University of Bath

PHD

User interfaces for numeric computation in symbolic environments

Richardson, Michael Gerard

Award date:
1997

Awarding institution:
University of Bath

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 13. May. 2019

User Interfaces for Numeric Computation in Symbolic Environments

submitted by

Michael Gerard Richardson

for the degree of Ph.D

of the

University of Bath

1997

Attention is drawn to the fact that copyright of this thesis rests with its author and with The Numerical Algorithms Group Limited. This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that copyright rests with the author and The Numerical Algorithms Group Limited and that no quotation from the thesis and no information derived from it may be published without the prior written consent of The Numerical Algorithms Group Limited.

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signature of Author .....

Michael Gerard Richardson

UMI Number: U098745

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI U098745
Published by ProQuest LLC 2014. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

UNIVERSITY OF BATH
LIBRARY
29 APR 1993
512677

Summary

This thesis considers the issues involved in providing user interfaces to libraries of numerical software, for use in symbolic computation packages; in particular, it discusses and analyses the experience of the author in providing such interfaces for IRENA (a link from the REDUCE computer algebra system to NAG Fortran libraries), placing this in the context of both the earlier NAG Library link from Macsyma (Naglink) and the recent, more basic link from Axiom (NAGlink).

Design goals for the reparameterisations of NAG routines in IRENA were that these should be informative, regular, orthogonal and minimal. The thesis examines the methods used to achieve these goals and analyses the resulting simplification in the routines' user interfaces.

The complexity of the original Fortran interface of a NAG routine may be expected to influence the magnitude of the reparameterisation task. A statistical analysis of the relationship between the amount of code required to carry out the reparameterisation and the number of parameters in the NAG routine reveals strong evidence of considerable nonlinearity in this relationship, which appears to include substantial quadratic and cubic components. (As these are of opposite sign, the net effect is to produce a curve which, over the range considered, departs from linearity most strongly near the origin.)

Other points covered include the need to consider both numeric and symbolic issues in designing interfaces, the possible unification of the various IRENA subsystems in future projects, the utility of developing a library of Fortran "jackets" for the NAG Libraries as part of such a project and the suitability of various strategies and languages for interface redefinition.

IN MEMORIAM

ALAN ROSS ANDERSON

Acknowledgements

I should like to express my thanks to: the management of The Numerical Algorithms Group for their support in the development of IRENA and in the writing of this thesis; to the many colleagues at NAG and Bath who freely supplied information and advice – in particular to Mike Dewar, without whose development of the underlying system code IRENA would never have materialised; to my wife and children, who have received far less attention than they deserved whilst this thesis was being written; and, particularly, to my supervisor, Professor James Davenport, for his support and especially his patience in the light of my protracted endeavours to produce this thesis as a part-time activity.

Finally, I should like to remember my friend and former teacher, the late Professor Alan Anderson, whose encouragement at a critical time was in large measure responsible for my return to mathematical studies.

Contents

I Background and design issues 1

1 Introduction 2

1.1 The IRENA project 2

1.2 Experience with Naglink 3

1.3 IRENA-0 design activity 4

1.4 Basic design considerations 5

2 Design philosophy 7

3 Outline of the IRENA system design 12

3.1 Overall design 12

3.2 GENTRAN use 16

3.2.1 Gentranopt 17

4 Classification of NAG parameters 18

4.1 Classification in NAG Library documentation 18

4.2 Functional classification of NAG input parameters 19

5 Strategy for meeting the design objectives 23

5.1 Informativeness 24

5.2 Regularity 24
5.3 Orthogonality 26
5.4 Minimality 26

6 Parameter representation in IRENA **28**

6.1 IRENA matrix representation 30
6.2 "Rectangular" regions 30
6.3 Function families 31
6.3.1 User-defined functions 31
6.3.2 Single parameter function families 32
6.3.3 More general function families 39
6.4 Data input from files 40
6.5 Output naming 40
6.6 Non-parametric output 41

7 Defaults and jazzing in IRENA-0 **42**

7.1 The defaults system 42
7.2 The basic jazz system 46
7.3 The extended jazz system 47
7.3.1 Jazz-functions 47
7.3.2 Output-functions 48

II Development of IRENA-1 49

8 Overview of IRENA-1 development 50

8.1 Choice of routines for IRENA-1 50

8.2 Difficulties encountered in completing IRENA-1 53

8.3 The effect of NAG routine complexity on IRENA development 54

9 Modifications to IRENA-0 58

9.1 Switches 58

9.2 Prompting 61

9.2.1 Name substitution 61

9.2.2 Boolean variables 62

9.2.3 Prompt types 63

9.2.4 Keywords 63

9.2.5 Phased-prompt 64

9.2.6 Promptall 66

9.3 Output enhancements 67

9.3.1 Output indexing 67

9.3.2 Hidden output 68

9.3.3 Enhanced conditional output 68

9.4 The IRENA help system 69

9.5 Mnemonically named functions 72

9.6 Handling Hermitian sequences 72

9.7 Zero-filling arrays 73

10 Default system usage in IRENA-1 75

10.1 Descriptions of selected defaults files 75

10.1.1 F02BJF 75
10.1.2 E04GCF 78
10.1.3 E02ADF and E02AEF 79
10.1.4 D01BBF 82
10.1.5 F04MAF 84

11 Jazz usage in IRENA-1 **86**

11.1 Jazz files for the routines discussed in chapter 10 87
11.1.1 F02BJF 87
11.1.2 E04GCF 93
11.1.3 E02ADF and E02AEF 95
11.1.4 D01BBF 97
11.1.5 F04MAF 99
11.2 Further jazz files 103
11.2.1 E04MBF 103
11.2.2 E01SEF and E01SFF 109
11.2.3 D01ALF 113
11.3 Cond-out 116
11.4 Conclusion 118

12 Jacketed routines **119**

12.1 Reasons for developing jackets 119
12.2 Other potential uses for jackets 123
12.2.1 Modular routines 123
12.2.2 User control of error tolerance 124
12.2.3 Reverse communication 124

12.2.4 Error recovery 125

12.2.5 Handling special features 126

12.3 Effectiveness of jackets 127

12.3.1 Utility to users 128

13 REDUCE-like interfaces **129**

13.1 An experimental higher level interface 131

14 Documentation of IRENA-1 **134**

14.1 The User Guide 135

14.1.1 The chapters 135

14.1.2 The appendices 139

14.2 Function descriptions 140

14.3 Other documentation 142

14.4 Comparison with NAG documentation 142

14.4.1 The User Guide 142

14.4.2 The function documents 143

III Conclusions and recommendations 147

15 Considerations for the design of future IRENA-like systems 148

15.1 Precedence of non-housekeeping defaults 148

15.1.1 Parameter evaluation strategy in IRENA-1 148

15.1.2 Control parameters 149

15.1.3 Data parameters 149

15.1.4 Recommendation for future enhancements 150

15.2 Are defaults distinct from jazzing? 151

15.2.1 Impact on specfiles 154

15.3 Generalising interface design 155

15.3.1 Analysis of IRENA jazz command usage 157

15.3.2 Routine selection 162

15.3.3 Compilation and ASPs 163

15.3.4 Conclusion 164

15.3.5 Example 165

15.4 Considerations for the design of a revised jazzing system 166

15.4.1 General features 168

15.4.2 Type declarations 169

15.4.3 Redefinitions 171

16 Recommendations for NAG Library development 176

16.1 Software guidelines for library development 176

16.1.1 Argument SubPrograms 177

16.1.2 Option setting 177

16.1.3 Uncontrollable termination 177

16.1.4 Natural data representations 177
16.1.5 Matrix representations 178
16.1.6 Complex quantities 178
16.1.7 Naming conventions 178
16.1.8 Misclassification and misplaced information 179
16.1.9 Special data representations 179
16.1.10 Assumed-size arrays 180
16.1.11 Summary 180
16.2 Material for direct inclusion in symbolic packages 181
16.3 Packages as sources of library material 182
16.4 Structure of NAG documentation 182
16.4.1 F06 documentation 184

17 Impact of IRENA on other NAG software **186**

17.1 Impact on the Axiom-NAG link 186
17.2 Impact on other software 189
17.2.1 Fortran 90 Library 189
17.2.2 The NAG Product Information Database 189
17.2.3 The MATLAB gateway generators 189

18 Final overview **190**

18.1 Effort required with more complex interfaces 190
18.2 Ease of use 191
18.3 Unification of control mechanisms 193
18.4 Functionality and multiple numeric calls 194
18.5 Symbolic-numeric interaction 196
18.6 Conclusion 197

IV Appendices 198

A Code size and authorage 199

A.1 Code attribution 199

A.2 Distributed size of IRENA-1 201

B NAG, Naglink and IRENA parameterisations of a specimen routine 202

B.1 Example calls 203

B.1.1 NAG (Fortran 77) call 203

B.1.2 Naglink call 203

B.1.3 IRENA call 204

B.2 Matrix defining parameters 204

B.3 Accessing the results 204

B.4 Workspace parameters 205

B.5 Other parameters 205

C Example IRENA-function description 206

D Brief descriptions of jazzing commands 209

D.1 Input jazzing commands 211

D.2 Output jazzing commands 219

E Data used in the complexity analysis 226

F Main GLIM run 232

G Source of nagpolysolve 246

H Fortran 90 jacket for D01AJF and related modules 251

H.1 General precision setting 252

H.2 Second level defaults 252

H.3 Derived types for named output 252

H.4 Jacket for D01AJF 253

H.5 Test program 258

H.6 Test program output 260

I Current and extended specfiles **262**

Glossary **275**

Bibliography **283**

List of Figures

3-1 Information flow in standard IRENA setup processes 13

3-2 Outline of run-time IRENA-function processing 14

8-1 Plot of IRENA file sizes versus NAG parameter numbers 55

9-1 IRENA Timing Output 60

10-1 Annotated defaults file for **F02BJF** 76

10-2 Annotated defaults file for **E04GCF** 79

10-3 Annotated defaults file for **E02ADF** 80

10-4 Defaults file for **E02AEF** 80

10-5 Defaults file for **D01BBF** 83

10-6 Defaults file for **F04MAF** 84

11-1 **F02BJF** jazz file – input jazz commands 87

11-2 **F02BJF** jazz file – output jazz commands 89

11-3 **E04GCF** jazz file 93

11-4 **E02ADF** jazz file 96

11-5 **E02AEF** jazz file 97

11-6 **D01BBF** jazz file 98

11-7 **F04MAF** jazz file – input jazz commands 100

11-8 **F04MAF** jazz file – output jazz commands 102

11-9 **E04MBF** jazz file – part 1 103

11-10 **E04MBF** jazz file – part 2 104

11-11 **E04MBF** jazz file – part 3 105

11-12 **E01SEF** jazz file 110

11-13 **E01SFF** jazz file 111

11-14 Body of **D01ALF** jazz file 114

11-15 Location of singular vectors in **F02XEF** output 116

11-16 Cond-out usage in the **F02XEF** jazz file 117

12-1 "Inverse" jacket for C06EAF 120

12-2 Jacket for E01SBF 121

List of Tables

5.1 Correspondence between IRENA and NAG parameter names 25

6.1 IRENA matrix representations 29

7.1 Functions available in defaults files 43

7.2 NAG constants available in defaults files 44

12.1 NAG routines jacketed in IRENA 1.0 120

14.1 Sizes in bytes of NAG and IRENA documents 145

14.2 Size ratios of IRENA and NAG documents 145

15.1 Frequencies of input jazz commands 157

15.2 General input jazzing classification (principal uses) 158

15.3 Frequencies of output jazz commands 160

15.4 General output jazzing classification (principal uses) 160

18.1 Numbers of user-supplied parameters, NAG and IRENA 192

A.1 Principal contributions to IRENA code 200

D.1 Index of jazz commands 210

D.2 Keys in the **reshape-output** control list 222

Part I

Background and design issues

Chapter 1

Introduction

1.1 The IRENA project

IRENA was a joint project between the University of Bath and NAG, to provide an Interface from **REDUCE** (see [14]) to the **NAG** Fortran Library (see, for instance, [26]). It had the dual objectives of providing a common environment for symbolic and numeric computation and of simplifying the use of the NAG Library by providing it with a more mathematical, less Fortran specific interface. This is discussed in greater detail in [4], [5], [6] and [32].

In particular, most of the systems code was developed by M.C. Dewar, as part of his Ph.D. thesis [5] at Bath – this was based on design work by both Dewar and the present author. Design features attributable to the present author will be described in later chapters. The main responsibilities of the present author were to generate IRENA user interfaces for a large number of NAG Fortran routines[1] – although, as will be seen, this also involved significant extensions to Dewar's original code – and to test the overall system[2].

In what follows, reference is made to IRENA-0, meaning the basic system, as developed by Dewar, and IRENA-1, the first version actually released by NAG. However, in the manner of

[1] Interfaces were originally developed for about 350 top-level NAG routines, of which 160 are included in the initial release of IRENA.

[2] In terms of code developed, this proved a fairly equitable division of labour, with Dewar providing about 0.59 Mbytes of source code in various languages, compared to Richardson's 0.54 Mbytes. A summary of the various items included in this total is presented in appendix A. The various terms used there will be met as this thesis develops and are also summarised in the glossary.

all cooperative projects, ours was the subject of simultaneous developments on both fronts. Thus, most of the extensions mentioned above were made to a basic system which was still under development, so that IRENA-0 represents, at most, a conceptual stage in IRENA's evolution. The term is used here as a convenient fiction, representing that part of the IRENA code developed (but not entirely specified) by Dewar.

1.2 Experience with Naglink

The "Naglink" package, developed by Broughan (see [2]), was an early attempt to produce a symbolic-numeric interface system, using Macsyma as the symbolic engine and, initially, the NAG Mark 11 Fortran Library [20] for the numerics. Since NAG was intended to market this package, the present author spent a considerable length of time working with Naglink, prior to the start of the IRENA project.

Early attempts by the present author to develop demonstrations of Naglink revealed a number of deficiencies, which convinced him of the need for thorough in-house testing of this package's facilities. Consequently, he embarked on a programme of translating the example programs, which form part of the description of each routine in the NAG Library Manual, into Macsyma/Naglink code. This involved translating approximately 500 Fortran programs and was his major occupation for about a year.

In Naglink, most routines followed the NAG parameterisation[3], although in some instances different parameterisations were introduced on an *ad hoc* basis. For instance, in some routines individual end-points of ranges of integration had to be provided as separate parameters, in others, an interval representing the range was required. Similarly, for some routines, the NAG naming conventions were followed, for others, they were not. Since the documentation of Naglink largely consisted of a subset of the material in the NAG Manual and, in general, only indicated

[3]However, input parameters defining dimensions of arrays and the like, which could be readily determined within Macsyma, were not required. For some NAG parameters, defaults were provided; these were not included as part of the function call but could be reset in the Macsyma environment, where they existed as *pre-variables*. On output, most Naglink functions returned that NAG parameter deemed the most important, with other output parameters available in the Macsyma environment as *post-variables*; where several NAG parameters were required to define the main result, these were returned in a list. The use of NAG parameter names meant that when an input/output parameter was handled as pre- and post-variables, the default setting was lost after the function call, so could not immediately be used on a subsequent call. The one general exception to this was the NAG error handling parameter `IFAIL`, whose input value indicates the action to be taken on encountering an error and whose output value is an error index; the input form of this was renamed `softfail`.

the names and types of parameters, it was essential to consult the full NAG Manual and often use considerable ingenuity in interpreting it, in order to understand the purpose of the Naglink parameters.

Although, as will be seen in later chapters, the NAG Library's own interfaces are not always entirely consistent – so that redefining them is certainly a legitimate activity – the author's experience with Naglink convinced him of the need for a much more systematic approach in both providing and documenting symbolic interfaces to NAG routines. This eventually had a significant impact on the design of IRENA. A comparison of NAG, Naglink and IRENA parameterisations of a specimen routine is given in appendix B, which illustrates some of the problems addressed by IRENA.

It should be stressed that the author's experience of Naglink also had positive aspects; features which influenced IRENA included the notion of environmental values, allowing parameters to take their value from settings of global variables, and range parameters which were generalised in IRENA as "rectangles".

1.3 IRENA-0 design activity

The IRENA project began with a period of intense consultation between Dewar and the present author, in which they attempted to define the prerequisites for a successful symbolic interface package for the NAG Library. Here, the initial "division of labour" between IRENA's two authors was also determined, with Dewar largely being responsible for developing the system code, in the framework of REDUCE, and Richardson having responsibility for the individual interfaces to NAG routines and the provision of test examples for these interfaces.

One area in which the present author contributed to the design of IRENA-0 was that of data representation. This was particularly influenced by his knowledge of the rather unsystematic representation of various types of matrices in the NAG Library and led to the specification of a set of types which would cover matrix usage in the Library, and to the jazz system for interface redefinition (see sections 3.1, 7.2 and 7.3, chapter 11 and appendix D). A further contribution was in the specification of the transformations which would be required in the defaults system, to convert information derivable from a subset of NAG parameters into values for others. These features are described in the succeeding chapters.

It must be stressed that the detailed design and implementation of these areas was Dewar's; Richardson's contribution was largely limited to a specification of required functionality, with occasional contributions to the code.

Later in the development of IRENA, it became apparent that there was a need for a simpler extension mechanism for jazzing, since additional jazz functions were frequently required to provide special transformations for only a small number of NAG routines' parameters. This mechanism, described in section 7.3, was designed and implemented by Dewar and widely used by the present author to program new jazz functionality.

