Symbolic and Numerical Computation for Artificial Intelligence

edited by

Bruce Randall Donald
Department of Computer Science
Cornell University, USA

Deepak Kapur
Department of Computer Science
State University of New York, USA

Joseph L. Mundy
AI Laboratory
GE Corporate R&D, Schenectady, USA

Academic Press
Harcourt Brace Jovanovich, Publishers
London San Diego New York
Boston Sydney Tokyo Toronto

ACADEMIC PRESS LIMITED
24–28 Oval Road
London NW1

US edition published by
ACADEMIC PRESS INC.
San Diego, CA 92101

This book is printed on acid-free paper

A catalogue record for this book is available from the British Library

ISBN 0-12-220535-9

Printed and Bound in Great Britain by
The University Press, Cambridge

Chapter 15

Symbolic and Numeric Computation: the Example of IRENA

James H. Davenport
School of Mathematical Sciences
University of Bath
Claverton Down, Bath BA2 7AY, UK
jhd@maths.bath.ac.uk
Michael C. Dewar
School of Mathematical Sciences
University of Bath
Claverton Down, Bath BA2 7AY, UK
mcd@maths.bath.ac.uk
Michael G. Richardson
Numerical Algorithms Group Ltd.
Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, UK
miker@nag.co.uk

Historically symbolic and numeric computation have pursued different lines of evolution, have been written in different languages and generally seen to be competitive rather than complementary techniques. Even when both were to used to solve a problem, *ad hoc* methods were used to transfer the data between them.

We first discuss the reasons for this dichotomy, and then present IRENA, a system being developed by the authors to present an integrated environment with all the facilities of Reduce combined with the functionality of the NAG FORTRAN library.

Not only does IRENA allow the Reduce user to make calls to the NAG Library interactively, it also converts a natural input representation to the required unnatural FORTRAN one and vice-versa on output, which results in a much more intuitive interface. Many parameters have default values and so need not be supplied by the user.

1. Introduction

"Computation" can mean different things to different people at different times. Indeed, it was only in the 1950s that "computer" came to mean a machine rather than a human being. The great computers of the 19th century, such as Delaunay, would produce formulae of great length, and only convert to numbers at the last minute. Today, "compute" is largely synonymous with "produce numbers", and often means "produce numbers in FORTRAN using the NAG library". However, a dedicated minority use computers to produce formulae. Generally the two methods have been viewed as alternatives, and indeed the following table shows some of the historic differences:

	Numeric	Symbolic
Common language	FORTRAN	Lisp
Best machine	Supercomputer	Workstation
Software collection	Library	System
Mode of computing	Batch	Interactive

Of course there are exceptions to these rules. MAPLE and MATHEMATICA are written in C, and MATHLAB is an interactive package of numerical algorithms; but in general it has been the case that computer algebra systems were interactive packages run on personal workstations, while numerical computation was done on large machines in a batch-oriented environment.

The reason for this apparent dichotomy is clear. Numerical computation tends to be very CPU-intensive, so the more powerful the host computer the better; while symbolic computation is more memory-intensive, and performing it on a shared machine might be considered anti-social behavior by other users.

Hardware has, of course, come a long way since these lines were drawn. Most researchers now have access to powerful workstations with a reasonable amount of memory and high-quality display devices. Graphics have become more important to most users and, even if numerical programs are still run on the departmental supercomputer, they are probably developed and tested on the individual's desk-top machine. Modern hardware is thus perfectly suited for both symbolic and numeric applications.

Not only has the last decade seen a revolution in hardware, this in its turn has led to major changes in the kinds of software available. Gone, thankfully, are the days when to exploit a computer effectively required deep knowledge of FORTRAN; modern packages are graphically-oriented with interactive front-ends driven by mice and icons. Thus computers have become more accessible and approachable, and with that has come a demand for user-friendly software to investigate and solve mathematical problems. This has lead to the growth in popularity of systems like Mathematica, while that of the large numerical libraries like NAG and IMSL has declined in relative terms (although absolute usage is still increasing).

