calling routines in external libraries. Indeed much of the numerical code in Maple has been licenced from NAG and incorporated using a variant of the standard mechanism. However to integrate this code seamlessly, and to provide good-quality documentation, is a very labour-intensive task. For small numbers of routines this is not a problem, but to integrate the whole library — all 1533 routines — in this way would be a huge challenge. Nevertheless there is demand from NAG users to be able to use the Library from within MATLAB, Maple, and similar environments.

This paper describes some of the general lessons learnt from developing two mechanisms for calling NAG routines from interactive packages. In the case of the Maple-NAG Connector the development work was done jointly with Maplesoft, and both companies market the product. In the case of the NAG Toolbox for MATLAB, the work was done entirely within NAG.

1.1 The Maple-NAG Connector

Figure 1 is a screenshot of a Maple session solving the differential equation:

$$y''' = -yy'' - \beta(1 - y'^2) \tag{1}$$

with boundary conditions

$$y(0) = y'(0) = 0,\ y'(10) = 1$$

using the NAG routine d02gac which uses a deferred correction technique and a Newton iteration.

The user need not know that he or she is calling the NAG C Library to solve the problem. The equations themselves are coded up as a Maple procedure, `fcn`, and all the parameters are Maple objects. There is an optional parameter, `comm`, which is used to provide additional data to the `fcn` without using a global variable (in this case the value of the parameter β). The boundary points are `a` and `b`, the boundary conditions are represented by `u` and `v`, and an initial mesh is defined in `x`. The parameter `tol` defines an absolute error for the final solution, which is returned in `y` (for which the user must allocate space). The mesh, `x`, is also updated and its size stored in `np`.

1.2 The NAG Toolbox for MATLAB

Figure 2 is a screenshot of a MATLAB session solving the problem given in equation 1. In this case the differential equation is coded up as a MATLAB function or *M-file*. This time we are using the NAG Fortran Library which does not offer the option to pass the parameter β via the main interface, so it is hard-coded in `fcn`.

A major difference between the MATLAB and Maple interface is that MATLAB does not support parameters whose value is updated on exit, instead the output parameters are returned as an array (in this case `x, y, np` and `ifail`). Optional parameters (not shown here) are provided as name/value pairs. The parameter `ifail` is a status indicator, discussed in section 2.2.