1.4 Basic design considerations

The basic philosophy driving the design of user interfaces in IRENA was to make the use of the NAG Library's numerical analysis routines as simple as possible[4]. For the present stage of the project, this was to be accomplished largely through interfaces to single routines or closely related sets of routines – it was envisaged that building higher level interfaces to entire numerical analysis subject areas would form further projects[5], built on this foundation (see, however, section 13.1). Reasons for this approach included

- work which had already been carried out on automatic routine selection had indicated that this was still a very open area – see for instance [34];
- the large number and variety of NAG routines suggested that even the development of individual interfaces would be a major project and
- commercial considerations clearly indicated that an incremental approach to the development, with exploitable intermediate products, was to be preferred to an open ended commitment to producing a fully comprehensive system.

[4] Simplicity of the interfaces, in turn, allows their documentation to be much more straightforward. The complexity of the NAG documentation, particularly for those who are not native English speakers, has been remarked on by a number of users of the Library: some instances of documentation which could be improved may be found in chapter 2 and in sections 4.2, 11.1.2, 16.1.8 and 16.4.1; an area where IRENA has already led directly to improvements in NAG documentation is described in section 7.1.

[5] One such project now being undertaken at Bath is investigating an expert systems approach to the selection of numerical routines, which may then be executed using technology similar to IRENA's. This is described in [10].

At the inception of the project, the most widely used and extensive NAG Library was the Fortran 77 Library and so this was chosen as the potentially most useful subject for the development of simpler user interfaces. The contents of the Library fall into two main areas – numerical analysis and statistics. It fairly soon became obvious that the task of developing interfaces to all of the routines in the Library would be impracticable on a reasonable time scale and, since natural interfaces for much of the statistical content already existed in the form of statistical packages such as Genstat and GLIM, [11] and [30], it was decided that the statistical routines would be omitted from IRENA. (An even more restricted set of routines, based on the later "Foundation Library", was eventually chosen for the first release: this is discussed in section 8.1.)

Chapter 2

Design philosophy

Shneiderman, on page 143 of [35], gives six "basic goals of language design":

(B1) precision

(B2) compactness

(B3) ease of writing and reading

(B4) speed of learning

(B5) simplicity to reduce errors

(B6) ease of retention over time

and six "higher level goals"

(H1) close correspondence between reality and the notation

(H2) convenience of carrying out manipulations relevant to users' tasks

(H3) compatibility with existing notations

(H4) flexibility to accommodate novice and expert users

(H5) expressiveness to encourage creativity

(H6) visual appeal

(the labels B1 to B6 and H1 to H6 are the present author's).

The higher level goals, in particular, correspond quite closely to our original objectives in building IRENA, although these were not expressed in precisely Shneiderman's terms. For instance, one of our high level goals was to provide a "more mathematical" interface to the NAG Fortran Library, with parameters reflecting mathematical rather than computing constructs: this reflects all of H1, H2 and H3. H4 may be seen in our decision to provide default values wherever possible for NAG parameters, whilst also providing users with a simple means to override these defaults, wherever this was meaningful. Goals H5 and H6 are more problematical for a system such as IRENA; however, a system which empowers the use of a major body of technical software, such as the NAG Library, by a less technically sophisticated audience should, one hopes, encourage the creativity of that audience; the question of visual appeal is, perhaps, marginal for a command driven system[1] but, even here, it could be argued that an IRENA function call, with a few parameters identified by keywords, is more visually appealing than its Fortran 77 equivalent, with its multiplicity of positional parameters.

(The goal of providing a truly mathematical interface will only be fully realised when many more "multi-routine" IRENA functions are built, corresponding to classes of NAG routines handling related problems. For instance, a single interface for all integration routines could provide a general `numeric_integrate` function, instead of separate functions which reflect the eleven quadrature routines in the Foundation Library – or, potentially, the 25 in the full library. At present, to integrate the function e^{-x^2}, say, from $x = 1$ to $x = 2$, could be achieved in IRENA using `d01ajf(range=[1:2], f(x)=exp(-x^2));` whereas integrating the same function from $x = 1$ to $x = \infty$ requires the call `d01amf(range=[1:*], f(x)=exp(-x^2));` utilising a different function. Such higher level system integration is a longer term goal of the research project[2] of which IRENA forms a necessary building block; although not fully realisable at this stage, the objectives listed below were nevertheless a major consideration in the design of the individual interfaces for IRENA and were, to a considerable extent, realised, as can be seen in the example at the end of this chapter.)

[1] However, this was a consideration in the design of the later Axiom-NAG interface, NAGlink, in which the principal input mechanism provided for the initial release, as suggested by the present author, was the use of Axiom's visual templates. See also section 17.1.

[2] See also [10].

To achieve ease of use in the IRENA interfaces, the author attempted to provide parameterisations with the following properties (the relevant Shneiderman goals are appended in parentheses):

- informative:
  - the meaning of parameters should be clear (B1, B5, H5) and
  - the usage of each routine should be easily learnt (B4, H1);
- regular:
  - similar data items in the same or different routines should be similarly parameterised (B4, B6);
- orthogonal:
  - distinct items of information should be kept separate (B3, B4);
- minimal:
  - data should be simple to input (B2, B3),
  - information should only be obtained when required (B2, H1),
  - proliferation of parameters should be avoided (B3).

The underlying NAG routines can fail to meet these criteria to varying degrees, as shown in the following examples (from the Mark 15 Library).

Informative naming

Due in part to the restriction in Fortran 77 of the length of names to six characters, some names in NAG routines are very cryptic: in the routine `D01GCF`, a set of "optimal coefficients" may be provided in an array `VK`. This name may have some explanation in the underlying theory but, if so, this is not apparent from the description. In IRENA, any structure (including this one) in which a set of coefficients is to be supplied is called `coefficients`. Another example is the parameter `KPLUS1`, mentioned under "Minimality" below: this is one more than the degree of the required polynomial approximation to a set of data points. In IRENA, this is replaced by a parameter `Maximum degree of polynomial fit required`, aliased as `degree` (and, for the benefit of those familiar with the NAG routine, as `k`), from which its value is inferred.

Regularity

In five routines, the parameter specifying the required tolerance in the location of a solution is called **EPS**, in five others it is called **XTOL**: in IRENA this parameter is called **location tolerance** (and aliased as **loctol**) throughout. Another instance of IRENA's enhanced regularity will be found in the uniform treatment of finite and infinite intervals, mentioned under "Orthogonality", below.

Minimality

Throughout the Library, there any many parameters which specify the length of input arrays. In principle, these lengths can be deduced from the size of the actual dataset and, in IRENA, this is done. However, in Fortran 77, a separate parameter is required to allow the dimension of an array to be declared within a routine. In some routines, extra parameters allow the use of output arrays whose dimensions do not match those of the data – for instance, in the polynomial interpolation routines **E02ADF** and **E02AGF**, the parameter **NROWS** specifies the leading dimension of the output array **A**; for the matrix stored in this array, however, the corresponding dimension is always given by the NAG parameter **KPLUS1** and, since IRENA creates its own output structures, the **NROWS** parameter is eliminated in the IRENA interface (and given the value **KPLUS1** in the Fortran code which IRENA generates).

Orthogonality

NAG routines commonly pack several objects into a single array. An extreme example of this is the parameter **W** in **D02YAF**, whose description in the NAG manual was as follows (the **IW2** mentioned here is an input parameter):

> Before entry, W(I,1), I = 1,2,...,N must contain the derivative of Y(I) at T = X.
>
> On exit, W(I,1), I = 1,2,...,N is unchanged; W(I,2), I = 1,2,...,N contains the entry value of Y(I); W(I,3) = W(I,1), I = 1,2,...,N. The exit values of W(I,J), I = 1,2,...,N, J.GT.3 depend on the value of IW2. The possible values are IW2 = 4, IW2 = 6 and IW2.GE.7. If IW2 = 4, W(I,4), I = 1,2,...,N is used as working space.
>
> If IW2 = 6, W(I,4), I = 1,2,...,N contains a local error estimate for the solution obtained from Euler's method on the step T = X to T = X + H; W(I,5), I = 1,2,...,N contains an estimate of the local error in Y(I) obtained from Merson's method on the step T = X to LT = X + H; W(I,6), I = 1,2,...,N contains a marker to indicate

the significance of the error estimate contained in W(I,5) – if W(I,6) = 0.0, then W(I,5) is to be considered as a significant error estimate, otherwise W(I,6) = 1.0.

If IW2.GE.7, W(I,4) and W(I,5), I = 1,2,...,N are unchanged on exit, whilst W(I,6) and W(I,7), I = 1,2,...,N play the role of W(I,5) and W(I,6), I = 1,2,...,N respectively in the discussion of the case IW2 = 6 above.

W(I,J), I = 1,2,...,N, J.GT.7 are unchanged on exit.

In IRENA, all such arrays are "disentangled" into functionally distinct components.

NAG scalar parameters may also serve more than one rôle – usually with particular values used as flags. An example is the parameter **NP** in **D02GAF** which, if greater than three, specifies the length of a (user-supplied) initial mesh over which an ordinary differential equation is to be integrated but, if zero, indicates that a default, equispaced mesh of length 4 is to be used. In IRENA, distinct optional parameters – **use default mesh** and **initial mesh** – handle these two rôles.

How the criteria were met in IRENA will be discussed in detail in chapter 5 – that they were met is exemplified in the calls to **d01ajf** and **d01amf** above, in which the parameter names **range** and **integrand** (whose alias **f** was used in the calls) are certainly informative, naming the precise mathematical objects which they represent, are regular across these routines (and indeed across all of the quadrature routines) and are minimal, with two required input parameters rather than the eight of each of the NAG routines. The NAG routines are not so regular – **D01AJF** specifies the range of integration by two parameters giving its endpoints, whereas **D01AMF**[3] uses a method which also serves to illustrate lack of parameter orthogonality: here, the range is specified by means of two parameters, **INF** and **BOUND** with **INF** indicating the type of range (**-1** meaning an infinite lower bound, **1** meaning an infinite upper bound and **2** meaning both bounds infinite) and **BOUND** giving the numerical value of the finite bound, whether upper or lower. Thus, **BOUND** can represent one of two distinct quantities, depending on the value of **INF** – the exposition of this in the NAG documentation is further obscured by **INF** being located before **BOUND** in the routine's parameter list and, therefore, its description. This may be contrasted with the IRENA parameterisation in which a single parameter **range** covers all cases, with "unbound" being represented by an asterisk (*) – a general convention throughout IRENA.

[3] An even more complicated parameterisation is used in the routine **D01BBF**; details of this and how it is rationalised in IRENA will be found in sections 10.1.4 and 11.1.4.

Chapter 3

Outline of the IRENA system design

3.1 Overall design

The overall design of the IRENA system is discussed in detail in [5]. However, it is probably worthwhile to recapitulate some of the major features of the design here, to provide a basis of terminology for later use.

IRENA consists of an extended version of the REDUCE computer algebra system (see, for example, [14], [19] or [31]), in which a REDUCE function is supplied to provide an interface to each included NAG routine[1].

Each IRENA-function simply consists of a call to a single, standard interface function called "interface" (written in the REDUCE system language, RLISP) to which the name of the NAG routine is a parameter; all have associated code causing uniform IRENA conventions to be obeyed in their parsing, allowing, for example, a keyword syntax to be used.

Results are returned to REDUCE using that package's Standard Lisp based foreign function interface system, *oload*. Unfortunately, this can handle at most five parameters, whilst NAG routines often return more than five objects: thus, some form of packing is required to make

[1] In what follows, in order to distinguish these functions from other utilities provided by IRENA, the interface functions are referred to as *IRENA-functions*.

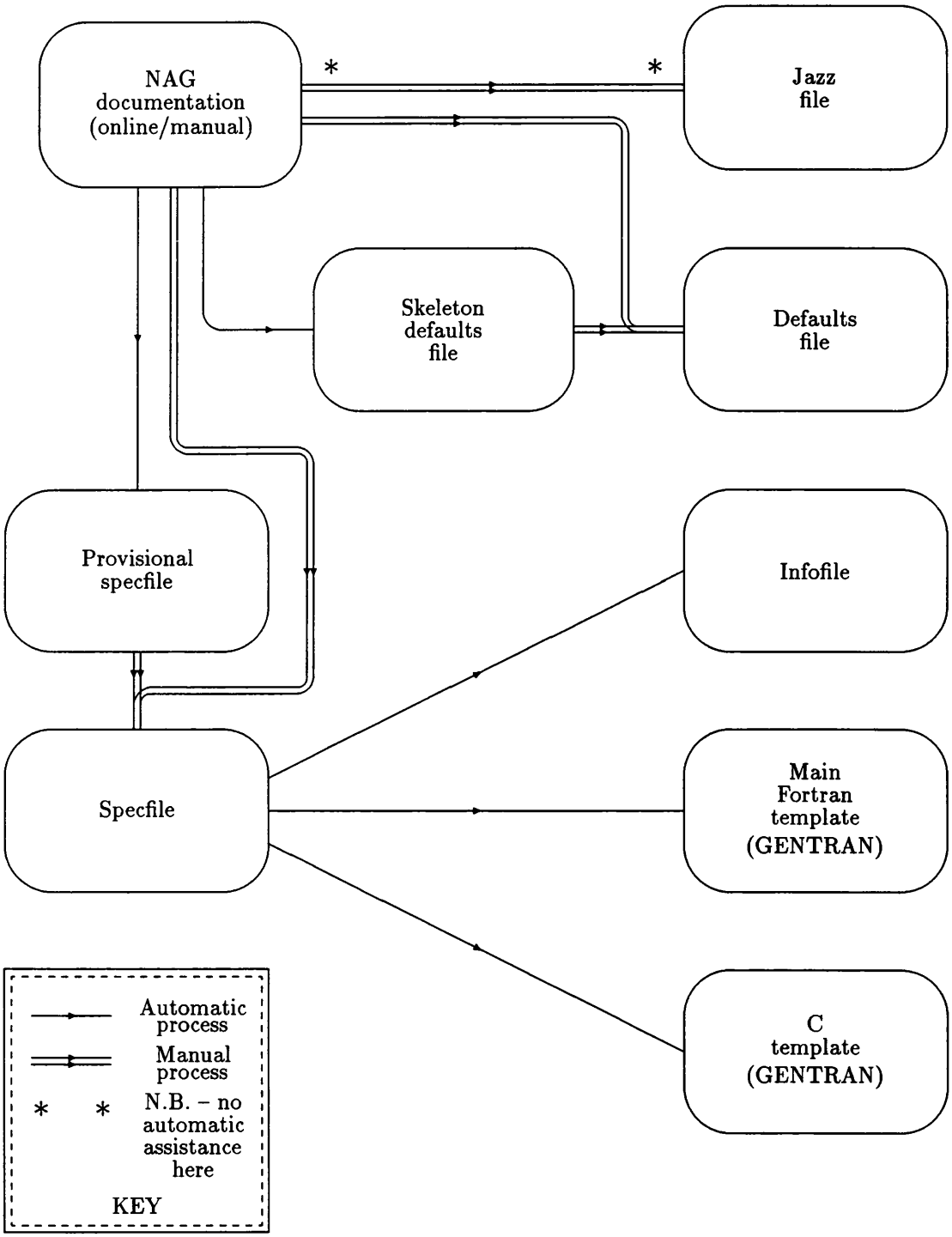

Figure 3-1: Information flow in standard IRENA setup processes

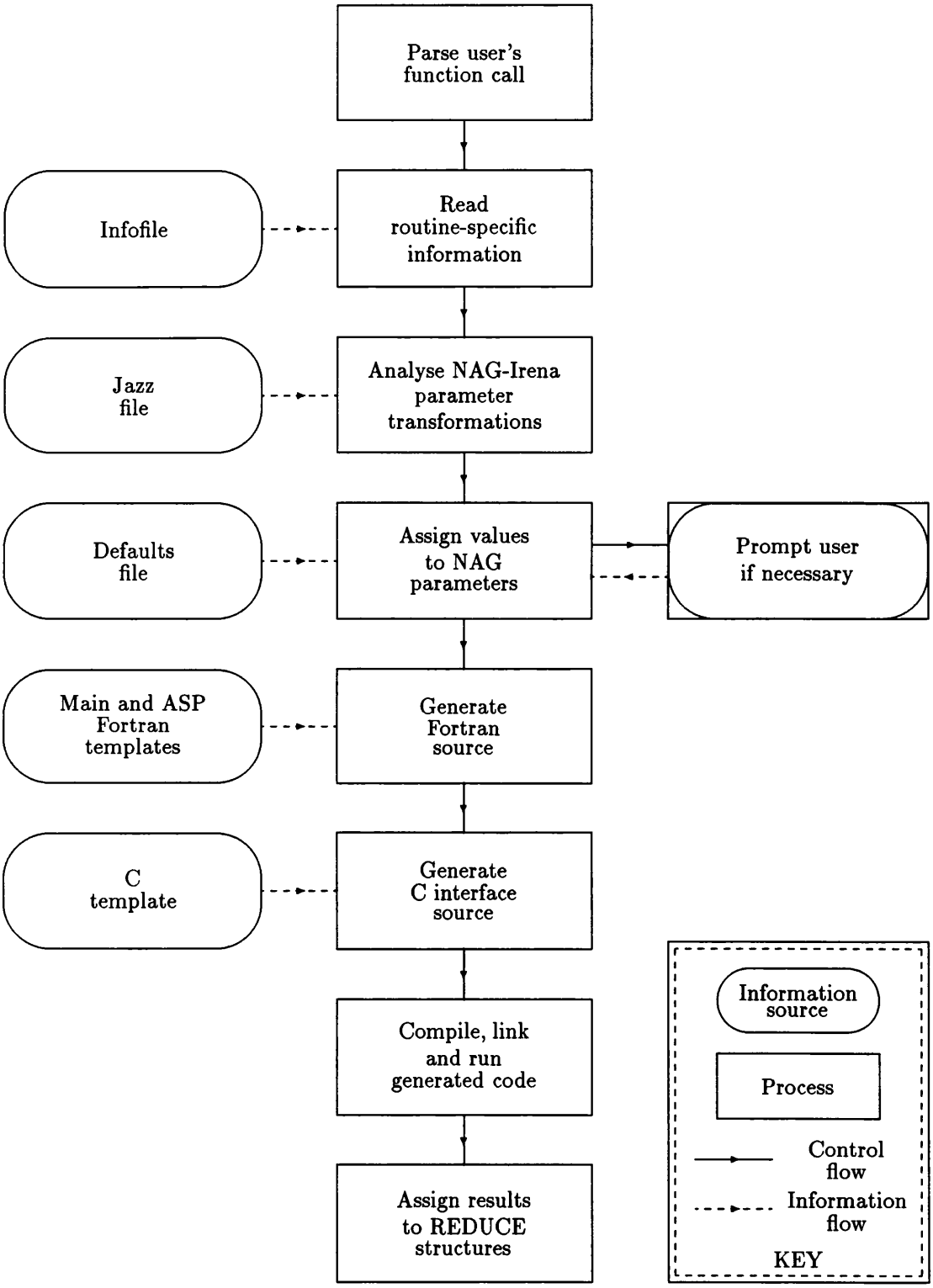

Figure 3-2: Outline of run-time IRENA-function processing

the results of NAG Fortran calls available to oload. Essentially, this was achieved by regarding the values returned by the Fortran routine as a single array, having previously generated an associated array of pointers which indicate the start positions of the various Fortran output parameters. Since Fortran does not provide a natural means of generating such an array of pointers, a small C interface was used to provide this functionality. Additionally, the C interface provides error handling facilities, should Fortran run time failures occur.