2. The Symbolic / Numeric Interface

Problem solving often has three phases:

1 model the problem symbolically;
2 solve the problem numerically; and
3 analyze and display the results,

where all three phases are accomplished with a computer it is quite common to use two or three systems. The user uses a computer algebra system to derive a set of expressions which represent his or her particular problem. Typically, these are then converted into FORTRAN and somehow pasted into a program which probably calls NAG or other library routines to determine a solution. The results might be graphed or printed directly by the FORTRAN program, read back into the algebra system for this purpose, or another system might be used altogether.

Most computer algebra systems have a fairly basic method of translating algebraic expressions into FORTRAN. Useful enhancements include the ability to segment large expressions into statements of less than twenty lines, and perform some optimization on the generated code. Many different problems in a wide range of disciplines have been solved with these simple techniques (for surveys see Fitch, 1979; Ng, 1979). However an important step forward was the development of the GENTRAN package for Macsyma (Gates and Wang, 1984) and Reduce (Gates, 1985). GENTRAN has many capabilities, but the most notable ones are the ability to translate whole program segments into the target language (FORTRAN, RATFOR or C), and to "flesh-out" skeletal programs or *templates* provided by the user. A user wishing to solve several cases of the same basic problem can write a general program once, and let GENTRAN fill in the problem-specific parts automatically.

This latter facility has been used in the development of systems which take as input a symbolic description of a problem, do the necessary analysis and manipulation, and generate as output a "ready-to-run" FORTRAN program [e.g. FINGER (Wang, 1986) — a package for finite element analysis]. The FORTRAN must still be compiled and run by hand, and the results processed by the user.

Computer algebra systems have many features which assist in the presentation and analysis of data. Most have some form of graphical capability and can convert expressions into a typesetting language like TEX. The missing link is the ability to perform numerical computations within computer algebra systems. While some basic algorithms are available, no system can remotely begin to match the coverage and versatility of the NAG Library which contains over 900 user-callable routines in a wide variety of areas: quadrature, differential equations, linear algebra, root finding, interpolation, optimization and statistics to name but a few.

To combine such capabilities with that of a computer algebra system in one package would have many advantages, and would greatly facilitate the development of hybrid problem-solving systems in a variety of domains. Yet to duplicate the functionality of the NAG Library in a symbolic system would be neither practicable nor sensible. The NAG Library has been under development for two decades and represents a massive investment of effort and expertise. It has also achieved a reputation for the excellence and reliability of its algorithms and their implementations. In any case computer algebra systems are geared towards dealing with the rationals rather than with floating point numbers, and as a consequence their floating point facilities are comparatively slow.

We have attempted to fill this gap by developing a system which uses GENTRAN and Reduce to write programs which can call virtually any routine in the NAG Library (Dewar, 1989; Dewar and Richardson, 1990; Dewar, 1991). The user provides a symbolic description of the particular problem which is then converted into the form required by the NAG Library. Although the user can take the generated code and run it by hand if he or she wishes, the compilation and linking can be done automatically, and the results returned to the user within the interactive environment as algebraic objects.

IRENA — an Interface between Reduce and the NAG Library — can thus be looked at from several viewpoints:

- it provides an easy-to-use, interactive front end to the NAG Library;
- it provides an integrated symbolic / numeric computation environment for general users;
- it provides a toolkit of numerical methods for developers of problem-solving packages.

The following sections describe it in more detail.

3. An Introduction to IRENA

We shall illustrate some of the main features of IRENA with an example. Suppose we wish to find:

$$\min\left((x_1+10x_2)^2+5(x_3-x_4)^2+(x_2-2x_3)^4+10(x_1-x_4)^4\right)$$

subject to the conditions:

$$\begin{array}{rcccl} 1 & \le & x_1 & \le & 3 \\ -2 & \le & x_2 & \le & 0 \\ 1 & \le & x_4 & \le & 3 \end{array}$$

starting from the initial guess $(3,-1,0,1)$. We have decided to use routine `E04JAF` which uses a quasi-Newton algorithm, and is designed to be used with continuous functions with continuous first and second derivatives. The session with IRENA is in figure 1.

The user has provided three parameters: the constraints as a rectangle named *bounds*, the initial guess as a vector *start*, and the function to be minimized as a function *f*. The parameters are provided by a sequence of keys separated by commas. The order is unimportant and, as we shall see later, the number of parameters used in a particular call may vary. The NAG routine actually takes eleven parameters, and also requires the user to write a subprogram to represent the function. These parameters are described in figure 2.