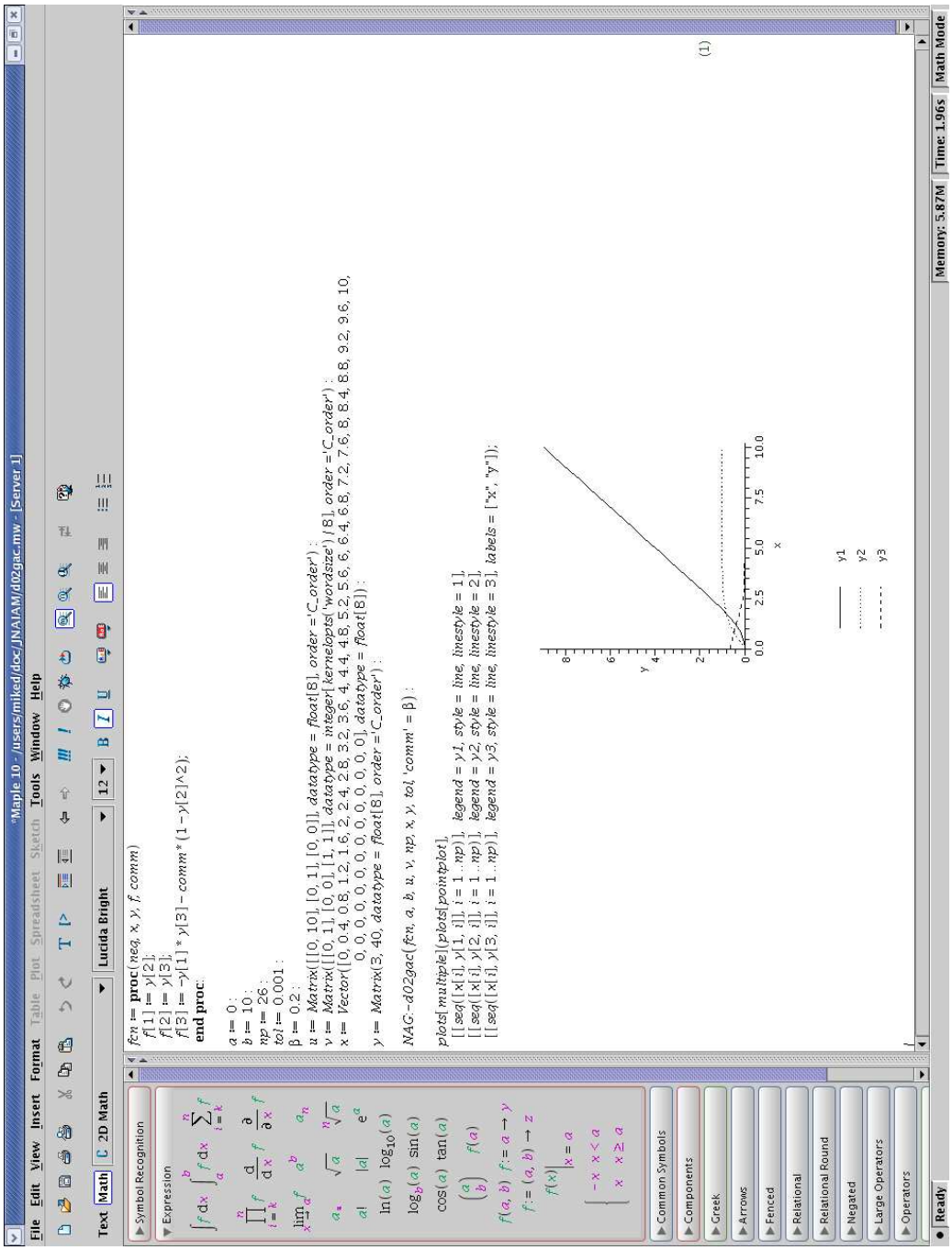

Figure 1: Solving an ODE using the Maple-NAG Connector

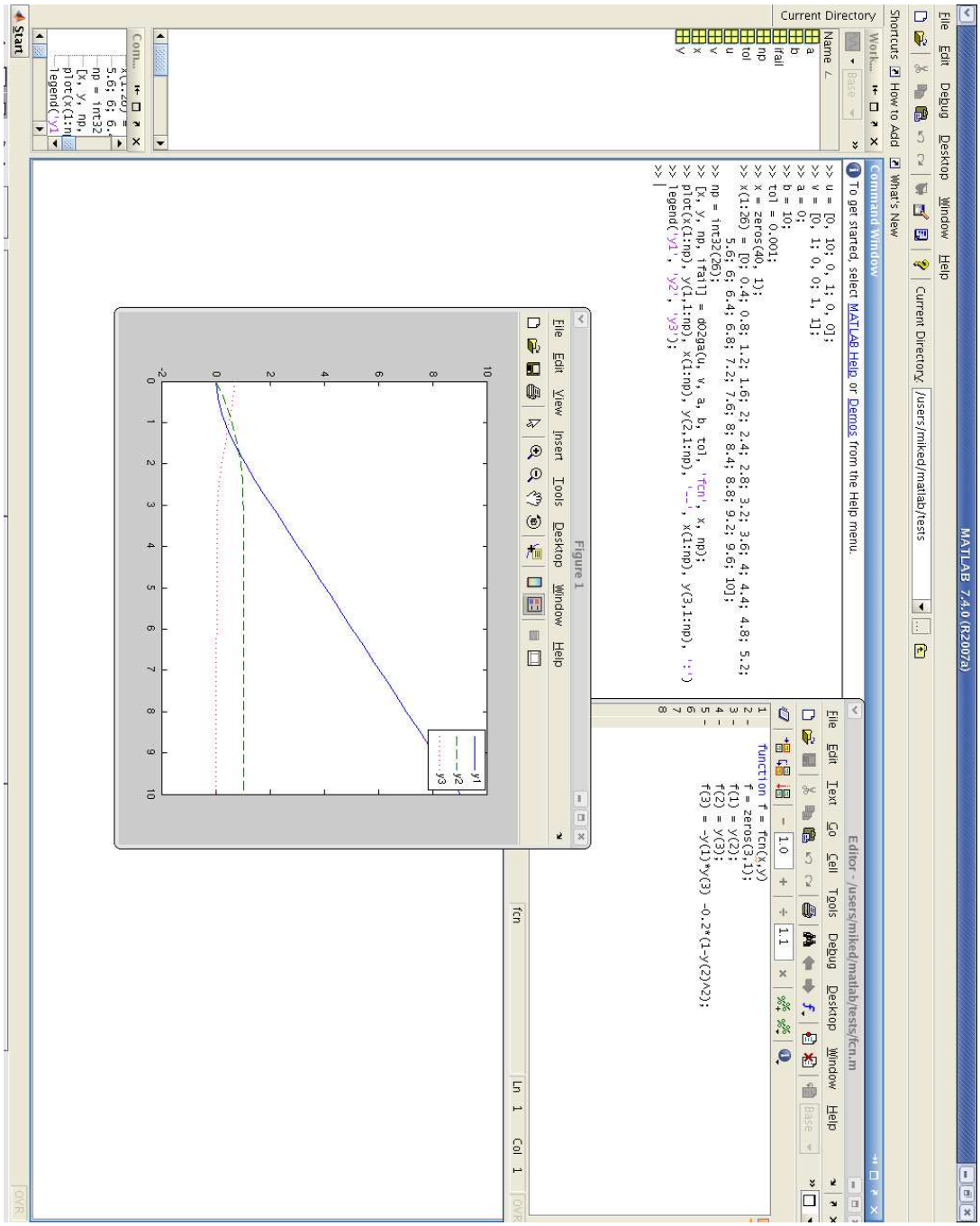

Figure 2: Solving an ODE using the NAG Toolbox for MATLAB

2 User Interfaces

When designing these interfaces we wanted to make them seem as natural as possible in their host environments. However we did not want to move too far from the C/Fortran interfaces for two reasons:

1. we wanted it to be relatively easy to transform a program in Maple or MATLAB into C or Fortran;
2. we wanted to automate the generation of the software components for the interfaces.

For these reasons the differential equation is represented as a procedure in Maple rather than as a symbolic object. Another reason is that it can be evaluated much more efficiently in that form.

2.1 Optional Parameters

We can simplify the Maple/MATLAB interfaces by removing some of the parameters or making them optional. This makes sense when

1. their value can be determined from other parameters;
2. a good default value can be provided covering most situations;
3. they are not used for some kinds of problem.

The most common example of (1) is when a parameter is an array dimension. These have to be provided explicitly in C and Fortran because arrays are just pointers to a piece of memory. In Maple and MATLAB, however, matrices and vectors "know" how big they are. The parameter `comm` in the Maple example in Figure 1 is an example of (3).

In the MATLAB toolbox we also remove all parameters related to workspace, and allocate them automatically. This is not necessary in Maple since the C Library already does this.

2.2 Error Handling

The NAG Libraries handle errors by setting a flag to a non-zero value (in the C Library this behaviour can be over-ridden if the user provides their own error handler). In Maple and MATLAB errors are indicated by throwing exceptions which, by default, cause messages to be displayed to the user. Thus the interfaces need to inspect the NAG error flag and, if necessary, throw an appropriate exception.

2.3 Data Types

In both packages we are very explicit about the types of objects we need to pass to the underlying NAG routine. This is particularly obvious to the user in MATLAB where numbers are, by default, double precision. In the example in Figure 2 for instance, the parameter `np` has to be coerced explicitly to a 32-bit integer. There is a trade-off here between usability and efficiency. Converting scalars to integers at run-time is trivial but converting arrays of doubles to arrays of integers involves additional memory allocation. Moreover, to avoid user errors we would need to step through the array and check that each element could be represented as an integer.

The real efficiency issue, however, arises with *call-backs*, arguments to the routine which are themselves subroutines or functions (in these cases Maple procedures or MATLAB functions). When solving real problems, these can easily be invoked many thousands of times, and doing a type conversion each time would be very inefficient (and could easily be as expensive as doing the computation itself). For this reason we decided to push the onus of type conversion onto the user, and not to hide the issue from him or her.