Among other actions, the RLISP "interface" function

- reads a routine-specific information file (the *infofile*) which provides details of the parameters of the NAG routine,
- reads a *jazz file*, which defines the mappings between NAG and IRENA parameters,
- assigns values to the NAG parameters by interpreting the value associated with each key in the IRENA-function call, by taking values from a *defaults file* and, possibly, by taking the values of REDUCE global variables and by prompting the user,
- uses the REDUCE *GENTRAN* subsystem (see [12]) to generate both the Fortran code to call the appropriate NAG routine and the C interface to link this to the parent REDUCE session,
- uses GENTRAN to generate Fortran code for any subprograms called as parameters by the NAG routine,
- compiles the Fortran and C code,
- uses the REDUCE oload system to load the compiled code into the parent session,
- runs the compiled code,
- transfers the generated results into the appropriate output objects and
- displays a list of output object names.

As well as the code to carry out the above actions, IRENA comprises a number of utility functions which will be described later, for the definition and manipulation of a variety of data structures, and the various special files mentioned here: the infofiles, the jazz files, the defaults files and the C and Fortran templates.

The infofile and templates for each routine are generated automatically from a more general specification in the *specfile*, a provisional version of which is itself generated automatically

from the NAG documentation database. This two-stage process allows anomalies in the generated material to be corrected at a single location, the specfile. Such anomalies can arise from typographical errors in the NAG documentation – for instance, in a number of cases, commas missing at line ends in NAG parameter lists caused the specfile generating program to malfunction – or may be due to anticipated exceptions such as unrecognised or misclassified ASPs ("Argument SubPrograms" – see section 3.2). Tailoring of the specfile was also undertaken when alternatives to the standard NAG routines were utilised (see section 12.3).

The flow of information during the generation of the various files mentioned here and the use of that information by a running IRENA-function are shown in figures 3-1 and 3-2, respectively. Not shown in the former is the generation of templates for those subprograms called as parameters of the NAG routines, since this was not usually done on a routine-by-routine basis. This is discussed further in section 3.2.

Jazzing, the (manual) specification of the required conversion between the NAG and IRENA parameter representations, and defaults file specification together formed a major component of the author's contribution to IRENA. An important but less time-consuming activity was the generation of the definitive specfiles: this was originally intended as a once only activity but in practice it had to be repeated whenever a new mark of the NAG Library was released, both to process new routines and to identify instances where parameters were reclassified (for instance from workspace to output or from input to dummy – see chapter 4); also, as already mentioned, individual specfiles were reprocessed when alternatives to NAG routines were introduced.

3.2 GENTRAN use

The application of GENTRAN in IRENA deserves some further mention, as it touches on areas of some difficulty in providing symbolic-numeric interfaces.

To quote the GENTRAN User's Manual [13]:

> GENTRAN is an automatic code GENerator and TRANslator which runs under REDUCE and VAXIMA. It constructs complete numerical programs based on sets of algorithmic specifications and symbolic expressions. Formatted FORTRAN, RATFOR or C code can be generated through a series of interactive commands or under the control of a template processing routine.

As mentioned in the previous section and shown in figure 3-1, GENTRAN templates are used to generate both the Fortran program corresponding to an IRENA-function call and the C interface which returns the Fortran results to REDUCE. Many NAG routines have parameters which are themselves subprograms (that is, Fortran functions or subroutines). These are referred to in IRENA terminology as *ASPs* (*A*rgument *S*ub*P*rograms). As well as code to call the NAG routine, IRENA must provide the Fortran code which defines the ASPs.

Unfortunately, there is great variety among ASPs – Dewar, in [5], mentions the existence of over seventy types and this number continues to grow, with eighty utilised in IRENA-1. By means of some fairly complex manipulations, Dewar managed to represent the majority of these with only ten GENTRAN templates, although, in IRENA-1, a further nineteen templates are required for individual ASP types.

In producing the IRENA specfiles, an automatic classification of ASPs is attempted, by pattern matching techniques applied to the description of the ASP in the NAG library documentation, using a database built up (by hand) from previous known instances. However, this procedure is somewhat error-prone: at times it misclassifies ASPs, leading to errors in the run-time system which are difficult to diagnose; also, at times, slight changes in the NAG description, between releases of the library, have prevented recognition of the types of previously classified ASPs.

3.2.1 Gentranopt

GENTRAN provides a switch, **gentranopt**, which is designed to optimise the generated Fortran source code, by recognising common sub-expressions and assigning these to intermediate variables. Initially, IRENA was set up with this facility enabled but, as illegal Fortran was sometimes found to result, **gentranopt** was set to default to **off** in the released version. In many cases this should cause no significant deterioration in IRENA's efficiency, since modern Fortran compilers usually provide an equivalent facility.

(In later releases of REDUCE a fuller optimisation may be performed, using the SCOPE package. See [15].)

Chapter 4

Classification of NAG parameters

4.1 Classification in NAG Library documentation

In the NAG Fortran Library manuals, [24] and [26], each parameter of any NAG Fortran Library routine is classified as input, output, input-output, external procedure, workspace, user workspace or dummy.

In IRENA, the input and output rôles of input-output parameters were treated separately, in order to preserve input data objects.

External procedures (ASPs, in IRENA terminology) are generated automatically from mathematical objects, each of which serves as an input parameter for IRENA.

Workspace arrays are provided automatically in IRENA, generally with no intervention from the user, since in most cases any sufficiently large workspace will suffice. (For a few, exceptional, routines, notably in the `D01` – integration – chapter[1] of the NAG Library, the length of a workspace array can affect the behaviour of the algorithm used, by controlling the degree of

[1] NAG Fortran 77 routines are classified into *chapters*, such as `D01`, following the scheme of the ACM modified SHARE classification index [1]. Routines in each chapter are named by following this alphabetic-numeric-numeric prefix by two further letters and a final letter specifying the type of library – in our case an `F`, indicating "standard precision Fortran".

subdivision available for quadrature[2]: for these routines only, users can easily override the default workspace length.)

User workspace arrays exist in a few NAG routines to provide an alternative to **COMMON** blocks for communication between the user's program and user-supplied external procedures. In IRENA, since the user workspace parameters did not have any obvious rôle in automatically generated code, they were treated as (redundant) workspace parameters of minimal size – that is, of dimension 1.

Dummy parameters have no effect on the functioning of NAG routines; they are occasionally found in, for instance, routines which have been internally revised between releases of the NAG Library, serving to maintain the previous interface (in which they would have had an input function). Apart from appearing in the routine call in the generated Fortran, they are essentially ignored in IRENA.

4.2 Functional classification of NAG input parameters

Examination of the input parameters of a typical Fortran 77 routine shows that these can be classified into three main types (although, as we shall see later, the boundaries between these are sometimes vague):

- **data** parameters, which define the problem to be solved,
- **control** parameters, which control aspects of the solution process and
- **housekeeping** parameters, which are logically dependent on other parameters but which are included either to meet the requirements of the Fortran language or for its more efficient use.

For example, the routine **C02AFF**, which determines the zeroes of a complex polynomial, has three purely input parameters: **A**, the array of polynomial coefficients (*data*), **N**, the degree of the polynomial (derivable from **A** and so classified as *housekeeping*) and **SCALE**, which indicates

[2]The NAG usage here fails to meet at least two of the IRENA design objectives – it is neither informative nor orthogonal. Although this is not expected to be a frequently used feature, IRENA attempts to improve the situation on the former count slightly, by using as the full name of the parameter **main workspace length (restricts subdivision)**, instead of NAG's **LW**, to indicate why this may be necessary. At a later release it may be feasible to introduce instead a parameter which explicitly controls the level of subdivision and from which the length of this array may be derived by IRENA.

whether or not automatic scaling should be performed on **A** to avoid over- and underflow when the coefficients differ by a large factor (*control*). It can be seen, from the example IRENA-function description document in appendix C, that the equivalent of **A** is an "essential input parameter" for IRENA, the equivalent of **SCALE** is an "optional input parameter" and there is no equivalent for **N**.

Data parameters require little further explanation – they include such possibilities as

- the coefficients of a set of simultaneous equations which is to be solved,
- the matrix of points to which an approximating curve is to be fitted and
- the function whose integral is to be approximated, together with
- the range of integration.

Control parameters appear in many NAG routines: common types are

- switches, determining which of a number of calculations the routine should perform or which of a choice of strategies it should adopt,
- convergence criteria, in the form of acceptable error levels,
- iteration limits,
- monitoring levels (for intermediate output) and
- **IFAIL**, a parameter whose input value determines the action to be taken in the event of failures detected within the routine.

(**IFAIL** also has an important output rôle, in returning a coded indication of routine-detected errors.)

The most common housekeeping parameters are the dimensions of data arrays. The dimensions necessary for workspace arrays can also normally be determined from other dimensions of the problem (but see below) and, in this case, should be regarded as housekeeping.

One practical significance of this classification is the treatment of default values. This concept is meaningless for (pure) data parameters, for which the user should always be expected to provide a value. Suitable defaults often exist for control parameters but should be easily overridden. In contrast, housekeeping parameters' values may always be determined mechanically and there

should never be a need for the user to override these. In IRENA-1, default values for control parameters and values for housekeeping parameters are provided in each routine's *defaults file.* (These "values" may, in fact, be quite complicated formulae depending on the actual values of other parameters.)

As indicated earlier, a parameter may exhibit features of more than one of the above categories – examples are

- weights in fitting routines which, in being either all equal or otherwise, act as control parameters distinguishing unweighted from weighted fitting but, when unequal, form part of the data and
- lengths of workspace parameters which, occasionally, act as control parameters by limiting the number of iterations which are possible.

In such cases, the highest appropriate level of the hierarchy data > control > housekeeping must be accommodated.

In an arbitrary selection of eight of the routines[3] included in IRENA-1, the relative abundance of the three types of input parameters mentioned here (with the higher level being taken in doubtful cases) was 26 data parameters, 12 control parameters and 16 housekeeping parameters.

This set of routines provides several examples of alternative possible classification...

`INF` in `D01AMF` (a routine for integration over a semi-infinite or infinite range) specifies whether the range is left-, right- or doubly infinite and so could be thought of as a control parameter, determining the type of integration to be performed. However, it may also be thought of as specifying one or both of the actual endpoints of the range and so being a data parameter. A strong reason for adopting the latter interpretation is that it leads to a regular parameterisation (see page 9) of the range as an interval – or, in IRENA terminology a *rectangle* – in which finite endpoints are represented by numeric values and infinite endpoints by an asterisk (*) – the standard IRENA symbol for *unbounded.* This corresponds to the form adopted in IRENA for the range in routines for integration over finite ranges, as can be seen in the examples on page 8.

`W` in `E01BGF` – a weight parameter with both control and data aspects, as discussed above.

[3] `C02AFF`, `D01AMF`, `E01BGF`, `E02DDF`, `F01RCF`, `F04ASF`, `S13AAF` and `S17ADF` – actually, a paged sample, obtained by randomly selecting one routine among the first twenty and including every twentieth routine thereafter.

M in S17DLF (a routine for calculating Hankel functions) specifies the type of Hankel function – $H^{(1)}$ or $H^{(2)}$ – required and so could be regarded as control or data. Similarly

FNU and N in this routine specify the range of orders of Hankel functions required and

SCALE indicates whether scaled or unscaled Hankel functions are required.

In addition, the parameters LAMDA, NX, N, MU and WRK in E02DDF (which calculates a bicubic spline approximation to a set of scattered data) all contain data if START has the value W but are ignored (and so would be classed as housekeeping) if it has the value C. This routine also provides examples of two parameters misclassified in the manual [26] – WRK, which is described as workspace but which is actually input-output, and IWRK, also described as workspace, which may contain valuable diagnostic data in the case of a failure and so should be output. Finally, the same routine illustrates the difficulties which may be encountered in trying to extract information automatically from a source meant for human reading: the manual's description of NXEST and NYEST includes the paragraphs:

> In most practical situations, NXEST = NYEST = $4 + \sqrt{m/2}$ is sufficient. See also Section 8.3.
>
> *Constraint*: NXEST $\geq$ 8 and NYEST $\geq$ 8.

In fact, these two parameters determine the dimension of the matrix containing the coefficients of the spline approximation, which can be deduced from the description of another parameter, C. Section 8.3 provides the equivalent information that they are upper bounds on the numbers of knots in the x and y directions but gives no further advice about suitable values. In IRENA these are treated as control parameters, with default values equal to $\max(4 + \sqrt{m/2}, 8)$ (where m is obtained as the value of the corresponding NAG parameter, M). That they may not be housekeeping parameters becomes apparent in the section "Error Indicators and Warnings" which leaves open the possibility that the condition IFAIL = 3 may be caused by these default values being too small.

Chapter 5

Strategy for meeting the design objectives

Recalling that the design objectives for the IRENA interfaces to NAG routines were that these should be:

- informative
- regular
- orthogonal *and*
- minimal

we shall now examine, in somewhat more detail, the strategy used to achieve these.

5.1 Informativeness

This was expanded in chapter 2 to:

- the meaning of parameters should be clear and
- the usage of each routine should be easily learnt.

There is, of course, a tension between the requirements for clarity and simplicity, the former tending to make the names of parameters longer, the latter shorter. This was resolved by having

1. short, simple key-entries (keys and keywords) on input,
2. descriptive prompting when parameters are omitted,
3. optional prompting for defaulted parameters,
4. key-entries related to prompts,
5. optional user defined key-entries,
6. descriptive names for output objects,
7. a simple output accessing function to avoid typing long names and
8. optional user defined alternative output names.

Thus, for input, a "full" version of each key or keyword was used in prompting, assisting in comprehension and learning, with simple, clearly related abbreviations being available for keys and keywords. For output, informative names were again used but the ◐ output function (described in section 9.3.1) provides a simple, alternative means to refer to output objects. Users have, in addition, a simple means to define their own, alternative, input and output names.

5.2 Regularity

"Regularity" was defined to mean that different routines should be similarly parameterised.

In the NAG library, naming conventions, conventions on the storage of data and the style of problem decomposition all vary considerably between – and even within – chapters. (Some examples appear in table 5.1.)

In developing the individual routine interfaces, naming conventions were maintained by making periodic checks on the total collection of names in use, with ruthless pruning of redundant choices. (This discipline could usefully be adopted in any area where names occur in the user interfaces of collections of software.)

To overcome the irregularities in the NAG Library's data storage conventions, structures were defined which reflected the objects in use, with automatic conversion to NAG parameter requirements.

Uniformity of style of problem decomposition was achieved, in part, by the emphasis on orthogonality (see below).

The following table shows the number of occurrences, in all "jazzed" routines, of various names corresponding to a small selection of the parameter names used in IRENA. Several NAG names appearing on one line indicate that these parameters together carry the same information as the single corresponding IRENA parameter.

Input Names		
IRENA name	NAG names	frequency
`Coefficients`	`A`	15
	`C`	12
	`TRIG`	7
	`CR, CI, BR, BI, AR, AI`	1
	`D, E`	1
	`VK`	1
`Location tolerance`	`EPS`	5
	`XTOL`	5
`Starting point`	`X`	21
	`ELAM, Y`	1

Output Names		
IRENA name	NAG names	frequency
`Eigenvalue` or `eigenvalues` (as appropriate)	`R`	7
	`RR, RI`	5
	`ELAM`	1
	`ALFR, ALFI, BETA`	1

Table 5.1: Correspondence between IRENA and NAG parameter names

5.3 Orthogonality

Whereas NAG routines commonly pack several objects into a single array, IRENA aims for one object per structure.

NAG scalar parameters may also serve more than one rôle – usually with particular values used as flags. For example, as we saw on page 11, the parameter **NP** in **D02GAF** represents:

if **NP** > 3 the length of (user-supplied) initial mesh, over which the ODE is to be integrated;

if **NP** $= 0$ that a default, equispaced mesh of length 4 is to be used.

If the user does not supply a mesh, it is not clear whether IRENA should prompt or use the default. This is resolved by asking the user, so that **NP** gives rise to two IRENA parameters, one indicating whether or not to use the default, the other (possibly) specifying a mesh. (If the user supplies a mesh we can, of course, deduce both.)