The three arguments passed by the user to IRENA determine all the objects required by the FORTRAN. The size of the problem — in this case the number of independent variables — determines the size of the workspace arrays. The rectangle yields the constraints, and the fact that the user has provided it at all causes `IBOUND` to be set correctly. The names have been made more descriptive and, in the case of `X` which is used

```
1: e04jaf(bounds=[ 1:3, -2:0, *:*, 1:3],
1:     vec start {3,-1,0,1},
1:     f(x1,x2,x3,x4)=(x1 + 10*x2)^2 + 5*(x3 - x4)^2
1:                                       + (x2 - 2*x3)^4 + 10*(x1 - x4)^4);
 ** ABNORMAL EXIT from NAG Library routine E04JAF: IFAIL =     5
 ** NAG soft failure - control returned

    There is some doubt about whether the point x found by
    E04JAF is a minimum. The degree of confidence in the result
    decreases as IFAIL increases. Thus, when IFAIL = 5 it is
    probable that the final x gives a good estimate of the position
    of a minimum, but when IFAIL = 8 it is very unlikely that the
    routine has found a minimum.

For an index to the following list, type '@0;'.  The values of its
entries may be accessed by their names or by typing '@1;', '@2;' etc.

{MINIMUM_VALUE,LOCATION_OF_MINIMUM,BL,BU}

2: location_of_minimum;

[        1.0         ]
[                    ]
[ - 0.0852325899147751]
[                    ]
[  0.4093035914975   ]
[                    ]
[        1.0         ]


3: minimum_value;

2.4337875121207
```

Figure 1. An example using IRENA.

Parameter	Type	Purpose
N	INTEGER	The number of independent variables
IBOUND	INTEGER	Determines how bounds are provided
BL	REAL Array	Contains the lower bounds
BU	REAL Array	Contains the upper bounds
X	REAL Array	On entry contains the starting point
		On exit contains the position of the minimum
F	REAL	On exit contains the value of the minimum
IW	INTEGER Array	Workspace
LIW	INTEGER	The dimension of IW
W	REAL Array	Workspace
LW	REAL	The dimension of W
IFAIL	INTEGER	The diagnostic parameter
FUNCT1	SUBROUTINE	Represents the function

Figure 2. The parameters of NAG routine E04JAF.

for both input and output, the names of the two structures are different. A non-zero value of IFAIL has been detected on exit, and an appropriate error message displayed. The output structures have been transformed into Reduce algebraic objects, and may be inspected at will. A list of all the output parameters is returned.

4. Providing Default values for NAG Parameters

Frequently, input parameters of a FORTRAN routine can be given default values appropriate for some or all instances of the routine's use. A feature of IRENA is the "defaults" system, which allows appropriate default parameter values to be specified by the system developers or the user. Once a default value is established for a parameter, it need not be specified in the IRENA function call, thus considerably simplifying the user interface, compared to the native FORTRAN call. Input parameters of NAG FORTRAN Library (and indeed many other) routines fall into three general categories:

1 data parameters;
2 algorithmic control parameters; and
3 housekeeping parameters.

A number of borderline cases occur in the NAG Library but these will not immediately concern us.

The concept of a data parameter is largely self-explanatory: these parameters specify the data which the user wishes to process. Specifying default values for such parameters would be completely inappropriate.

Algorithmic control parameters, as their name suggests, control the execution of the algorithm. Examples are limits on the number of iterations of a process and convergence criteria. Appropriate defaults can often be found, perhaps for particular cases, by careful scrutiny of the NAG Library manual or other sources. However, the user may well wish to override such defaults. Other types of control parameters exist, controlling, for instance, the levels of diagnostics or printing of intermediate results. These are included in the same category since users may, at times, be expected to override the defaults.

In the category of housekeeping parameters we include all parameters whose values do not logically form part of the statement of the problem or constraints on its mode of solution. This includes such things as the dimensions of input and workspace arrays. Normally their values are in some sense dependent on the size of the particular problem being solved. Although it is unlikely that the user would wish to override the default value of such a parameter, there is no mechanism preventing this.