2.4 Documentation

Both packages come with comprehensive documentation delivered through the host environment's help system. In the case of Maple this offers a rich XML-based environment with good support for displaying mathematical expressions. In MATLAB there is a much more basic HTML environment with little support for mathematical layout (because of this we also offer the help documents as PDF files that can be downloaded from the NAG website). An example of the Maple help for `d02gac` is given in Figure 3.

3 Implementation

Both Maple and MATLAB provide mechanisms that allow a user to link external code into the kernel. In the Maple case this is achieved via the `define_external` command, which takes a specification of the external procedure and generates the necessary software components. Since the resulting interface is quite rudimentary we add an additional wrapper written in the Maple language to implement the interface that we actually want to offer to the user. This wrapper processes optional parameters, handles errors, does data and type conversion where the representation of the Maple object does not match the underlying C representation etc. In the MATLAB case we create a *mex file*: a C routine which serves very much the same purpose as the Maple wrapper, processing optional arguments, handling marshaling and unmarshaling of data etc.

None of these software components is trivial to write. For our example routine the mex file is 335 lines of C, and the size of the Maple components are 120 lines altogether. It should be noted that while d02ga is not the simplest routine in the Library, it is by no means the most complicated. The mex file for d03pp, a routine for integrating a system of parabolic PDEs, is over 1300 lines long. The total size of the mex (source) files for mark 21 of the NAG Fortran Library is over 360,000 lines of code. The equivalent figure for Maple is 90,000 lines. This is partly due to the fact that Maple uses the 895 routines from the NAG C Library while MATLAB uses 1511 routines from the NAG Fortran Library, but also because Maple offers a much higher level external interface which automatically generates much of the low-level C code required.

Clearly writing these interfaces by hand would be a huge task, especially when documentation and testing is taken into account. We therefore chose to automate the process. For each routine in the NAG Library we maintain a specification of its interface in XML. These specifications have evolved over several years and are used in a number of development projects within the Company. They started life as the original documentation sources of the routines which were in SGML, however as well as text describing how the routines are used they contain structured descriptions of each parameter. For example, the description of the parameter `x` in Figures 1 and 2 is

```
<paramhead xid="X" type="rarray" class="IO" purpose="data">
  <dim1><arg>MNP</arg></dim1>
</paramhead>
```

This indicates that `X` is a real vector of length `MNP` whose intent is input and output, and which contains some of the data used to describe the problem (in this case the mesh). A description of how the parameter is used, written in English but with mathematical expressions marked up in a variant of Content MathML [4], is also included.

The interfaces are generated according to a set of rules based on this markup. For example the `purpose` attribute defines what the function of the parameter is; if it is `data` then the parameter forms part of the specification of the problem and is compulsory, but if it is `leading-dimension` then it is the principal dimension (or stride) of an array and, provided that array is provided by the user on input, can be omitted from the Maple and MATLAB interfaces. A parameter such as `MNP` which is the dimension of an input array can usually be inferred from another parameter but,

Figure 3: Maple help for d02gac

since in this case there are sometimes circumstances where one might wish to make `MNP` smaller than the size of the array, it is treated as optional. This allows the user to override the default, namely the length of `X`.

The advantage of this rule-based approach is that it leads to a great degree of consistency in the interfaces provided. Special markup is supported to allow a developer to override the automatically generated interfaces, but it is rarely used. Authors of new material for the NAG Library are required to add the necessary markup as part of the documentation process.

3.1 Robustness

A common programming error when using a C or Fortran Library is for code to read or write beyond the end of an array (often referred to as a *buffer overflow*). Since in both languages arrays are simply pointers to memory, it is difficult for either the programmer or the compiler to insert runtime tests for such events. In the Maple and MATLAB environments, such an occurrence in our interface code would corrupt memory and at best cause an immediate error. We say "at best" because a common behaviour that we have observed in both packages is that errors are not signalled when the memory is corrupted, but rather occur later on when the memory is next used.

By inferring the values of parameters which represent array dimensions we can avoid most of these problems, but there are cases when they could still occur. A trivial example would be for the optional parameter `MNP`, described above, to be set to a value larger than the length of `X`. More realistic cases arise where the bounds of an array are functions of several input variables.