Disentangling instances of non-orthogonality such as this requires a clear understanding – by a human agent – of the possible ways in which the facilities provided may be utilised, for appropriate jazzing of the routine in question to be achieved.

5.4 Minimality

Minimality of parameter requirements is achieved by:

- not requiring housekeeping parameters,
- providing defaults for control parameters,
- not requiring users to specify output parameters and
- merging parameters defining partial structures.

Two examples will illustrate the final point.

As Fortran-77 has no double length complex data type, corresponding to the real **DOUBLE PRECISION** type, NAG routines often store the real and imaginary elements of a complex structure in separate arrays. For instance, in the routine **C06ECF**, which computes the discrete

Fourier transform of a sequence of complex values, two one-dimensional arrays `X` and `Y` are used, on input, to hold the real and imaginary parts of the sequence and, on output, to hold these parts of the transform. In IRENA these are replaced by complex structures called, for input, `sequence` and, for output, `fourier_transform`[1].

In routines which are designed to handle sparse matrices, the non-zero elements and their locations are specified separately. For example, in the routine `F01BRF`, the elements are specified by the user in a one-dimensional array called `A` (a widely used NAG name for an arbitrary matrix). The row and column positions of these elements are specified in the one-dimensional arrays `IRN` and `ICN`, respectively. In IRENA, these three arrays are replaced by a single parameter (also known as `A`, for consistency with NAG practice). This parameter may be supplied either as a list of triples {*row-address*, *column-address*, *value*} or, mainly for the convenience of those who already have data in the NAG format, as the corresponding list of three lists.

[1] In fact, the situation is slightly more complicated, in that `C06ECF` can also, with the assistance of the routine `C06GCF`, calculate the inverse Fourier transform. In IRENA, this assistance is provided automatically by using a jacketed Fortran routine (see section 12.1) whenever the user specifies the keyword `inverse`. In this case, the IRENA output parameter is renamed `inverse_fourier_transform`.

Chapter 6

Parameter representation in IRENA

In a number of ways, the NAG parameterisation of data is not ideal for the representation of objects in a mathematical package. As well as lacking regularity and orthogonality, as touched on in chapter 5, the NAG parameters may, for largely historical considerations of efficiency, use lower level structures than might appear natural: for example, matrices are sometimes stored as one-dimensional objects with essential structural information (such as the dimensions or, for irregular band matrices, the band width for each row) being given separately.

A major contribution of the present author to the original IRENA design was to specify many of the requirements for data representation. Three specific areas of input data representation which grew out of this specification are described in this chapter, together with some aspects of output data handling.

Type	Representation
With row lists:	The uppermost row is first, throughout; the row and column indices are represented by i and j, respectively.
full	each inner list specifies a row
symmetric	each inner list specifies row elements for which $i \geq j$ *or* each inner list specifies row elements for which $i \leq j$
skew-symmetric	each inner list specifies row elements for which $i > j$ *or* each inner list specifies row elements for which $i < j$
Hermitian	each inner list specifies row elements for which $i \geq j$ *or* each inner list specifies row elements for which $i \leq j$
strict upper triangular	each inner list specifies row elements for which $i < j$ (final list empty)
upper triangular	each inner list specifies row elements for which $i \leq j$
upper Hessenberg	each inner list specifies row elements for which $i \leq j + 1$
strict lower triangular	each inner list specifies row elements for which $i > j$ (initial list empty)
lower triangular	each inner list specifies row elements for which $i \geq j$
lower Hessenberg	each inner list specifies row elements for which $i \geq j - 1$
general band (variable bandwidth)	each inner list specifies row elements, lying within the envelope, and is packed out with zeroes for symmetry about the diagonal
symmetric band (variable bandwidth)	each inner list specifies row elements, lying within the envelope, for which $i \geq j$
With diagonal lists:	The uppermost diagonal is first, throughout.
band (fixed bandwidth)	each inner list specifies a "diagonal"
symmetric band (fixed bandwidth)	only the superdiagonal and diagonal (or diagonal and subdiagonal) lists are given
Sparse:	
sparse	three inner lists, each in the 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
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 sparse	as sparse, restricted to either upper or lower triangle.
symmetric long sparse	as long sparse, restricted to either upper or lower triangle

Table 6.1: IRENA matrix representations

6.1 IRENA matrix representation

All matrix processing in IRENA is handled by functions (one for each matrix representation) which take two parameters, the matrix name and its value. Each function updates the appropriate property lists for the particular matrix. In addition, there are functions to convert IRENA matrices (of any type) to REDUCE matrices and REDUCE matrices to IRENA rectangular matrices.

All IRENA matrices are represented as lists of lists. In most cases, the inner lists represent all rows (or partial rows) in the natural order; more rarely they represent diagonals. Generally, the same function is used to introduce upper and lower forms, since the form can be detected automatically. For strict lower and strict upper triangular matrices, the first or last inner list, respectively, is empty; this empty list may optionally be omitted, for matrices whose order is more than two (as there is then no possibility of confusion between upper and lower matrices).

As mentioned in section 5.4, two representations of sparse matrices are allowed: the first emulates the NAG convention of using three separate arrays, by having a list of three lists; the second, "long", form uses a list of triples, to give a more natural representation. A full list of IRENA matrix types appears in table 6.1. There is, additionally, a vector type, represented as a single list. There is no separate "diagonal" type, as this can easily be represented by a band matrix: $\{a, b, c, d, \ldots, z\}$ has little advantage over $\{\{a, b, c, d, \ldots, z\}\}$.

The vector and matrix facilities of IRENA, including a number of further utilities provided by the present author, are fully described in appendix A of [33].

6.2 "Rectangular" regions

In a number of areas, particularly integration and constrained optimisation, a region of interest is defined by a pair of bounds in one or several dimensions. In NAG routines, such a region is usually specified by means of two scalars or two one-dimensional arrays, one containing the lower bounds and the other the upper bounds. In IRENA, we have the concept of a *rectangle*, in which upper and lower bounds occur together, separated by a colon. Pairs of bounds are delimited by commas and the whole is enclosed in square brackets. For example, translating the standard NAG example for `D01FCF` (in a system in which IRENA's monitoring is switched off and the precision of printing floating point numbers is set to six digits) gives:

```
1: d01fcf(f(z1,z2,z3,z4)=4*z1*z3^2*e^(2*z1*z3)/(1+z2+z4)^2,
1:         range=[0:1,0:1,0:1,0:1])$

{integral,relative_error_estimate,number_of_function_evaluations}

2: integral;

0.575362
```

Wherever it is meaningful, an asterisk[1] (*) may be used to indicate *unbounded* ($\pm\infty$). Thus, a call to the bounded optimisation routine **E04JAF** might appear as:

```
e04jaf(f(w,x,y,z)=(w + 10*x)^2 + 5*(y - z)^2 + (x - 2*y)^4 + 10*(w - z)^4,
       bounds=[1:3,-2:0,*:*,1:3], vec start {3,-1,0,1})$
```

Rectangles are set up in IRENA as REDUCE objects, so it is also possible to define them outside the keyline, for instance with

```
bounds := [1:3,-2:0,*:*,1:3];
```

6.3 Function families

6.3.1 User-defined functions

The only common situation in which the user must provide information to specify an ASP (see section 3.2) is when this defines a function or family of functions. In this case, defining the functions (using appropriate dummy arguments) in the IRENA-function call is sufficient to define the ASP. For example, continuing the session begun in the previous section, we can use

[1] REDUCE interprets the asterisk as **TIMES**. This is reinterpreted with an appropriate value by the jazz system when the IRENA-function is processed.

D01AJF to determine the integral

$$\int_0^{2\pi} x^2 \sin x \, dx$$

as follows[2]:

```
3: d01ajf(f(x)=x^2*sin(x),range=[0:2*pi])$

{integral,absolute_error_estimate,number_of_subintervals_used}

4: integral;

 - 39.4784
```

6.3.2 Single parameter function families

In some cases, it is necessary to specify a family of functions rather than a single function. For example, D02BBF integrates a system of n first order ordinary differential equations. The derivatives, which depend on the independent variable t and the n dependent variables x_1 to x_n, may be specified by the user as functions f1, ..., fn.

As a concrete example, consider the case of a simple harmonic oscillator, subject to damping which decays exponentially with time. Taking some simple values (10, 0, 0.5, 4 and 2.5) for the various physical parameters, the behaviour of the oscillator over the first 10 units of time and its state at time $t = 50$ may be obtained as follows:

```
5: d02bbf( range=[0:50],
5:         vec initial_values {10,0},
5:         f1(t,x1,x2)= x2,
5:         f2(t,x1,x2)= -0.5*x1 - 4*e^(-t/2.5)*x2,
5:         vec output_points {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} )$

{solution_point,solution,solution_at_output_points,error_control_used

 }
```

[2]The pi here is evaluated in the Fortran by means of a call to the NAG routine X01AAF, using the same mechanism as in defaults processing, discussed in section 7.1. That no specific action is required from the user to specify how π should be evaluated illustrates how symbolic and numeric components may be integrated to provide a natural interface.

```
6: solution_at_output_points;

[  8.95653      - 1.49471 ]
[                         ]
[  7.26678      - 1.85794 ]
[                         ]
[   5.2848      - 2.07993 ]
[                         ]
[  3.17412       - 2.1004 ]
[                         ]
[  1.17739      - 1.84286 ]
[                         ]
[ - 0.413047    - 1.29323 ]
[                         ]
[  - 1.33963    - 0.538857]
[                         ]
[  - 1.4842     0.233599  ]
[                         ]
[  - 0.94373    0.793941  ]
[                         ]
[ - 0.0287051   0.962179  ]


7: solution;

 [ - 0.0107723]
 [           ]
 [ - 0.871786 ]
```

In some problems, large families of related functions may occur and, in this case, it would be impracticable to specify each function separately. For this reason, IRENA allows such a family to be specified as a single entity – an *fset* – either in the keyline or at the REDUCE level[3].

The standard NAG example program for the routine **C05NBF** solves the tridiagonal set of equations

$$
\begin{aligned}
& (3-2x_1)x_1 - 2x_2 + 1 = 0 \\
-x_{i-1} + & (3-2x_i)x_i - 2x_{i+1} + 1 = 0 \qquad 2 \le i \le 8 \\
-x_8 + & (3-2x_9)x_9 + 1 = 0
\end{aligned}
$$

To solve this using IRENA requires functions to be defined representing the left-hand sides of these equations.

We can define these as fsets, as follows[4]:

```
8: fset f[1](x[1:9])     =          (3 - 2*x(1))*x(1) - 2*x(2)   + 1;

9: fset f[i=2:8](x[1:9]) = -x(i-1) + (3 - 2*x(i))*x(i) - 2*x(i+1) + 1;

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

If fsets are defined at the REDUCE level, as these were, they may be displayed in an expanded form by means of the **fdisplay** operator:

```
11: fdisplay f;
```

[3] In the case of function names with multiple subscripts, described in section 6.3.3, only the fset notation is available.

[4] Note that an fset definition uses the equals sign (=) rather then the REDUCE assignment operator (:=). This allows simpler internal processing, since REDUCE does not attempt to evaluate the expression on the left. It also serves to distinguish the definition from an assignment, since the fset does not exist as an object which can be accessed or manipulated directly by users: it should, perhaps, rather be regarded as resembling a REDUCE *rule*.

```
                        2
F[1]= - 2*X(2) - 2*X(1)  + 3*X(1) + 1


                        2
F[2]= - 2*X(3) - 2*X(2)  + 3*X(2) - X(1) + 1

.

                        2
F[3]= - 2*X(4) - 2*X(3)  + 3*X(3) - X(2) + 1


                        2
F[4]= - 2*X(5) - 2*X(4)  + 3*X(4) - X(3) + 1


                        2
F[5]= - 2*X(6) - 2*X(5)  + 3*X(5) - X(4) + 1


                        2
F[6]= - 2*X(7) - 2*X(6)  + 3*X(6) - X(5) + 1


                        2
F[7]= - 2*X(8) - 2*X(7)  + 3*X(7) - X(6) + 1


                        2
F[8]= - 2*X(9) - 2*X(8)  + 3*X(8) - X(7) + 1


              2
F[9]= - 2*X(9)  + 3*X(9) - X(8) + 1
```

Note that the two end functions, `f1` and `f9`, were also defined using fset notation. It is not possible to mix the suffixed (`fn`) and fset notations in defining a single family of functions.

An fset definition consists of the word `fset`, followed by an optional list of subscripts in square brackets, a list of parameters in parentheses, an equals sign and an expression.

The list of subscripts consists of individual integers, or ranges of integers of the form $m:n$, separated by commas.

The list of parameters consists of individual identifiers and sets of identifiers, separated by commas, where a set of identifiers has the form *name*`[`k`:`l`]`, k and l being integers.

The expression is any valid REDUCE arithmetic expression. It may include individual members of any sets of identifiers which appear to the left of the equals sign. These are denoted by the *name*, followed by an integer subscript, in parentheses.

The following version of the `C05NBF` example demonstrates the use of fsets in the IRENA keyline:

```
12: c05nbf(fset f[1](x[1:9])     =           (3-2*x(1)) * x(1) - 2*x(2)   + 1,
12:        fset f[j=2:8](x[1:9]) = -x(j-1) + (3-2*x(j)) * x(j) - 2*x(j+1) + 1,
12:        fset f[9](x[1:9])     = -x(8)   + (3-2*x(9)) * x(9)           + 1,
12:        vec start {-1,-1,-1,-1,-1,-1,-1,-1,-1} );

{zero,residuals,location_tolerance_used}

13: zero;

[ - 0.570655]
[           ]
[ - 0.681628]
[           ]
[ - 0.701732]
[           ]
[ - 0.704213]
[           ]
[ - 0.701369]
```

```
[          ]
[ - 0.691866]
[          ]
[ - 0.665792]
[          ]
[ - 0.596034]
[          ]
[ - 0.416412]
```

```
14: residuals;

[  0.00000000656011  ]
[                    ]
[ - 0.00000000417547 ]
[                    ]
[ - 0.00000000519317 ]
[                    ]
[ - 0.00000000239601 ]
[                    ]
[  0.00000000202249  ]
[                    ]
[  0.00000000481792  ]
[                    ]
[  0.0000000025795   ]
[                    ]
[ - 0.00000000388374 ]
[                    ]
[ - 0.000000000135886]
```

The usefulness of fsets is increased by the possibility of referring to REDUCE global variables, and in particular REDUCE matrices, in the defining expression. For instance, the NAG **E04FDF** example generates least-square estimates of X_1, X_2 and X_3 in the model

$$Y = X_1 + \frac{T_1}{X_2 T_2 + X_3 T_3}$$

from 15 sets of values of Y and the Ts. The user must provide a set of functions in which f_i calculates the residual for the ith set of data – that is, the difference between the observed Y and the value calculated from the observed Ts, expressed as a function of arbitrary Xs. Specifying the Y and T values in a REDUCE matrix allows the functions to be defined as an fset, as in the following[5]:

```
15: y := mat((0.14, 0.18, 0.22, 0.25, 0.29, 0.32, 0.35, 0.39,
15:                 0.37, 0.58, 0.73, 0.96, 1.34, 2.10, 4.39))$

16: t1 := mat((1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15))$

17: t2 := mat((15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1))$

18: t3 := mat((1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1))$

19: off asp!-loops;

20: fset residual[j=1:15](x1,x2,x3) =
20: x1 + t1(1,j)/(t2(1,j)*x2 + t3(1,j)*x3) - y(1,j)$

21: vec start {0.5,1,1.5}$
```

[5] The switch **asp-loops** is turned off at line 19. This is necessary if matrix elements are to be used in an fset definition, since this technique is incompatible with the normal IRENA mode of processing fsets, which builds Fortran loops to produce more compact code. With the switch left on, the individual elements of the various REDUCE matrices would not be evaluated in the generated Fortran. The data vectors here are defined using the REDUCE **mat** function, with each matrix consisting of a single row, since REDUCE has no distinct concept of a vector.
A certain lack of minimality and orthogonality was introduced into IRENA itself, here, as the connection between using matrix elements to define an fset (at the immediate user level) and the **asp-loops** switch which controls the style of Fortran generation (of less direct interest to many users) is not necessarily immediately apparent. Unfortunately, the present author only became aware of this particular design decision late in the development of IRENA, at which point it was not feasible to attempt its rectification.