NAG workspace arrays are required to have a certain minimum size. Below this size, the algorithm will simply not function. Increasing the array size above this will normally have no observable effect; occasionally it will cause the routine to run faster. In a very few cases, the array size effectively provides a limit on the number of iterations of the algorithm: in these cases, we would consider the array size to be an algorithmic control parameter. Defaults for the lengths of workspace arrays are therefore normally classed as housekeeping but occasionally as algorithmic control.

Each routine may have associated with it a default file containing default *values* for control parameters and default *expressions* for the housekeeping parameters. The user is

Table 1. Matrices with row lists: uppermost row first throughout. (i represents the row, and j the column index).

Type	Representation
full	each inner list specifies a row
symmetric	each inner list specifies that part of a row for which either $i \geq j$ or $i \leq j$
skew-symmetric	each inner list specifies that part of a row for which either $i \geq j$ or $i \leq j$
Hermitian	each inner list specifies that part of a row for which either $i \geq j$ or $i \leq j$
strict upper triangular	each inner list specifies that part of a row for which $i < j$
upper triangular	each inner list specifies that part of a row for which $i \leq j$
upper Hessenberg	each inner list specifies that part of a row for which $i \leq j + 1$
strict lower triangular	each inner list specifies that part of a row for which $i > j$
lower triangular	each inner list specifies that part of a row for which $i \geq j$
lower Hessenberg	each inner list specifies that part of a row for which $i \geq j - 1$
general band (variable bandwidth)	each inner list specifies that part of a row lying within the envelope, the list being packed out with zeroes for symmetry about the diagonal
symmetric band (variable bandwidth)	each inner list specifies that part of a row, lying within the envelope, for which $i \geq j$

also free to set up his or her own default files, and can override the default value of any parameter by giving it a value in the call to IRENA.

5. Improving The NAG Interface

As can be seen from the example in figure 1, we have adopted new forms for some of the parameters. The bounds were expressed as a *rectangle*, the starting point was a *vector*, and the subroutine was represented by a polynomial. In fact, the user could have used the standard NAG forms of most of the parameters (all except the subroutine), but clearly the enhanced interface is nicer. In this section we shall describe some of the techniques we use to do this for arrays and scalars, and in section 6 we shall talk about how we represent those parameters which are in fact subprograms.

5.1. Matrices

Reduce represents matrices densely, with each element being specified explicitly. In numerical computation, however, one is often dealing with matrices with some special structure. Thus we have provided a suite of functions which allow the user to input these structured matrices to IRENA. These can also be converted to Reduce matrices and manipulated in the usual way. An example is the vector *start* in figure 1. A full list of the other types, and their representations, is given in tables 1–3. In general this is a list of lists, the type of matrix determining the relationship between the inner lists and the actual elements.

Table 2. Matrices with diagonal lists: uppermost diagonal first throughout.

Type	Representation
band (fixed bandwidth)	each inner list specifies a "diagonal"
symmetric band (fixed bandwidth)	only the superdiagonal and diagonal (or diagonal and subdiagonal) lists

Table 3. Sparse matrices.

Type	Representation
sparse	3 inner lists, each in same arbitrary order, containing: first list — row indices of non-zero elements second list — column indices of non-zero elements third list — non-zero elements
symmetric sparse	as sparse, restricted to either upper or lower triangle.
long sparse	a list of triples {r,c,v} representing the row index, column index and value, respectively, of the non-zero elements (in arbitrary order)
symmetric long sparse	as long sparse, restricted to either upper or lower triangle

5.2. Input Jazzing

The process by which scalars and matrices are transformed is known as *jazzing*. This section describes how we deal with input parameters and transform the user's representation into the one NAG expects.

5.2.1. Aliases

The simplest way in which jazzing is used is to provide different names for parameters. There are a number of cases where this is useful:

- FORTRAN restricts names to six characters, and these are often not very meaningful;
- It is preferable to have the same names for equivalent parameters across a whole chapter, and indeed in some cases across the whole Library;
- Different groups of users use different terminology.

There can be several aliases for the same object, or even aliases to aliases, and of course the user is still free to use the original parameter name, if desired. In addition to the system-provided aliases, we allow the user to set up his or her own alias file.