To avoid this we need to check the sizes of all arrays before calling the NAG routine. In this case if a MATLAB user tried to set `MNP` to a value greater then the length of `X` they would see the following error:

```
>> x = zeros(26, 1);
>> x(1:26) = [0; 0.4; 0.8; 1.2; 1.6; 2; 2.4; 2.8; 3.2; 3.6; 4; 4.4; 4.8; 5.2;
              5.6; 6; 6.4; 6.8; 7.2; 7.6; 8; 8.4; 8.8; 9.2; 9.6; 10];
>> [x, y, np, ifail] = d02ga(u, v, a, b, tol, 'fcn', x, np, 'mnp', int32(40));
??? The dimension of parameter 7 (x) should be at least 40
```

There are a number of other cases where we need to insert extra checks, not present in the underlying library, to catch programmer errors. We do this partly because we can (the nature of objects in MATLAB and Maple gives us extra information not available to the C or Fortran program), partly because this is how we expect code in these environments to behave, but also because the user of MATLAB or Maple does not have the tools (or quite possibly the software engineering skills) to track down and fix such problems themselves. This is a fundamental difference between an interactive package and a library designed to be used within a software development environment.

3.2 Testing

The Libraries are tested extensively to ensure that they are algorithmically robust and can handle errors in their input data correctly. For the embedded products however we additionally need to test that the interface between the host environment and the library is operating correctly. This is done by taking existing test programs and capturing the input and output data, and translating it to Maple or MATLAB statements as appropriate. We can then test that calling from the higher-level environment gives the same result as calling from C or Fortran. The necessary code to capture the data can be generated automatically from the XML specifications of the routines described earlier.

Note that there may be legitimate differences in the results obtained from the different environments. An example of this was provided by a user who was trying to minimise a function of the form $= e^f/e^g$ where f and g were expressions whose values could get very large. The user was unhappy with the answer that he was getting from MATLAB and wrote a program to call the NAG Library, which gave him the correct result. He then tried solving the same problem using the MATLAB toolbox and once again the result was wrong. It turned out that the evaluation of the exponentials in MATLAB was overflowing which did not happen in Fortran, presumably because there the compiler optimised the function to e^{f-g}.

With Maple we need to take care that we use hardware floating point arithmetic when testing to ensure that we get consistent results. A user, however, has the option of working to higher precision if he or she wishes.

3.3 Documentation

NAG provides detailed documentation for every routine in its library. This same documentation is transformed via stylesheets into a form suitable for inclusion in the host environment's help system. To ensure consistency with the software much of the logic about the structure of the interface is shared between the stylesheets used to generate the interface code, and those used to generate the documentation. The documentation is re-structured to follow the appropriate house style.

4 Conclusions

Although the implementation details of these two packages are rather different, the issues encountered during the design and implementation are very similar. Both environments allow the interfaces to routines to be simplified through the use of optional parameters and the elimination of parameters representing array dimensions etc, while still maintaining the functional flexibility that the routine interface offers.

It is clear that users of an interactive package have very different expectations to users of a library. The latter group are, by necessity, quite skilled in programming and software development. The former group, however, spans a much wider range of skillsets. To provide a safe environment for such users we had to make our packages completely robust even when that compromised efficiency.

To embed such a large number of routines in a package and produce good-quality documentation it was essential to automate as much of the process as possible. This is particularly true given that the host environments will change in future, as will the NAG Library. For these tasks we found XML and its various associated tools extremely useful, if a little clumsy at times.

Acknowledgements

The author would like to thank David Carlisle of NAG and Dave Linder of Maplesoft for their significant contributions to the development of the two packages described here. He would also like to thank the referees for their constructive comments.

References

[1] *The NAG Fortran Library Mark 21*, NAG Ltd, Oxford, 2005, `http://www.nag.co.uk`.

[2] *MATLAB 2007a*, The Mathworks Inc, Natick, 2007, `http://www.mathworks.com`.

[3] Michael B. Monagan, Keith O. Geddes, K. Michael Heal, George Labahn, Stefan M. Vorkoetter, James McCarron and Paul DeMarco, *Maple 10 Programming Guide*, Maplesoft, Waterloo, 2005, `http://www.maplesoft.com`.

[4] David Carlisle, Patrick Ion, Robert Miner and Nico Poppelier, *Mathematical Markup Language (MathML) Version 2.0*, World Wide Web Consortium, Recommendation REC-MathML2-20010221, February 2001