```
22: e04fdf()$

{location_of_minimum,minimum_sum_of_squares,

 singular_values_of_estimated_jacobian_of_f,

 right_singular_vectors_of_estimated_jacobian_of_f}

23: minimum_sum_of_squares;

0.0082149

24: location_of_minimum;

[0.082411]
[       ]
[ 1.133  ]
[       ]
[ 2.3437 ]

25: on asp!-loops;
```

In this example, the **fset** functions were called **residual**, rather than **f**, reflecting the quantities which they actually calculate. In fact, either name may be used for the functions used by the corresponding ASP.

6.3.3 More general function families

Fsets are not restricted to a single subscript: occasionally doubly subscripted functions are required. We could, for example, define

```
fset g[i=1:2,j=1:3](x,y) = i*x^j/y;
```

– as can be seen from this example, the ranges of the various subscripts on the left of the definition are simply separated by commas.

It is also possible to use fset notation to define a single function, in the REDUCE environment, which can be recognised by IRENA, provided that the **envsearch** switch is **on** (the default). In this case, the subscripting information may be omitted completely – for instance:

```
fset h(x,y,z) = x^2 + y^2 + z^2;
```

6.4 Data input from files

It is possible to input a file of IRENA commands (as with any REDUCE commands) by using the REDUCE construct `in` *filename*. In addition to this, IRENA provides a facility whereby the value of any parameter may be stored in a file and substituted in the keyline or in response to a prompt by using the form `!?"`*filename*`"`[6]. For example, if the contents of the file `data/f02aaf.data.1` were

```
{{ 1,  2,  3,  4,  5},
    { 6,  7,  8,  9},
       {10, 11, 12},
          {13, 14},
             {15}}
```

then the eigenvalues of the symmetric matrix which these values represent could be obtained by means of the call `f02aaf(sym!-mat a !?"data/f02aaf.data.1")`.

6.5 Output naming

As well as using informative names for output parameters, IRENA provides a means for users to distinguish between identically named output parameters produced by different functions. This is accomplished by defining the full name of any output parameter (the *long form*) to consist of the name displayed in the function's output list (the *short form*), prefixed with the name of the function and a hyphen. The short form is, in fact, an alias, which may be reused by other routines. The long form, however, is preserved (until the next call to the function which generated it), so that the user retains access to the earlier result without having to take any special action.

[6] In this, the exclamation mark is present as REDUCE's *letteriser* – that is, as an escape character.

6.6 Non-parametric output

In a number of NAG routines, notably in the D02 (ordinary differential equations) chapter, intermediate values of the computed solution are printed at a set of points determined by a user-supplied subroutine. In an interactive environment such as IRENA, it is much more useful to users to have access to such results in a structure; this was achieved by writing the results to a temporary file and reading this back into REDUCE in the form of a matrix, which was then treated as an additional IRENA output parameter.

The ASP specifying the desired points was constructed automatically from an array of points specified by the user. (Users requiring a more powerful means of determining the set of output points – for example, in terms of the previous point and value – still have the option, as with any IRENA ASP, of specifying the required NAG subroutine directly, in a Fortran file. However, it is anticipated that in most cases the loss of generality in the IRENA ASP will be outweighed by its ease of use, at least in the initial investigation of a problem; should a more sophisticated subroutine later be required, that generated by IRENA is available to the user as a template; this is expected to be of particular use to those programmers who are not highly expert in Fortran.)

Chapter 7

Defaults and jazzing in IRENA-0

7.1 The defaults system

The IRENA "defaults" system allows appropriate default values for NAG routine parameters to be specified, as constants or functions of other parameters, by the system developers or the user[1]. Similar defaults may be defined for "quasi-NAG" parameters, introduced by the jazz commands **scalar** and **vector**.

The functionality of the defaults system was originally specified by the present author; apart from a few small additions by him (such as the **housekeeping** entry, the **sum** function and the rôle of **unset** as a unit), the detailed design and implementation was entirely Dewar's.

The defaults system allows for the presence of both system and user defaults files for each routine, with the latter taking precedence over the former (and runtime user-supplied values, of course, taking precedence over both).

The defaults for a routine are specified using a simple language with the following features:

- entries may be signalled as "housekeeping", to distinguish them from "control" parameters (see sections 4.2 and 9.2.6);
- as in REDUCE, comments may be introduced using the percent sign (%);

[1] One effect of the IRENA project has been the formalisation of how "suggested values" of parameters are presented in the NAG Fortran Library Manual. Whereas, previously, such suggestions were dispersed throughout the routines' descriptions, from Mark 14 on, a special section of the parameter description has been provided for this purpose.

Function	Value
`abs(X)`	absolute value of X
`dim(X)`	the length of the one-dimensional array X
`dim(X,1)`	the first dimension of the two-dimensional array X
`dim(X,2)`	the second dimension of the two-dimensional array X
	if the first parameter of **dim** evaluates to **unset** then the result is **unset**
`have(X)`	**true** if X has a value (including **unset**), defined in the keyline or the defaults file. **false** otherwise (**have(X) and X ~= unset** tests for an actual value)
`length(S)`	the length of the string S
`matrixp(X)`	**true** if X exists as a REDUCE matrix, **false** otherwise (Note that vector and matrix results are returned by IRENA as REDUCE matrices.)
`max(X)`	the largest element of the array X
`max(X1, ..., Xn)`	the largest number among $X1, \ldots, Xn$
`min(X)`	the smallest element of the array X
`min(X1, ..., Xn)`	the smallest number among $X1, \ldots, Xn$
`multiplicity(X)`	the number of functions called $X1, \ldots, Xn$ supplied in an IRENA-function call
`nth-root(X,Y)`	the nth root of X, where n is the integer part of Y
`params(X)`	the number of parameters of the function X supplied in an IRENA-function call
`rectanglep(X)`	**true** if X exists as an IRENA rectangle, **false** otherwise
`scalarp(X)`	**true** if X exists as a REDUCE scalar, **false** otherwise
`sum(X)`	the sum of the elements of the array X
`sum(X1, ..., Xn)`	the sum of the numbers $X1, \ldots, Xn$

For the forms **max(X)**, **min(X)** and **sum(X)** to work, the array X must exist (perhaps aliased) as a REDUCE or IRENA object. In a few cases, the NAG array may have been constructed from separate REDUCE or IRENA level components, by means of a "jazz-function" – see appendix D: in this case X exists only in the Fortran generated and so cannot be processed by IRENA.

For **max**, **min** and **sum**, **unset** acts as a unit: that is, parameters which evaluate to **unset** are ignored, unless or until there is a single parameter, in which case the value **unset** is returned.

Table 7.1: Functions available in defaults files

Name	Routine called	Value
asetsz	X02CAF	estimated active set size (paged environments) or else zero
defnad	X04AAF	Fortran unit number for advisory messages
defner	X04ABF	Fortran unit number for error messages
fpbase	X02BHF	the base used in the computer's arithmetic
fpdigs	X02BEF	the number of decimal digits which can be relied on in floating point numbers
fpemax	X02BLF	the maximum exponent in floating point numbers
fpemin	X02BKF	the minimum exponent in floating point numbers
fpeps	X02AJF	the smallest number which, added to 1, yields a number > 1
fphuge	X02ALF	the largest floating point number
fpprec	X02BJF	the precision (in base fpbase digits)
fprnge	X02AMF	the smallest positive floating point number z such that, for any x in $[z, 1/z]$, the following may be "safely" calculated: $-x$, $1/x$, SQRT(x), LOG(x), EXP(LOG(x)) and y**(LOG(x)/LOG(y)) for any y
fprnds	X02DJF	.TRUE. if rounding is always correct in the final bit, .FALSE. otherwise
fptiny	X02AKF	the smallest positive floating point number
maxint	X02BBF	the largest integer
pi	X01AAF	π
scmaxa	X02AHF	the largest number for which SIN and COS return a result with some meaningful accuracy
ufevnt	X02DAF	.FALSE. if underflowing numbers are simply set to zero, .TRUE. otherwise

Table 7.2: NAG constants available in defaults files

- arithmetic may be performed on the values of both NAG and "quasi-NAG" parameters;
- conditional values may be specified;
- antecedents of conditionals may involve relational operators and tests on the existence of parameter values;
- functions exist to provide the dimensions of arrays, the maximum and minimum of arrays or sets of values, the number of parameters in a user-specified function and the size of a set of related functions provided by the user;
- a special value **unset** takes account of the situation where a NAG parameter will not be accessed on a particular call; note that this is considered a valid value (for instance, in checking whether a value has been set);
- special symbols ***userabserr***, ***userrelerr***, ***usermixerr*** and ***userinputerr*** provide a second level default mechanism, in that they are set globally and used to specify parameter defaults. Their values may be reset at the REDUCE level by the user, thereby redefining default values throughout the system;
- a special value **canceldefault** allows a user defaults file to undo the effect of a setting in the system defaults file, without setting another value; this is distinct from **unset**.

A full list of the functions available in specifying defaults is given in table 7.1.

In addition to the special symbols mentioned above, IRENA provides a number of other symbols, representing mathematical and "machine" constants, which may be used in specifying defaults. These are implemented by inserting calls to appropriate NAG routines in the Fortran code generated by IRENA. They are listed in table 7.2. (Not all of these are used in IRENA system defaults files – see the footnote on page 77.)

Note that the definitions in table 7.2 relate to a Fortran environment. The "value" entries corresponding to **X02** routines are merely descriptions: precise definitions may be found in the **X02** chapter of the NAG Fortran Library Manual [26] or Foundation Library Reference Manual [24].

As already noted, the standard REDUCE commenting convention applies in defaults files – that is, any text occurring after a percent sign (%) on any line is ignored.

Annotated examples of system defaults files, chosen to illustrate various features of the defaults system, will be discussed in chapter 10.

7.2 The basic jazz system

The IRENA "jazz" system is used to redefine the forms of NAG parameters, to help meet the objectives set out in chapter 5.

For each NAG routine within IRENA, there is an individual jazz file, containing descriptions of the conversions between NAG and IRENA parameters. Each entry in this file consists of a jazz command name, enclosed in curly brackets, generally followed by a list of NAG variable names, a colon and a list of the names of IRENA structures. The use of curly brackets was adopted to help visually distinguish the various parts of a jazz command and to serve as an implicit terminator for the previous command. Logically, it is not strictly necessary, since the components of IRENA lists are separated by commas; a new command may be recognised when an atom is encountered which is not preceded by a comma.

The mapping between the NAG and IRENA parameters is determined by the particular jazz command. There are conceptually two classes of jazzing, applicable to input and output parameters respectively.

For input parameters, with a very few exceptions, it is not necessary for users to adopt the jazzed form of parameters. Thus, a user who is familiar with the Fortran routine may continue to use the NAG parameter names and definitions[2] (although the jazzed names have the advantage of greater uniformity). Additionally, multiple jazzings of parameters are allowed, so that alternative interfaces may be designed in situations where there is more than one natural representation of a problem. In principle, tailored interfaces could also be provided for specific sets of users; initially, this will probably be limited to the choice of alternative, discipline-specific names for particular objects.

For output parameters, use of the jazzed form is obligatory, since it would be confusing to return the same object in several different ways. However, it is possible, where necessary, to define more than one output object containing the same information – for example, where a single element of an array contains information of interest to the user but the entire array is needed for input to a different NAG routine.

[2] In contrast, in the initial release of NAGlink, the Axiom-NAG link described in section 17.1, only a NAG-like parameterisation was provided, so that a working interface could be released more quickly. In the higher level NAGlink functions, provided by the present author for the second release, more natural parameterisations are used. However, the question of names does not arise there, except in documentation, since Axiom functions have purely positional parameters. In NAGlink, the alternative interfaces are provided by separate functions, so the higher and lower level features cannot be mixed in a single function call.

A limited subset of jazzing is available to users in the *alias* system, which allows alternatives to NAG or IRENA names to be defined by users. As a user may wish to have different names for input and output aspects of a NAG input/output parameter, two commands are provided for use in alias files: `in` provides an input alias and `out` an output alias. The syntax of these commands is similar to that used for jazz commands:

`{in}` *existingname* : *newname*

and

`{out}` *existingname* : *newname*

where *existingname* is a NAG or IRENA name for the object being aliased and *newname* is the alias which the user wishes to establish. Naturally, IRENA input and output names, respectively, must be used with `in` and `out`.

As with defaults files, the standard REDUCE commenting convention applies in jazz and alias files.

Brief descriptions of individual jazz commands may be found in appendix D and examples of their use are presented in chapter 11.

7.3 The extended jazz system

As mentioned in section 1.3, special transformations were often required for application to a small number of routines. Most commonly, these concerned NAG array parameters although, at times, scalars were also involved.

As the original jazzing mechanisms were rather deeply embedded in the IRENA systems code, Dewar was asked to provide a more easily extensible system for defining new jazz functionality. The necessary components for defining new input and output jazz commands are described below.

7.3.1 Jazz-functions

New input jazz commands can be defined by *jazz-functions*. These commands are slightly restricted in form, compared to general jazz commands, in that only a single NAG name is allowed. Thus, each instance represents a means of defining a single NAG parameter in terms of IRENA objects. On the other hand, rather than a simple list of IRENA structures, any valid Lisp object is allowed on the right of the colon.

A further restriction is that this system was not designed to produce REDUCE objects but only to generate Fortran code defining the NAG parameter: the intention behind this was to avoid creating large, temporary REDUCE structures, corresponding to NAG parameters, which the user would be unlikely to want to access.

To define a new jazz command, three REDUCE procedures were normally needed:
a *check-function*, to determine whether any necessary structures had not yet been provided,
a *dim-function*, to define the dimensions if the NAG parameter was an array *and*
the *jazz-function* itself, which generated Fortran assignments, defining the NAG parameter.
(In his thesis [5], Dewar refers to the jazz-functions as *trans-functions*.)

In cases where several NAG parameters were derived from one IRENA object, the various jazz-functions would all be associated with the same check-function.

The restriction to processing a single NAG parameter and the difficulty of generating REDUCE objects meant that the system was rather inefficient for processing single IRENA structures which represented several NAG parameters, since similar sections of code often had to be written for the various NAG parameters and the associated check-function would be invoked for each them.

In principle, it would have been possible to create intermediate REDUCE structures in the check-functions but, in practice, this did not prove a very satisfactory approach, since the logic of IRENA itself determines when a check-function is called, making it difficult to ensure that such objects are created before they are needed by other jazz commands.

7.3.2 Output-functions

New output jazzing functions are defined by *output-functions*. In this case, considerations of redundant REDUCE objects do not arise and the functions which can be defined are less restricted than general jazz commands, since, rather than a list of IRENA names, any valid Lisp list is allowed.

No associated functions are required in this case.

Part II

Development of IRENA-1

Chapter 8

Overview of IRENA-1 development

8.1 Choice of routines for IRENA-1

At the onset of the project, around 1987, NAG's principal numerical library was the Mark 12 Fortran Library, which contained 688 user-callable routines. At the time, it seemed a tractable task to define simplified interfaces for the majority of these routines, if not for them all.

However, as IRENA began to take shape and the practicalities of producing individual interfaces were better understood, the true scale of this task became apparent. Whereas the few simple routines initially considered as test cases had required little redefinition of their interfaces, when more complex routines were considered it soon became clear that the amount of work required to produce a natural interface to a routine increased more than linearly with its size (see section 8.3). What was perhaps more significant was that the incremental time to expand the processed set of routines increased considerably with the number of routines already processed, due to several factors:

- at times it was necessary to revisit earlier routines to adjust their jazzing, for stylistic consistency with what was found to be necessary for later additions;
- as the size of the set of processed routines grew, more errors were uncovered in REDUCE,

GENTRAN and IRENA's own system code; (errors were also discovered in the NAG Library and its documentation but these tended to be uncovered as each routine was processed); corrections for these errors and other modifications of the underlying software at times caused previously satisfactory jazzing to fail;

- new routines at times required extensions to jazzing facilities; again, this could require adjustments for earlier routines.

From each of these causes, the amount of extra work incurred by processing one further routine increased with the number already processed.

A further problem was that the NAG Fortran Library presented a moving target. The library is updated on a roughly 18 month cycle: not only are a significant number of routines replaced at each mark and further routines added, to provide new functionality, but corrections and refinements are made to the documentation, on which the generation, both automatic and manual, of IRENA components is based. An example of this is the reclassification of "workspace" parameters as "output", when some intermediate result stored there is found to have a practical use – this is perhaps one of the worst causes of "tangled" output, such as that described in section 2 for the parameter `W` in the routine `D02YAF` (the name `W` is a clue to the original use of this parameter as workspace).

After the obvious strategy of ignoring those routines which were "scheduled for withdrawal", the first reduction in the IRENA "target set" (for which individual interfaces would be produced) was the dropping of the statistical chapters of the NAG Library, for two main reasons:

- as mentioned in section 1.4, a natural interface already existed for much of this material in the form of statistical packages and
- the types of "natural" structures in statistics differ considerably from those which occur in numerical analysis, so that a disproportionate amount of effort was expected to be involved in catering for the relatively small number[1] of statistical routines.

However, as the project developed, the difficulties of "catching up" with even the non-statistical part of the NAG Library became ever more apparent and a decision was eventually taken to concentrate on the routines of the "NAG Workstation Library", which presented both a smaller

[1] This has remained fairly stable, at about 20% of the user-callable routines: 125 statistical routines from a total of 688 at mark 12 of the Library, 244 from a total of 1134 at mark 16.

and a more stable target set (the first release, in 1986, contained 112 non-statistical routines – see [21] – the second, which was in preparation when this decision was taken, was eventually released in 1992 as the "NAG Foundation Library" and contained 173 "fully documented" non-statistical routines).

As well as the statistical chapters, two chapters of NAG utilities (**X04** – Input/Output Utilities and **X05** – Date and Time Utilities) were excluded from IRENA as irrelevant or redundant in the REDUCE-IRENA environment. In addition, the routine **C05ZAF** was excluded: its function is to "check [the] user's routine for calculating 1st derivatives" ([25]). Since IRENA generates the equivalent of such a routine automatically, using REDUCE's symbolic differentiation, there is no need for this routine nor any opportunity for users to apply it.

The only function of the four routines **E04DJF**, **E04DKF**, **E04UDF** and **E04UEF** is to supply optional parameters to other routines (**E04DGF** and **E04UCF**). In IRENA-1, this functionality was omitted, on account of time constraints, although it was later provided by incorporating these four routines into jackets written for **E04DGF** and **E04UCF** (see section 12.1), so that all of the NAG optional parameters were made explicit. Where appropriate, they could then be given defaults equivalent to the internal NAG defaults by the usual IRENA mechanism. Naturally, these four routines do not appear separately in IRENA.

The Foundation Library also includes 83 "Fundamental Support Routines" which are "documented in compact form". In general, separate IRENA interfaces were not provided for these, the only exceptions being the two routines in the **X01** (Mathematical Constants) chapter. Interfaces for these already existed, before the decision to restrict the initial version of IRENA to the Foundation Library was taken and, since REDUCE does not explicitly provide Euler's constant, γ, which is calculated by **X01ABF**, this routine was included. Since a decision had been taken that, in general, routines from the Foundation Library would be included in IRENA on a chapter by chapter basis, the only other routine in this chapter, **X01AAF** which calculates a value for π, was also included (although clearly redundant at the user level).

Additionally, 167 other top level routines from the full NAG Library are included, as auxiliaries, in the Foundation Library. These are listed in the Foundation Library Handbook but are otherwise undocumented there. One of these routines, **A00AAF**, provides precise details of the version of the Library in use, in particular for use in reporting errors, and was included in IRENA for the same reason. One other routine from this set, **C02AJF**, was used in the pre-release version of the Foundation Library, in the example program of another routine, and so was included in the IRENA target set, to allow the equivalent usage. This usage was eliminated

from the Foundation Library before its final release and so the **C02AJF** interface was also removed from IRENA-1.

Finally, the routine **D03FAF**, which solves the three-dimensional Helmholz differential equation, is the only NAG routine which uses a three-dimensional Fortran array. As none of the GENTRAN code in IRENA was designed to handle this case and, in fact, an assumption that at most two-dimensional structures would occur was fairly deeply embedded in Dewar's code, this routine (and so the entire **D03** chapter, which contains two other routines) was excluded from the target set. As any future version of IRENA is likely to have a completely reworked jazzing mechanism, it was not considered worthwhile to attempt to extend the dimensionality of the present version, to accommodate this one routine.

The effect of the decision to concentrate on the Foundation Library, on work already performed, was that 122 fully processed routines and another 67 for which only defaults files had been produced were not included in the original release. However, the jazz files for these routines have largely been maintained to reflect general changes and most could be made available with little extra work.

8.2 Difficulties encountered in completing IRENA-1

The previous section mentioned how the work involved in processing any NAG routine for inclusion in IRENA tended to increase with the number of routines already processed. Several other factors also slowed down the development of the system.

- Frequently, when an additional NAG routine was processed, it was found that its ASPs did not fit into the existing structure, so that their functionality had to be analysed, a natural representation chosen and new code written to convert this to a GENTRAN template.
- A similar situation frequently occurred with other parameters, in that existing jazzing commands failed to map between the natural representation and the required NAG parameters, so that new jazz functions had to be written. (See, for example, **cmplxquots** and **outputconj** in section 11.1.1.)
- The existing system did not always provide the necessary control of the interface and, as it evolved, became complex and awkward to use. An example of this lack of control, discussed further in section 9.2.5, was that no general means was available to determine the

order in which IRENA would issue prompts, so that designing a logical basis for interaction between the system and the user could be very difficult[2].

- Changes to underlying software outside of the project's control, such as compilers and loaders, at times disrupted the functioning of the system. In particular, a change to the internal representation of real numbers, introduced in REDUCE 3.5, invalidated much of the original system code, which was written using the earlier representation: the ramifications of this extended for a considerable period.

From the first three of these points, it can be seen that the process of creating IRENA interfaces involved a great deal of coding of special cases. Due to the overall strategy adopted for IRENA, which was written as an extension of REDUCE, this code was written in the REDUCE system language RLISP and so is not portable outside of the REDUCE environment.

Functions available in the version of Lisp used to implement RLISP are transparently available in RLISP, even if they do not form part of the RLISP definition. In our case, the RLISP was implemented in PSL (Portable Standard Lisp) and the difficulty of possibly porting IRENA code is further increased, even in RLISP environments, because PSL functions are used in the IRENA code. Although PSL is the traditional basis of RLISP, there is a growing number of implementations based on varieties of Common Lisp, as well as some early implementations in Cambridge Lisp. Porting IRENA in its present form to such REDUCE platforms would involve detecting and recoding the PSL function usage. Fortunately, a different strategy, discussed in chapter 15 and in particular in section 15.3, offers a much more portable alternative for much of the system.

8.3 The effect of NAG routine complexity on IRENA development

In an attempt to quantify the effect of the complexity of the underlying NAG routines on the development of the corresponding IRENA interfaces, GLIM [11] was used to examine the relationship between the size of the IRENA jazz and defaults files and the number of parameters of the NAG routine, for the routines included in IRENA-1.

[2]This illustrates a "lack of scalability", analogous to that found in the investigation discussed in section 8.3, in that, for the simpler routines initially investigated, the order of prompting did not constitute a problem. A more general mechanism to control the prompting order would be difficult to graft onto the existing system, since IRENA incrementally determines which parameters are required at any point, in terms of what is already known, making prediction of the dependencies difficult. This feature is quite deeply embedded in the system design.

Initially, only the total parameter count of the NAG routines was used.

In figure 8-1 the sums of the lengths of pairs of jazz and defaults files[3] are plotted against the number of NAG parameters. By eye, the trend appears quite linear.

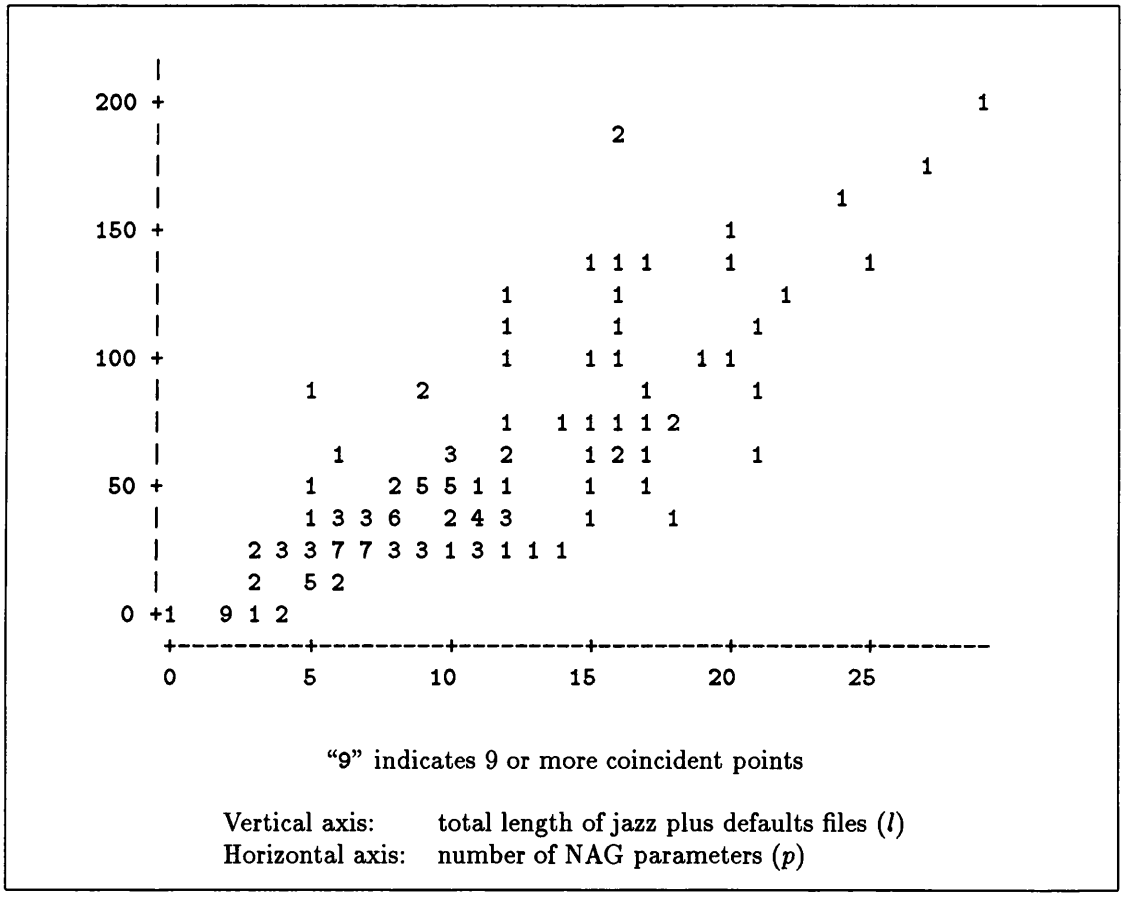

Figure 8-1: Plot of IRENA file sizes versus NAG parameter numbers

Using GLIM to fit a straight line through the origin gave the relation $l = 5.292p$, with a standard error of 0.1723 in the coefficient of p. Adding a quadratic term gave a small (6%) but significant improvement in fit and the relation $l = 3.924p + 0.08515p^2$, with standard errors of 0.4690 and 0.02727, respectively, in the coefficients. This model accounted for 70% of the deviance about the mean in the observations. (The effects of higher-order terms were not significant, here.)

[3] The results for the sizes of the jazz and defaults files separately are very similar and are not presented here.

For the routine with the greatest number of parameters (29) the estimate of the quadratic effect is about 63% of that of the linear term; the average is about 28%.

The wide scatter of the points in figure 8-1, representing the residual 30% of the deviance, suggests that one or more other factors may have influenced the size of the jazz and defaults files; in other words, the complexity of the task of processing a NAG routine for IRENA requires a more complex measure than simply the number of parameters. For instance, another possible consideration was the number and complexity of the NAG ASPs.

To provide data for a more detailed investigation, a C program was written which analysed the specfiles of the routines and produced counts of the numbers of parameters in the categories "input", "output", "input/output", "workspace" and "dummy". For the further categories "function" and "subroutine", the total number of NAG routine parameters in the category and the total number of parameters of those parameters were counted[4]. These are shown in appendix E as "main" and "2nd level" respectively.

Using these extended data, a further GLIM analysis was carried out; a transcript of this appears in appendix F. In this, output parameter counts for function routines and parameter counts for function ASPs were augmented by one, to give the value returned by the function the same consideration as other output values, returned in parameters. During the run, as the significance of the output parameter count was only moderately high, the input, output and input/output totals were combined, with the input/output figure being doubled, since both input and output rôles contribute to the effort of jazzing these parameters.

Not surprisingly, the effects of the number of dummy parameters was insignificant – these require no manual intervention in their processing (and, in any case, only two occur in the Foundation Library). The workspace parameter count also proved to be insignificant – the only manual operation involved in connection with these was the transcription of the values of associated workspace length parameters into the defaults file; however, these parameters are classified as input.

Perhaps more surprisingly, the effect of the ASPs was not great – there is a linear effect from the total ASP size (calculated as the total of the number of ASPs and of their parameters, augmented by one for each function ASP) but this is dominated by the effect of the ordinary

[4] The program also, incidentally, identified the only occurrence in the routines used in IRENA-1 of an external (that is, subroutine or function) parameter with no user-supplied parameters of its own. D01BBF takes as its first parameter the name of a NAG supplied subroutine, determining which of four possible quadrature rules should be applied. This illustrates how unique parameterisations occur in individual NAG routines, frustrating attempts at automatic processing.

input and output parameters, which, in contrast to the previous analysis, has both quadratic and cubic components. The final relationship was $l = 1.758s + 0.5104t^2 - 0.01177t^3$ where l is again the combined lengths of the jazz and defaults files, s is the above measure of the total ASP size and t is the combined input/output ("transput") parameter count[5]. The standard errors in the three coefficients were, respectively, 0.3188, 0.02446 and 0.001007 and the model accounted for 82% of the total deviance about the mean. A clearer impression of the relative sizes of the various effects may be gained by rewriting the relationship as $l = 1.758s + (0.7144t)^2 - (0.2275t)^3$.

The t dependency has a maximum at about $t = 29$ which is close to the largest value (32) of t which occurs. In fact, the data in this region are rather sparse, with only five values above 23, so little reliance should be placed on the exact form of the fitted expression at high t values. The behaviour for small t values is of more interest – and is much more reliable, since the data here are much denser, with more than half of the observations having t values of 9 or less.

For a more striking demonstration of this non-linearity, without the assumption of any particular model, the values of **sum** (l) and **iot** (t) were sorted into increasing **iot** order and the ratio of total values of **sum** and **iot** was calculated for each quartile. The respective values were 2.206, 3.563, 4.433 and 5.569; a linear dependence of l on t would be expected to result in approximately equal values of this ratio in the four quartiles.

A factor which would be expected to influence the difficulty of processing a routine and which may account for some of the nonlinearity observed is the interconnectedness of the NAG parameters, in the sense that the meaningful jazzing of one parameter may involve an understanding of the rôles of others. It is, however, difficult to see how this might be quantified.

[5] Although there is no linear component in t, the interaction of the quadratic and cubic components, which have opposite curvatures, produces a long, almost linear, central section in the graph of l against t.

Chapter 9

Modifications to IRENA-0

A general description of IRENA-0 may be found in [4] or [6] or (with some IRENA-1 features) in [32] – a more complete account exists in [5]. Listed below are a number of general features which were added to this by the present author, to improve the general user interface.

9.1 Switches

Switches are a feature of REDUCE allowing a general level of user control over various aspects of the system, most commonly over the mode of evaluation or display of categories of expressions. Switches are essentially bivalued, are global in scope and are commonly (but not exclusively) represented by global system variables; switch settings may occur in the same context as any other REDUCE instructions, including in the user's REDUCE initialisation file, which is read on REDUCE startup. They can thus be used to tailor REDUCE's default behaviour to the user's needs.

Various switches added to IRENA to improve the user interface are described here.

Verbose

It was felt that some users would not wish to receive the standard instructions on how to supply parameter values, from each incomplete IRENA-function call, nor on how to access the output list, in every case. A switch **verbose** was therefore introduced to allow these to be turned off. It does not, however, inhibit other, less common, messages nor those generated by **monit**, described next. By default, **verbose** is **on**.

Monit

A significant time lapse occurs between the call of an IRENA-function and the return of the output list (see figure 9-1). On the original system at NAG, running on a Sun-3, this was typically half a minute. To reassure users during this period, messages indicating the stages of the IRENA process were added, controlled by the switch **monit**, which is **on**, by default.

Irena-timing

This switch, meant mainly as a development tool, extends the function of **monit** to include processor and elapsed timing information. The processor times are obtained from the PSL **time** function, which according to its documentation [36], returns "CPU time in milliseconds since login time" but, in fact, appears to include only time spent in PSL processes, since REDUCE was loaded, not in external processes (such as compilation) nor in processes loaded with oload (such as the Fortran execution). No attempt was made to circumvent this, since plans for the next release of IRENA included running the Fortran as a separate process, possibly on a remote machine. An additional reason was that, for an interactive system, obtaining an objective measurement of elapsed times was considered more significant than investigating Fortran processor time, especially since enhancement in the performance of the Fortran was not within the remit of the project.

Typical **irena-timing** output is shown in figure 9-1.

It can be seen that the elapsed time is dominated by the time to load from the NAG Library. In this example, the full Mark 15 Library was used. In the released version of IRENA, this was replaced by the smaller Foundation Library, giving a significant reduction in the elapsed