5.2.2. New Scalars

Sometimes the NAG parameter is not the natural parameter. For example, the NAG routine `E02ADF`, which computes least-squares polynomial approximations to an arbitrary set of data points, has a parameter `KPLUS1` whose value is one plus the maximum degree required. This form of jazzing transforms the IRENA parameter, in this case k, to its NAG equivalent.[†]

5.2.3. KEYWORDS

Some NAG parameters can only take a limited number of values: e.g. `.TRUE.` or `.FALSE.`. In this case we define a set of keywords, each of which is equivalent to one of these cases. For instance, the routine E02BCF evaluates a cubic spline and its first three derivatives from its B-spline representation. It has a parameter, `LEFT`, which specifies whether left or right handed values are to be computed, depending on whether its value is 1 or not. IRENA has a pair of keywords *left* and *right*, which can be interpreted as `LEFT=1` and `LEFT=0` respectively. Thus a typical call to `E02BCF` would look like:

```
4: e02bcf(vec knots {0,1,3,3,3,4,4,6},
4:         vec coefs {10,12,13,15,22,26,24,18,14,12}, x=0, right);
```

Normally we also provide the NAG parameter with a default value, so that one of the keywords is in some sense redundant, and supplied for symmetry only.

5.2.4. RECTANGLES

NAG normally represents a rectangular region either as two scalars (in the one-dimensional case), or two arrays of lower and upper bounds. In IRENA we define a rectangle to be a single object in its own right, consisting of a set of pairs of numbers surrounded by square brackets, e.g. the constraints on the variables in the optimization routine in figure 1.

5.2.5. VERY LOCAL CONSTANTS

Sometimes NAG attaches special meanings to certain values. For example, in the example shown in figure 1, the FORTRAN arrays `BL` and `BU` contain the lower and upper constraints on the values of the x_i. If the value given is a very large negative or positive number respectively, then this is taken to mean that the value of that particular x_i is unconstrained in that particular direction. In the example x_3 is completely unconstrained, and in the rectangle *bounds* its constraints are denoted by asterisks. Each asterisk in fact means something different. For the upper bound it means *fphuge* — the largest floating point number — while for the lower bound it means *-fphuge*. The asterisk is a *very local constant*, and it enables us to provide a uniform interface within a routine. In general the asterisk character is interpreted as meaning that a parameter is "unset", i.e. it is not given a value.

[†] This compares with the situation where the NAG user might be expected to provide values for both `K` and `KPLUS1`. In this case we would class `KPLUS1` as a housekeeping parameter and give it the default value $k + 1$.

5.2.6. Jazzing Matrices

There are three main reasons for jazzing arrays on input:

1 NAG arrays are sometimes confusing, with different columns being used for different purposes (e.g. `W` in `D02YAF` which has a variable number of columns used for inputting values of derivatives, returning results, and workspace). Jazz allows the user to specify their separate logical components and then assembles them correctly.
2 Matrices with a special structure are represented by NAG routines in a multitude of ways to make efficient use of memory, e.g. two triangular matrices might be packed into one FORTRAN array to save space. However, we have provided representations for matrices which preserve these structures, and so need to transform the IRENA or Reduce representation to the NAG one. Note that we do not insist that, for example, a triangular matrix be represented explicitly as an IRENA triangular matrix; jazz will try and coerce any IRENA or Reduce matrix to the required type.
3 NAG routines often expect the user to provide a large array, only some of whose elements are set. This is usually to allow the routine to manipulate elements of the array in place, rather than using separate workspace. An example is `MU` in `E02DAF` whose first and last four elements are zeros. It is nicer to allow the user to provide the smaller structure, which jazz then "pads-out" to the larger one.

5.2.7. Complex Objects

The FORTRAN standard (ANSI, 1978) is somewhat confused about the representation of complex numbers. Whilst it provides a single precision complex data type, there is no double precision analogue. This has led to a great deal of inconsistency between the available compilers: some follow the standard and offer no double precision complex data type at all; while others do, but call them by a variety of names (`double complex` on SUNs, `COMPLEX*16` on IBM machines etc.). As a result, the implementors of algorithms have chosen a variety of representations for complex objects. For scalars the normal ones are:

- a pair of scalars representing the real and imaginary parts;
- a vector of length two, containing the real and imaginary parts (and corresponding to the normal implementation of the complex data type).

while for arrays the common representations are:

- a pair of arrays representing the real and imaginary parts;
- a single vector, those elements with odd indices being the real parts and those with even ones being the imaginary parts (again, this corresponds to the normal implementation of the complex data type).