```
                                IRENA timings

                              processor     elapsed
Beginning code generation ...
   time taken ..............   204 ms        1 s
Beginning compilation ...
   time taken ..............   17 ms         6 s
Beginning loading ...
   time taken ..............   51 ms         26 s
Beginning execution ...
   time taken ..............   < 1 ms        < 1 s

Fortran execution time is not included in the above processor times.
```

Figure 9-1: IRENA Timing Output

time, from 26 to 11 seconds, reflecting the relative sizes of the two libraries (1038 and 423 routines, respectively). Thus, for interactive use, there is a clear tradeoff between the diminished functionality and the shorter response times which result from using a smaller numeric library.

Promptall

This switch is discussed in section 9.2.6.

Mnemprompts

The IRENA *mnemonic functions* provide an alternative means of calling NAG routines which return a single value, mimicking the syntax of normal REDUCE procedures. They also, at times, provide a single interface to several NAG routines. In this latter case, control parameters may occur for some routines but not others: these have system default values which might conceivably be cancelled by the user. To avoid the possibility of confusion in such a case, since the user is generally unaware of the underlying routine being used, the switch `mnemprompts`, temporarily reset in the code of the mnemonic function, allows `promptall` to operate for the routine in question, regardless of its actual setting. (The same effect could probably be obtained by a local resetting of `promptall`.)

Fortinclude

In IRENA-0, solving a series of similar problems involved running through IRENA's complete generate-compile-execute-load cycle for each problem, which soon increases the elapsed time to unacceptable levels for interactive use. The design of IRENA makes it difficult to avoid repeating this cycle[1]; however, a partial solution is provided by the **fortinclude** switch, which allows the user to specify two lists of filenames: it is assumed that the files contain Fortran fragments, to be inserted after the automatically generated non-executable Fortran statements (type declarations etc.) and before the automatically generated executable statements. The contents of the files in the first list are inserted in the main program and in all subprograms, those from the second list in the main program only. **99999 CONTINUE** is inserted as the last executable statement in the main program.

This facility is meant mainly for use in generating free-standing programs (with the switch **codeonly on**). It allows, among other things, the construction of loops in the main program, to handle different data values (for which a **READ** may be provided) and the communication of data values into ASPs via **COMMON** blocks. An example of its use is given in Keady and Richardson [17].

9.2 Prompting

A number of enhancements were made to the IRENA-0 prompting mechanism, some of which are described below.

9.2.1 Name substitution

The most radical prompting enhancement was to allow the use of **@**, in place of the name of the object being requested – for instance, in IRENA matrix references such as the use of **vec** below. (An outline of IRENA matrix types was given in section 6.1 – for a full description, see Appendix A of the IRENA User Guide [33]). This feature allowed much more meaningful prompts to be introduced without requiring the user either to remember a shorter alias or to

[1] In the later Axiom "NAGlink", drawing on experience with IRENA, part of this cycle is eliminated by not recompiling those code components which are unchanged, following in part the suggestions made in section 15.3.3, below. This will be discussed in section 17.1.

re-type the prompt-alias. For instance, in **e04naf**, in response to the prompt

```
(Real vector) linear term coefficients?
```

it is possible to reply

```
vec @ {1,1,3,2,5};
```

(where **vec** introduces the IRENA vector type, mentioned in section 6.1.)

In the case of matrices, the user may wish to provide a reply of the above form or simply refer to a REDUCE algebraic object, **ltc**, say, by typing

```
@=ltc;
```

The **@** mechanism thus provides a simple means of avoiding problems which could have arisen in simply allowing the omission of names whose position is variable.

For objects other than matrices, the user's response would always have begun with the name of the object followed by an equals sign. In these cases, IRENA-1 now appends **@=** to its prompt.

9.2.2 Boolean variables

In many NAG routines numeric or string variables are used as switches. For a more natural interactive interface, these were identified by the jazz command **boolean**, which indicates that the prompt type should be set to **Y or N**. This feature was introduced much earlier than the more general **set-type**, described below, and its use was later replaced by instances of that command. An associated **prompt-alias** command is normally used to supply a prompt in the form of a question requiring a "Yes" or "No" answer.

(For keyline use, keywords equivalent to these answers are required. Providing both these and the boolean prompt, using the standard IRENA jazzing and default facilities, is quite complicated and an automatic mechanism for generating the necessary commands was developed. An example appears in section 15.2. Normally, when keywords are defined, IRENA prompts with a choice of these. To inhibit this in the "boolean" case, a variant jazz command **qkeyword** was added.)

In some NAG routines, one exceptional parameter value indicates a non-standard problem, other possible values represent normal data – this case is handled by introducing an IRENA `scalar` – essentially an additional, quasi-NAG parameter, and treating this as described above.

9.2.3 Prompt types

In IRENA-0 prompting, the name of the required object is prefixed with a parenthesised type description which is determined automatically, initially from information in the infofile, where the Fortran type of each NAG parameter, whether it is an array and, in particular, whether it is a vector are specified. To extend this feature to new data types defined by jazz-functions, the procedures **`set-type`** and **`get-type`** were added, the former to be used in defining the jazz-function's associated check-function (see section 7.3.1), signalling the type of any as yet unavailable objects, the latter for use in setting the prompt type.

Occasionally, the automatic determination of prompt type in IRENA-0 had infelicitous results: for example, for some of the built-in jazz commands the type of a jazzed object differs from that of its NAG counterpart but this information was not available to the prompting mechanism. To overcome such problems, the jazz commands **`{set-type}`** and **`{set-type@}`** were added, to override the automatic type. The second of these is used to indicate cases where `@=` should be added to the end of the prompt. Considerable use was later made of these facilities, to tailor prompts to match the type of response required from the user without having to include this information in a parameter's prompt-alias.

To allow maximum flexibility, when the automatic type is overridden the type is not printed in parentheses. If the parentheses are required, they may be included as part of the supplied type (which has the form of a REDUCE string). An example of printing such a type description is given in section 9.2.4.

9.2.4 Keywords

When prompting for a variable for whose values keywords had been defined, IRENA-0 simply displayed the string **`(One of the following)`**, followed by a list of possible keywords, which could run on to continuation lines. Here again there was a tension between the need for long, mnemonic keywords and short, easily typed replies. To overcome this without interfering with

the underlying mechanism, in most cases pairs of keywords were introduced with the same meaning. The keyword prompt was modified so that two keywords appeared on each line, to help associate these pairs, and any string defined by a **set-type** command was first displayed on a separate line. For instance, the integer-valued parameter **ISELCT** in **F04MCF** controls the form of the left-hand side of the system of equations to be solved: with **promptall on**, the prompt generated for this parameter is

```
Form of left-hand side
(one of the following)  l*d*l-transpose*x, l1,
                        l*d*x, l2,
                        d*l-transpose*x, l3,
                        l*l-transpose*x, l4,
                        l*x, l5,
                        l-transpose*x, l6?
```

(The visual template interface style used in NAGlink – see section 17.1 – allows a similar facility to be implemented rather more neatly there, with the forms of equation allowed being displayed adjacent to a set of "radio buttons", only one of which can be activated at any time.)

9.2.5 Phased-prompt

One major difficulty in controlling the interaction between IRENA and the user was that the order in which IRENA issued its prompts was determined by the previous overall history of its parameter processing in the current call. This could result in unfortunate prompting sequences – for example, in **d02raf** the prompt for the NAG parameter **X** (**Initial mesh?**) could occur before that for **INIT** (**Do you wish to supply an initial mesh?**)

In some simple situations, it was possible to control the prompting order via the dependencies among the various parameters defaults; however, defaults were specified in terms of the original NAG parameters, so that, where the routine had been reparameterised for IRENA, this method was not available. Thus, without a major redesign of the system, it was not always possible to arrange for prompts to occur in a logical order.

This was particularly troublesome in cases where a single NAG routine could operate in different modes, with a single NAG parameter serving both to input a data value for a "normal" mode of operation and as a flag to signal an "abnormal" mode, by taking an "impossible" value (such as `-1` for a scaling parameter). In this case, the parameters in question were originally replaced in IRENA with distinct "flag" and "data" parameters, which, unfortunately, were often prompted for in the wrong order. To overcome this, a new jazz command **phased-prompt** was introduced, to replace the direct reparameterisation with a two stage prompting mechanism: first, a prompt is issued, expecting a reply of either `Y` or `N`, one of which corresponds to the special, flag value, with the other triggering a second prompt for the parameter itself. In some cases, further action is required to set up the "special" value.

For example, in the jazz file for `d02cjf`, we have[2]

```
{phased!-prompt} G : determine! where! an! end!-point! criterion! is! zero
                     (n > unset) end!-point! criterion
```

which means that, if the user replies `N` to the prompt:

```
(Y or N) determine where an end-point criterion is zero?
```

the function `G` is given the value **unset**; if the user replies `Y`, however, the further prompt

```
(Function)  end-point criterion?
```

is issued, to obtain an actual function definition. The normal use of the special value **unset** is to inhibit the generation of Fortran assignments for the parameter in question; however, in this particular case, `G` must be the dummy NAG routine `D02CJW` – this is organised by trapping the **unset** value in the GENTRAN template for `G`'s ASP[3].

[2] The exclamation mark (`!`) is REDUCE's escape character or "letteriser", which is used ensure that the following character represents itself and not, for example, an arithmetic operator or (in the case of a space) the end of a token.

[3] This technique is used in various cases where a specific NAG dummy routine is required. It slightly extends the meaning of **unset** to include "set to a dummy". This does not interact with the normal use of **unset**, which is only concerned with inhibiting the generation of Fortran assignments, and the only cost is the need to include the appropriate code when developing the ASP's template – whose use, however, tends to be restricted to the small set of routines requiring this facility.

9.2.6 Promptall

IRENA does not normally prompt for parameters for which default values are provided. Whilst this is completely appropriate for housekeeping parameters, in other cases users may sometimes wish to reset defaulted parameters and, therefore, need to be reminded of their names. Other aspects of this topic are discussed in section 15.1.

As there was, at this point, no internal IRENA help, the **promptall** switch was introduced: its function is to cause IRENA to include prompts for the values of defaulted parameters not currently identified as **housekeeping** in the system defaults file. (This identification sometimes includes parameters for which IRENA would already prompt, if they were required – these may be thought of as conditionally housekeeping. Flagging them as housekeeping does not, of course, inhibit IRENA from prompting for them when they are required.) The user has the option of overriding the system declaration of which parameters are **housekeeping**, as part of the user defaults system. An example of a **housekeeping** entry in a defaults file will be presented in section 10.1.1.

When the user wishes to retain the default value of a parameter, during an IRENA run, this can be accomplished by providing a null response to the prompt; if no default exists, IRENA will say so and prompt again. An obvious enhancement for a future release would be to display default values if the user so requested.

A program to automatically generate lists of the most obvious housekeeping parameters, those derivable from the dimensions of input arrays, was written by a NAG sandwich student, N. Atta, to the present author's specification and under his guidance. Additionally, this program generated default values for these parameters and also extracted suggested values for parameters defining workspace dimensions from the NAG documentation. This program used technology similar to that developed by Dewar for generating the GENTRAN templates – see [5].

At present, not all defaults files have been processed to include a **housekeeping** entry and those which have do not all include conditionally housekeeping parameters. Full implementation of this scheme has now been deferred until a later release of IRENA, when it may be revised to allow the distinction of housekeeping, control and data parameters, as discussed in section 15.1.

To take advantage of **promptall**, the IRENA jazz files include prompts for defaulted parameters.

9.3 Output enhancements

9.3.1 Output indexing

To meet the goal of "informative naming", described in chapter 2, the names of objects output by IRENA-functions, as well as those of input parameters, should be as meaningful as possible. However, few users are likely to be happy typing long names to refer to these objects so a new function @ was introduced to provide indexed access to the output list of IRENA-functions. This allows the nth item of the output list to be referred to as @n. In case the length of the output list should make this mechanism inconvenient, @0 provides an index to it. In normal usage, IRENA briefly indicates how to use this mechanism before printing the output list, although, as indicated in section 9.1, this may be turned off using the **verbose** switch.

These displays may be demonstrated by the following call, based on the standard NAG Library example, of **d02raf**, which solves a two-point boundary problem for a system of ordinary differential equations:

```
d02raf(range=[0:10],
       slope1(x,y1,y2,y3,eps)= y2,
       slope2(x,y1,y2,y3,eps)= y3,
       slope3(x,y1,y2,y3,eps)= - y1*y3 - 2*(1 - y2*y2)*eps,
       left_hand_boundary_condition1(y1,y2,y3,eps)= y1,
       left_hand_boundary_condition2(y1,y2,y3,eps)= y2,
       right_hand_boundary_condition1(y1,y2,y3,eps)= y2 - 1)$
```

This generates the display:

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

{final_mesh,solution_on_final_mesh,absolute_error_estimates,

 overestimate_of_final_continuation_parameter_increment}
```

and typing @0 produces the further display:

```
1: Final_mesh
2: Solution_on_final_mesh
3: Absolute_error_estimates
4: Overestimate_of_final_continuation_parameter_increment
```

Additionally, it is possible to produce a complete, titled print-out of all items in the output list, using the form @@. As this would generally be too extensive for interactive use, it is not mentioned in the output display but is documented in [33].

9.3.2 Hidden output

It is sometimes the case that the NAG routine returns parameters which are redundant for the IRENA user – for instance, the number of output values of a particular type returned in an array which, in IRENA, is trimmed to contain only these values. However, such parameters may still be required by IRENA for its own processing – for example, to define the elements of this trimmed array.

In order to prune such parameters from the output list, the convention was adopted that no output parameter whose name began with ***noname*** would appear in the output list[4]. The standard IRENA jazz command **output** could then be used to rename unnecessary parameters. This facility was often used with the conditional renaming mentioned in section 9.3.3 to conditionally inhibit output parameters on occasions when they held no useful information: for instance, error related output when no error had occurred.

9.3.3 Enhanced conditional output

The specification of IRENA-0 allowed for conditional renaming of NAG output parameters, depending on the value of some NAG parameter. As originally implemented, this took the form:
{output} *nagname* : case (*valuelist*) *namelist*
and had the effect that the IRENA name used for the output object was taken as the element of *namelist* corresponding in position to the input value of the parameter *nagname* in *valuelist*;

[4]This is consistent with the normal REDUCE convention that names which include non-alphanumeric characters may be freely used as system variables and that users should avoid such names.

optionally, an additional, final entry in *namelist* provided a default name. This was adequate to handle the case where a NAG routine would calculate different quantities, depending on the setting of some input parameter which served as a switch. However, as more routines were processed, examples were found in which the content of a NAG output parameter would be indicated by the value of some other output parameter (often **IFAIL**, the error indicator). To accommodate this, the syntax of this construct was modified, allowing **out(***valuelist***)** as an alternative to *valuelist*, signalling that output values should be used in the test.

A less common situation, where the same information might conditionally be located in different NAG structures, led to the introduction of the **cond-out** jazzing command, described in section 11.3 (and briefly in appendix D).

9.4 The IRENA help system

As already mentioned, IRENA tries to make the use of its functions as transparent as possible, by using English language prompts and descriptive output parameter names. It is also part of IRENA's basic design that, when a failure occurs in running a NAG routine, instead of displaying the NAG **IFAIL** error code, it displays text based on the Fortran Library Manual's description of the meaning of the particular **IFAIL** value.

It was not, in general, practicable to recast these messages in terms of IRENA parameters, so, in order make them intelligible, a help facility was introduced in IRENA, to provide details of the relations between the NAG parameters and IRENA's. This makes use of a free-standing C program, specified by the present author but partly written by N. Atta and completed by M. Mc Gettrick (a teaching company associate at NAG). The interface to IRENA, which makes use of the REDUCE **system** command to run the program, was provided by the present author.

The help system also allows users to check the default settings of NAG parameters.

The system consists of four functions, callable from within IRENA:

- **jazzing**
- **default**
- **details** and
- **explain**

Jazzing

The syntax of this function is `jazzing(`*NAG parameter name*`,'`*IRENA-function name*`)`[5] or `jazzing(`*NAG parameter name*`,@)`. The first form generates a listing of all jazz entries for *IRENA-function name* which involve *NAG parameter name*; the second form does this for the most recently called IRENA-function. Thus, a call to `d01apf` might give the error message

```
** On entry, B.le.A or ALFA.le.-1 or BETA.le.-1 or KEY.lt.1 or KEY.gt.4:
   A     =   2.00000D+00   B    =   1.00000D+00   ALFA =   0.00000D+00
   BETA =   0.00000D+00   KEY =                 1
** ABNORMAL EXIT from NAG Library routine D01APF: IFAIL =      4
** NAG soft failure - control returned

   On entry, B <= A,
   or        ALFA <= -1,
   or        BETA <= -1,
   or        KEY < 1,
   or        KEY > 4.
```

from which it is obvious that the problem lies with `A` and `B`. What this means in terms of IRENA parameters can now be determined:

```
71: jazzing(A,@);

{rectangle} A,B : range
```

in other words, the problem lay in the use of the rectangle **range**, which must have been defined as `[2:1]` instead of `[1:2]`.

[5] It is necessary to quote the function name with an apostrophe (') to prevent IRENA from attempting to obey the function.

Default

The syntax of this function is **default**(*NAG parameter name*,'*IRENA-function name*) or **default**(*NAG parameter name*,@). The first form prints out the default setting of *NAG parameter name* for the function *IRENA-function name*; the second form does this for the most recently called IRENA-function.

The error message displayed in the previous section might cause the user to wonder what setting of **ALFA** had been used. If this (or its IRENA equivalent, **alpha**) was not set in the IRENA call, it must have taken the default value

```
72: default(alfa,@);

      ALFA : 0
```

(so this was not the cause of the problem).

Details

The syntax of this function is **details**(*NAG parameter name*,'*IRENA-function name*) or **details**(*NAG parameter name*,@). It produces output equivalent to the **jazzing** and **default** functions together, thus:

```
73: details(beta,@);

d01apf has no jazzing for beta.

Default Information:

      BETA : 0
```

Explain

Having received the output of the **jazzing** function, the user might wish for clarification of a jazz command occurring there. This may be obtained with **explain**.

The syntax of this function is **explain**(*jazz command*). It produces a brief description of the particular jazz command. For instance, continuing the previous example (and using the REDUCE facility to omit parentheses around a single argument):

```
74: explain rectangle;

RECTANGLE : (Input jazz) defines two NAG scalars (or array elements) or
                         two linear arrays as equivalent to an IRENA
                         rectangle
```

The **explain** function references a partial database of descriptions of jazz commands, produced by the author. Certain of the less common and more complicated jazz commands are not yet documented – the priority for completing this work will depend on user response.

9.5 Mnemonically named functions

IRENA functions normally return a list of the names of objects which have been created in the REDUCE environment. In cases where this list contains only a single object, additional "mnemonically named" functions, which return that object, were defined. These are described more fully in chapter 13.

9.6 Handling Hermitian sequences

The C06 chapter of NAG routines is largely concerned with the calculation of discrete Fourier transforms of both real and complex sequences. Several of these routines are designed to process Hermitian sequences, packed as real sequences of the same length. (A Hermitian sequence consists of a real number followed by a sequence of complex numbers, this complex subsequence being conjugate to itself reversed.) The packed sequences can be stored as real IRENA vectors, REDUCE single column matrices etc. Functions were provided in IRENA for the convenient

handling of Hermitian sequences: **display-hermitian** takes a packed Hermitian sequence and displays it in full:

```
1: vec h {1,2,3,4,5,6,7};
2: display!-hermitian h;

1

2 + 7*I

3 + 6*I

4 + 5*I

4 - 5*I

3 - 6*I

2 - 7*I
```

Functions **hermitian2packed** and **packed2hermitian** were also provided, for converting between the two representations. The latter provides the same functionality as the NAG routine C06GSF, without the overhead of the IRENA Fortran cycle.

9.7 Zero-filling arrays

In addition to providing code to improve IRENA's user interface, the present author also, at times, corrected faults and infelicities in the design and implementation of the IRENA-0 system code. Perhaps the most fundamental of these was the following.

A number of the GENTRAN templates for IRENA-0 ASPs on occasion produced incorrect results, which were found to be due to the presence of Fortran arrays with unassigned elements. In most ASPs, all elements were assigned automatically but, in these particular cases, a dense representation was being used for a sparse matrix, in order that an available NAG routine could be used to perform a transformation required by the ASP.

In other instances, where sparse matrices were being used in general routines, large amounts of code were being generated which consisted entirely of zero assignments. Even in the reasonably sized examples provided with the NAG library, the length of the Fortran source was at times sufficient to break early versions of the NAG Fortran 90 compiler; "real-life" problems would almost certainly have caused similar difficulties for many compilers, as well as adding significantly to the compilation time for the generated code.