There are a few rather more obscure representations, mainly where we are dealing with an array with some special structure (e.g. Hermitian).

In IRENA we expect the user to provide a normal Reduce complex object, which we will then convert to the appropriate format.

5.2.8. Unpacking Matrices

Most jazzing so far has consisted of taking small logical objects and constructing larger FORTRAN objects from them. Occasionally, however, we would like to take a matrix provided by the user and make each element into a FORTRAN scalar. For example, the NAG routine **C02AJF** finds the roots of a quadratic equation with real coefficients using the well-known formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where the coefficients a, b, c are provided by the user as three scalars. However, for consistency with the rest of the C02 chapter, we would like the user to be able to provide them as a vector called *coefficients*. Hence the interface can be either:

```
5: c02ajf(vec coefficients{10,3,1});
```

or:

```
6: c02ajf(a=10,b=3,c=1);
```

5.3. Output Jazzing

In this section we describe how we take the objects returned by the FORTRAN routines, and coerce them into something more natural for the user.

5.3.1. Matrices

There are three main uses for output jazzing of arrays:

1 Disentangling FORTRAN arrays into their logical components. In section 5.2.6 we described W in D02YAF which has some columns used only for input and some for output. Clearly we want to return the output columns as separate structures.
2 Unpacking arrays to restore some structure to them. For example, two triangular matrices are sometimes returned as a single array, IRENA unpacks them and returns the two matrices separately.
3 Trimming large structures into smaller ones. Analogous to the third case in section 5.2.6 we now wish to get rid of the excess padding we added earlier. Alternatively, we might wish to return just the "interesting" bits of a workspace array: for example, many quadrature routines return partial results this way which can be useful if an error occurs.

5.3.2. Complex Objects

On input we generally create real objects from given complex objects, on output we do the reverse and generate complex objects from the real data returned by the FORTRAN routine.

5.3.3. Packing objects into larger structures

Analogous to section 5.2.6, here we take several output parameters and turn them into an array. For instance, in the example given, the FORTRAN routine returns two vectors **ZSM** and **ZLG**, representing the two (complex) roots. We transform them into the array *roots*, for consistency with the rest of C02.

5.3.4. Output Aliasing

Analogous to the input case, we often wish to give FORTRAN output parameters more meaningful names.

6. Argument Subprograms (ASPs)

Argument subprograms are parameters to NAG routines which are themselves either functions or subroutines. They are used for a variety of purposes, as detailed below, and occur throughout the library. The nearest Reduce equivalent to a FORTRAN subprogram is an algebraic procedure, but to ask the user to provide ASPs in this form is unsatisfactory for the following reasons:

- The differences between the two languages make producing equivalent pieces of code difficult. FORTRAN uses call-by-reference semantics, while Reduce uses call-by-value. FORTRAN requires explicit type declarations, while RLISP (the language recognized by the Reduce interpreter) only requires that variables be local (i.e.*scalar*) or globals (the default). A Reduce procedure can be translated into FORTRAN by GENTRAN, and the tokens *real* and *integer* can be used instead of *scalar* to give type information, but the two pieces of code will not be semantically equivalent. To generate correct code requires that the Reduce user understand FORTRAN, something we whole-heartedly wish to avoid.
- Matrices are treated in a rather clumsy fashion by Reduce; they can only be global variables, rather than local.
- We are trying to get away from the idea that the user needs to write a *program* to solve a problem, and to ask the user to encapsulate a mathematical object as a Reduce procedure would undermine that. We prefer to ask the user to provide these objects in their natural form: for example, to generate an ASP which returns the value of a function at a given point, only the definition of the function is necessary.

Thus we provide alternative representations for all ASPs. In this way the ASP system is conceptually similar to the jazz system, the main difference being that in this case the alternative form is compulsory.

6.1. Function values

The most common use of an ASP is to evaluate a given function or set of functions at an (arbitrary) point, e.g. an integrand or a set of differential equations. In IRENA the user is expected to supply a set of expressions corresponding to the functions, and the ASP system will generate the appropriate FORTRAN. For example, a call to the integration routine D01GBF might look as follows:

```
7: d01gbf( region=[0:1,0:1,0:1,0:1],
7:         f(w,x,y,z)=4*w*y^2*e^(2*w*y)/(1 + x + z)^2);
```

The dummy parameters w, x, y, z are replaced by the correct FORTRAN parameters, in this case elements of the array X, during code generation to produce the following:

```
      DOUBLE PRECISION FUNCTION FUNCTN(NDIM,X)
      DOUBLE PRECISION X(NDIM)
      INTEGER NDIM
      FUNCTN=4*DEXP(DBLE(2*X(3)*X(1)))*(X(4)**2+2*X(4)*
     . X(2)+2*X(4)+X(2)**2+2*X(2)+1)**(-1)*X(3)**2*X(1)
      RETURN
      END
```

In many cases we are dealing not with a single function but with a set of functions, e.g. a set of differential equations. We provide two methods for entering such functions. The first is analogous to the simple case above, while the second is useful for large sets of "related" functions. Mathematicians usually represent sets of functions with subscripts; in IRENA the equivalent notation is to suffix the function name with its index to produce a new name. This method is used in the key list as follows:

```
8: d02bbf( range=[0:8],
8:         vec initial_values {0.0,0.5,pi/5},
8:         fcn1(tt,yy,v,phi)=tan(phi),
8:         fcn2(tt,yy,v,phi)=-0.032*tan(phi)/v - 0.02*v/cos(phi),
8:         fcn3(tt,yy,v,phi)=-0.032/v^2,
8:         vec output {1,2,3,4,5,6,7,8});
```

An extension to more general functional notation is useful where we have a set of functions like:

$$f_i(x_1, x_2, \ldots, x_9) = -x_{i-1} + (3 - 2 * x_i) * x_i - 2 * x_{i+1} + 1$$

with appropriate modifications for the extreme values of i. In IRENA this is coded as follows, either in the global Reduce environment, or in the key list:

```
9:  fset fcn[1](x[1:9])=(3-2*x(1))*x(1)-2*x(2)+1;
10: fset fcn[i=2:8](x[1:9])=-x(i-1)+(3-2*x(i))*x(i)-2*x(i+1)+1;
11: fset fcn[9](x[1:9])=-x(8)+(3-2*x(9))*x(9)+1;
```

We can find the zero of this set of equations with the following call to C05NBF. The values of *fcn* will be picked up automatically.

```
12: c05nbf(vec start {-1,-1,-1,-1,-1,-1,-1,-1,-1});
```

6.2. Jacobian and derivative values

Quite often NAG routines require the user to write a routine to calculate the Jacobian or Hessian matrix, or the derivatives of a given set of functions. Although in theory an easy task, in practice errors are easy to make but difficult to detect. In IRENA we calculate derivatives automatically, expecting the user to provide only the original functions (which are normally required anyway for another ASP).

Because Jacobians and Hessians by their very nature consist of large numbers of related expressions, the efficiency of the generated code can be dramatically increased through the use of symbolic optimization.

6.3. Dummy Routines

Sometimes NAG offers the user the choice of either writing their own routine, or of calling one in the Library. This is often the case when the routine is required to monitor the progress of the computation and output diagnostic information. In these cases (where appropriate) we automatically generate a dummy routine to call the NAG routine without further input from the user.

6.4. Output Routines

Occasionally NAG routines require the user to provide a routine to output intermediate information during the execution of the algorithm. IRENA provides a procedure which may or may not require some input from the user, such as an array of points at which to generate diagnostics. Moreover the resulting information is available as an actual structure within Reduce, rather than simply printed out.

6.5. Matrix Manipulation Routines

These are routines required to manipulate matrices in some way, often a specific (user-supplied) matrix for a given problem. In such cases IRENA requires that the user supply the relevant matrix, and the routine is generated using either Reduce's symbolic manipulation facilities, or a call to an appropriate NAG routine.

6.6. Regions

Some NAG routines are concerned with evaluating the endpoints of regions. We expect the user to express the region as an IRENA *rectangle* (see section 5.2.4), whose end points may be either expressions or constants.

7. Implementation Details

The Sun 4 implementation of mark 14 of the NAG Library is big — nearly nine megabytes of compiled code — and so it is not practical to simply link the whole of the NAG Library with Reduce! In addition, although IRENA provides access to any individual routine, it does not (yet) allow the nesting of routines, to minimize an integral for example. Thus we decided early on to aim to generate *complete programs* to solve problems, and then dynamically link these into the running Reduce. Since we are generating

```
13: integrate(integrand(x)=sin(x)/x,region=[-1:1]);

For an index to the following list, type '@0;'.  The values of its
entries may be accessed by their names or by typing '@1;', '@2;' etc.

{INTEGRAL,ABSOLUTE_ERROR_ESTIMATE,INTERVALS,A_LIST,B_LIST,E_LIST,

 R_LIST}

14: integral;

1.89216614073437

15: absolute_error_estimate;

1.05036320792971E-14
```

Figure 3. Choosing a NAG routine.

programs, there are also facilities to *only* generate code (i.e. disable the compile / link / execute cycle), which can then be taken away and executed on a different processor, or in a batch environment etc.

Although this approach may appear slow it does offer advantages. In practice, if the user needs to generate an ASP then this will need to be compiled and linked anyway. A more detailed discussion of this topic, and a description of the method used to perform the dynamic linking, can be found in Dewar and Richardson (1990) and Dewar (1991).

We have described the way we can simplify the interface to a NAG routine. Clearly with over 900 routines to deal with, to do this for every routine requires a great deal of effort. Thus we have developed a suite of tools and methods to generate a skeletal interface automatically from the NAG documentation. So, while the jazzing of each routine must be done by hand since it is very much a subjective matter, most of the rest of the work to build the interfaces is accomplished automatically. The full details may be found in Dewar (1991).

8. Applications

Obviously a major use of IRENA is as an interactive numerical analysis package. However, a more interesting idea is to use it as part of a larger problem solving system. One area which has already been investigated is that of quadrature (Dewar, 1991). Given an integral to be solved over a particular region, the package will select the NAG routine which seems best suited to the particular integral and call it. An example is shown in figure 3.

9. Conclusions

In this paper we have described how the division between symbolic and numeric computation arose largely because of the differing hardware requirements of the two paradigms and suggested that, since this is no longer a problem, the time is ripe to attempt to reconcile the two approaches. To this end we have developed IRENA, which provides an environment containing high-quality symbolic and numeric methods. Not only does IRENA give the Reduce user access to a suite of high-quality numerical algorithms, but it also uses the symbolic facilities of Reduce to considerably simplify the interfaces to them. Although one use of IRENA is as a simplified, interactive front-end to the NAG Library, we believe that it has much greater potential. We believe that IRENA and Reduce together provide a toolkit of high-quality building blocks for developers of problem-solving systems in many domains.

References

ANSI (1978), *American National Standard Programming Language FORTRAN,* Technical Report ANS X3.9, American National Standards Institute.

M.C. Dewar and M.G. Richardson (1990), "Reconciling symbolic and numeric computation in a practical setting", *Proc. Design and Implementation of Symbolic Computation Syst.*, Springer-Verlag, NY, 195–204.

M.C. Dewar (1989), "IRENA —an integrated symbolic and numerical computation environment", *Proc. Int. Symp. Symbolic Algebraic Computation*, ACM, Portland, Oregon, 171–179.

M.C. Dewar (1991), *Interfacing Algebraic and Numeric Computation*, PhD Thesis, School of Math. Sci., University of Bath, Claverton Down, Bath.

J.P. Fitch (1979), "The application of symbolic algebra to physics — a case of creeping flow?", *Proc. EUROSAM '79*, Springer-Verlag, NY, 30–41.

B.L. Gates and P.S. Wang (1984), "A LISP-based RATFOR code generator", *Proc. 1984 MACSYMA User's Conf.*, Schenectady, NY, 319–329.

B.L. Gates (1985), "Gentran: an automatic code generation facility for Reduce", *ACM SIGSAM Bull.*, 19(3), 24–42.

E.W. Ng (1979), "Symbolic–numeric interface: a review", *Proc. EUROSAM '79*, Springer-Verlag, NY, 330–345.

P.S. Wang (1986), "FINGER: a symbolic system for automatic generation of numerical programs in finite element analysis", *J. Symbolic Computation*, **2**, 305–316.