To overcome these problems, all local and input arrays generated by IRENA were initially zero-filled, the former by generating appropriate `DATA` statements, the latter by means of `DO` loops. Code was then added to the array-translation facilities in IRENA to filter numerically zero values from the array element assignments.

Chapter 10

Default system usage in IRENA-1

In this chapter, annotated examples of some moderately sized system defaults files, chosen to illustrate various features of the defaults system, are presented and described. Although REDUCE (and IRENA) are not, normally, case sensitive, a convention generally adopted to simplify the understanding of IRENA files is that NAG names appear in upper case and IRENA names in lower case. That convention is followed in these examples.

The jazz files corresponding to defaults files examined in this chapter are displayed in chapter 11. Material from these chapters will be reexamined in section 15.2, which considers how the activities of defaults specification and jazzing are interrelated.

10.1 Descriptions of selected defaults files

10.1.1 F02BJF

`F02BJF` calculates the eigenvalues and, optionally, the eigenvectors of the generalised eigenproblem $\mathbf{Ax} = \lambda\mathbf{Bx}$ where $\mathbf{A}$ and $\mathbf{B}$ are real, square matrices.

```
% Defaults for F02BJF

     housekeeping : N, IA, IB, IV

     N : min(dim(A,2),dim(B,2),IA,IB)  % The order of the
                                       % matrices  A and B.

     IA : dim(A,1)                     % The first dimensions
                                       % of the Fortran arrays
     IB : dim(B,1)                     % A and B.

     EPS1 : fpeps                      % A tolerance for treating
                                       % small elements of the
                                       % transformed matrices as zero.

     matv!-key : 1                     % An IRENA scalar, used to
                                       % communicate with the jazz
                                       % system.

     MATV : if matv!-key = 1           % A NAG scalar used to
               then TRUE               % indicate whether the
            else if matv!-key = 2      % eigenvectors should be
               then FALSE              % found.

     IV : N                            % The first dimension
                                       % of an output array.

end;
```

Figure 10-1: Annotated defaults file for F02BJF

Taking each entry in its defaults file in turn:

housekeeping : N, IA, IB, IV

This IRENA-1 feature, previously mentioned in section 9.2.6, was added by the present author. Originally, it was intended only to inhibit prompting for housekeeping parameters, when the **promptall** switch[1] was on. However, for more complex routines, where there are varieties of ways of specifying some parameters, it may be used to inhibit *unnecessary* prompting for any items in the following list. If a value is required for a member of the list (usually because it is needed to calculate the value for a NAG parameter, which the user has not provided) then the normal prompting mechanism takes effect.

[1] Which permits the issuing of prompts for defaulted variables, providing the user with an additional mechanism to override defaults: see section 9.2.6.

Here, **housekeeping** introduces a list which consists of the NAG housekeeping parameters, **N**, **IA**, **IB** and **IV** (which can be automatically flagged as housekeeping, since they are derivable from array dimensions – see section 9.2.6). The user should never need to override the default settings for the above NAG parameters.

Note that, even with **promptall on**, IRENA does not prompt for objects defined by the jazz commands **scalar** and **vector** (unless these are required to calculate the default values of NAG parameters). For example, although we never want to prompt for the **scalar matv-key** (whose function is described below), it is not necessary to include it in the **housekeeping** list.

```
N : min(dim(A,2),dim(B,2),IA,IB)
```

Such entries may be generated largely automatically, as described in section 9.2.6. **A** and **B** represent square matrices but, since NAG allows oversized arrays, a decision was taken at the start of the project to take the minimum of the dimensions in cases such as this. In fact, as IRENA users may be expected to supply the matrices rather than the equivalent of the NAG arrays and, especially, as the NAG Fortran Library manual no longer emphasises this feature, taking this minimum is largely redundant. However, since taking such minima adds very little overhead to the process of generating values for the NAG parameters and may still, occasionally, be appropriate, no action has been taken to remove them.

```
IA : dim(A,1)
IB : dim(B,1)
```

The first dimensions of the arrays **A** and **B** are required as NAG input parameters **IA** and **IB**.

```
EPS1 :  fpeps
```

EPS1 is a tolerance for regarding small elements of the transformed matrices as zero. Here, it is given the value **fpeps** which translates in the generated Fortran into a call to the NAG routine **X02AJF**[2]. This returns the smallest number representable in Fortran which, when added to 1, yields a result greater than 1.

[2]Fpeps is one of several "fp" constants available in the IRENA defaults system to obtain "machine characteristics" via NAG X02 routines; these are included in table 7.2. Of the fp constants, only fpeps and fphuge – the largest floating point number – are used in IRENA-1 system defaults files.

`matv!-key : 1`

The scalar `matv-key` is introduced (in the jazz file) to allow keyword settings for **MATV**, corresponding to the values `.TRUE.` and `.FALSE.`, whilst keeping a direct "`(Y or N)`" style of prompt for **MATV** itself, rather than a prompt for a choice of the keywords (see section 15.2). Setting the actual default here, rather than in the **MATV** entry, keeps the operation of specifying the default separate from that of defining the meanings of the keys.

`MATV : if matv!-key = 1 then TRUE else if matv!-key = 2 then FALSE`

This translates the representation of keywords as `matv-key` values (defined in the jazz file) into settings of **MATV**. The actual default, which comes from the default setting of `matv-key`, is `.TRUE.`, meaning that eigenvectors should be found.

`IV : N`

The first dimension of the output array `V` (in which the eigenvectors are returned) is required as a NAG input parameter. It is given the value of the NAG parameter `N`.

10.1.2 E04GCF

`E04GCF` finds the unconstrained minimum of the sum of squares of `M` nonlinear functions of `N` variables. Its system defaults file (figure 10-2) involves only housekeeping parameters.

Defaults file entries illustrating additional features are:

`M : multiplicity(LSFUN2)`

`Multiplicity` returns the number of functions defined by the user in the family `LSFUN2`.

`ns : 7*N + 2*M + M*N + (N*(N+1))/2 + 1 + max(1,(N*(N-1))/2)`

This IRENA scalar is used in the jazz file to give tractable expressions in `output` and `reshape-output` commands, which reorganise the contents of the supposed workspace array `W`.

```
% Defaults for E04GCF

      housekeeping : M, N, LIW, LW

      M : multiplicity(LSFUN2)          % The number of functions
                                        % to be included in the
                                        % sum of squares.

      N : dim(X)                        % The number of variables
                                        % in the system.

      LIW : 1                           % The required length of the
                                        % integer workspace array IW.

      LW : if N = 1                     % The required length of the
              then 11 + 5*M             % real workspace array W.
           else
              8*N + 2*N*N + 2*M*N + 3*M

      ns : 7*N + 2*M + M*N +          % An IRENA scalar which
           (N*(N+1))/2 + 1 +          % identifies the position of
           max(1,(N*(N-1))/2)         % useful output information in
                                      % the real "workspace" array.

end;
```

Figure 10-2: Annotated defaults file for E04GCF

10.1.3 E02ADF and E02AEF

E02ADF computes weighted least-square polynomial approximations of a range of degrees to sets of data points and E02AEF[3] evaluates such an approximation at a single point.

Features of interest in the E02ADF file are:

`W(0) : 0`

This dummy default is used to establish, for IRENA, that W is a *one*-dimensional array, as this cannot be otherwise deduced.

[3] Since some of the entries in the defaults files of E02AEF and the succeeding examples are quite long, these files have not been annotated in the figures. Where necessary, the purpose of the parameters is explained in the body of the text.

```
% Defaults for E02ADF

      housekeeping : M, NROWS

      M : min(dim(X),dim(Y))          % The number of data points.

      NROWS : KPLUS1                  % The first dimension of
                                      % the output array A.

      W(0) : 0

      W(*) : 1                        % The weights.

      xmin : min(X)                   % IRENA scalars used to build an
                                      % output rectangle defining the
      xmax : max(X)                   % domain of applicability of the
                                      % approximation.

end;
```

Figure 10-3: Annotated defaults file for **E02ADF**

```
% Defaults for E02AEF

      housekeeping : NPLUS1, XCAP

      NPLUS1 : dim(A)

      XCAP : if ((x ~= unset) and (xmax ~= unset) and (xmin ~= unset))
                then ((x - xmin) - (xmax - x))/(xmax - xmin)

      x : if have(XCAP) or scalarp(XCAP) or scalarp(normalized_x)
             or scalarp(normalized! x) then unset

      xmin : if have(XCAP) or scalarp(XCAP) or scalarp(normalized_x)
                or scalarp(normalized! x) then unset

      xmax : if have(XCAP) or scalarp(XCAP) or scalarp(normalized_x)
                or scalarp(normalized! x) then unset

end;
```

Figure 10-4: Defaults file for **E02AEF**

```
W(*) :  1
```

This defines the default value 1 for every element of the array W; thus, the default gives unweighted fitting[4]. Note that a single * here means "every array element", however many dimensions the array in question may have.

```
xmin :  min(X)
xmax :  max(X)
```

These values are used in the defaults file of **E02AEF** to transform a set of arguments of the polynomial, supplied by the user, to the normalised form required by that routine.

The minimum and maximum values must, in fact, occur as the first and last elements of X; however, the defaults system does not provide a means of obtaining the values of particular array elements; to avoid introducing such a major new feature, **min** and **max** were used. This approach also allows consistent defaults coding to be used in **E02ADF** and in those other fitting routines which do not insist on ordered data sets.

Interesting features in the **E02AEF** file are:

```
XCAP : if ((x ~= unset) and (xmax ~= unset) and (xmin ~= unset))
          then ((x - xmin) - (xmax - x))/(xmax - xmin)
```

Here, **x** is the point at which the user wishes the interpolant to be calculated and **xmin** and **xmax** represent the ends of the domain of applicability of the approximation – these will probably have come from **E02ADF**. Like any defaults file instruction, this only takes effect when all of the quantities on the right have values. If **x**, **xmin** and **xmax** do have values and provided that none of them is the special value **unset** this instruction uses them to calculate **XCAP**, which represents **x** normalised to the domain of applicability, as required by **E02AEF**. The particular form of the calculation is that recommended by the NAG manual as guaranteeing a loss of accuracy of at most four times the machine precision.

```
x : if have(XCAP) or scalarp(XCAP) or scalarp(normalized_x)
       or scalarp(normalized! x) then unset
xmin : if have(XCAP) or scalarp(XCAP) or scalarp(normalized_x)
          or scalarp(normalized! x) then unset
```

[4]The IRENA documentation and the test program for **e02adf** draw attention to this particular default and point out that **canceldefault** may be used in the user's defaults file, to override it.

```
xmax : if have(XCAP) or scalarp(XCAP) or scalarp(normalized_x)
          or scalarp(normalized! x) then unset
```

These commands prevent IRENA from prompting for **x**, **xmin** and **xmax** when they are not required – that is, when the user has provided the NAG parameter **XCAP** directly. The **have** test checks whether a value has already been established for **XCAP**; effectively, this means in the keyline or through defaults processing. The **scalarp** tests check whether **XCAP** or any of its aliases exist as global REDUCE scalars, when **envsearch** is on. (This conditionality of **scalarp** on **envsearch** was added by the present author.)

The check for the existence of aliased versions of a parameter could obviously be carried out automatically; however, the use of **scalarp** is sufficiently infrequent that this has not yet been considered worthwhile.

10.1.4 D01BBF

D01BBF, which calculates appropriate weights and abscissae for use in the multidimensional quadrature routine **D01FBF**, is unusual, in that it has no housekeeping parameters.

As remarked in section 8.3, **D01BBF** uses a unique parameterisation of the range of integration (and, in the case of semi-infinite ranges, of the choice of quadrature formula). The defaults file is used here to convert the more regular representation used in IRENA into the form required by the NAG routine:

```
D01XXX : if lowerlimit = unset and upperlimit = unset then 'D01BAW
         else if (lowerlimit = unset or upperlimit = unset)
              and formula = 2
         then 'D01BAY
         else if lowerlimit = unset or upperlimit = unset then 'D01BAX
         else 'D01BAZ
```

This assumes the IRENA default interpretation of * as **unset**. If both endpoints of the range of integration are specified as * then a doubly infinite range is required, implying Gauss-Hermite quadrature, specified to **D01BBF** by giving the parameter the value **D01BAW**, the name of the appropriate NAG auxiliary routine. A single * specifies a semi-infinite range; in this case, the IRENA scalar **formula** is used to discriminate between the two possible quadrature formulae available, the values **1** and **2** corresponding to keywords defined in the jazz file to specify Gauss-Laguerre

```
% Defaults for D01BBF

      D01XXX : if lowerlimit = unset and upperlimit = unset then 'D01BAW
               else if (lowerlimit = unset or upperlimit = unset)
                    and formula = 2
               then 'D01BAY
               else if lowerlimit = unset or upperlimit = unset then 'D01BAX
               else 'D01BAZ

      A : if lowerlimit ~= unset then lowerlimit
          else if upperlimit = unset then parameter_a else upperlimit

      B : if lowerlimit ~= unset and upperlimit ~= unset then upperlimit
          else parameter_b

      ITYPE : if D01XXX = 'D01BAW or D01XXX = 'D01BAX then 1 else unset

      N : 64

end;
```

Figure 10-5: Defaults file for **D01BBF**

and Gauss-rational quadrature, respectively. The appropriate auxiliary is specified in each case. The remaining case corresponds to a finite range, for which only Gauss-Legendre quadrature is available, again specified by the choice of auxiliary.

```
A : if lowerlimit ~= unset then lowerlimit
    else if upperlimit = unset then parameter_a else upperlimit
B : if lowerlimit ~= unset and upperlimit ~= unset then upperlimit
    else parameter_b
```

Here, the alternative uses of the NAG parameters **A** and **B** are accommodated. Where either limit is finite (so not **unset**) the corresponding NAG parameter takes that limit as its value; otherwise, the NAG parameter must represent a parameter of the quadrature formula and takes its value from an appropriately named IRENA scalar.

The NAG parameter **ITYPE** allows a choice of two mathematically distinct weighting strategies for Gauss-Laguerre and Gauss-Hermite quadrature: "normal weights" and "adjusted weights" – the latter was chosen as the default.

N specifies the number of points to be used in the quadrature; the maximum allowed is used as the default.

10.1.5 F04MAF

F04MAF solves a sparse, symmetric, positive-definite system of linear equations, whose coefficient matrix has been pre-factorised by **F01MAF**.

```
% Defaults for F04MAF

      housekeeping : N, LICN, LIRN, ACC, NOITS

      N : min(dim(B),dim(WKEEP)/3,dim(IKEEP)/2)

      LICN : max(min(dim(AVALS),dim(ICN)),(NZ - N + inform1))

      LIRN : max(dim(IRN),inform2)

      ACC(1) : cc

      cc : !*userabserr!*

      ACC(2) : unset

      NOITS(1) : mni

      mni : 100

      NOITS(2) : unset

      IFAIL : 100*ifailc + 10*ifailb + 1

      ifailb : 0

      ifailc : 0

end;
```

Figure 10-6: Defaults file for **F04MAF**

Novel features are displayed by the following entries:

`ACC(1) : cc`

`ACC(2) : unset`

The NAG array `ACC` consists of two elements: the first is used to input a convergence tolerance, the second to output the final value of the quantity to which this tolerance applies (the 2-norm of the residual of the normalised equations). To decouple these on input, an IRENA scalar `cc` – the "convergence criterion" – is used to obtain the value of the first element whilst the second is specified as **unset**.

```
cc :  !*userabserr!*
```

The default value of the tolerance is set to ***userabserr***, the general default for absolute error tolerances, discussed in section 7.1. If not reset by the user, this takes the value 0.0001.

```
NOITS(1) :  mni
mni :  100
NOITS(2) :  unset
```

The two-element array **NOITS** is used to control and report on the number of iterations, in a similar manner to **ACC**'s use for the tolerance. In this case, the IRENA scalar **mni**, "maximum number of iterations", is given a numerical default directly.

```
IFAIL : 100*ifailc + 10*ifailb + 1
ifailb :  0
ifailc :  0
```

F04MAF is one of a small number of NAG routines (seven occur in the IRENA-1 subset) in which **IFAIL** takes a three-digit value on input, each digit having a separate significance. The last digit serves the normal input function of **IFAIL**, so that setting this to 1 gives a *soft failure*, meaning that the NAG routine does not terminate the calling program on detecting an error. The middle digit, if zero, suppresses the output of error messages by the routine and this was selected as the default, since IRENA itself prints similar messages when an error return from **F04MAF** occurs. A zero value of the first digit suppresses warning messages: the authors felt that, in general, displaying a potentially large number of warning messages was not appropriate in interactive use, so the inhibition of such messages was uniformly chosen as the default behaviour. The introduction of the IRENA scalars **ifailb** and **ifailc** allows the user to override the error and warning defaults separately (using meaningful keywords defined in the jazz file).

Chapter 11

Jazz usage in IRENA-1

An outline of the function of each jazz command is given in appendix D. Some examples of jazz files are examined in this chapter, which adopts the same approach as that used in chapter 10; in particular the convention that NAG names appear in upper case and IRENA names in lower case is also adopted here. We begin with the jazz files corresponding to some of the defaults files examined in chapter 10; some additional jazz files are then examined, to display points not covered in this initial set. In general, only those entries in each file which exhibit features not already discussed in this chapter are described in detail. Finally, a particularly complicated output parameterisation, which occurs in a small number of routines, is examined.

Except where otherwise indicated, the files are shown as supplied with the IRENA system, although the annotation of certain obscure points is removed to the body of the text. (For the purposes of this chapter, larger files have been split into multiple figures, either by separating input and output commands or simply by splitting the file into sections.)

11.1 Jazz files for the routines discussed in chapter 10

11.1.1 F02BJF

For this routine, the input and output commands in the jazz file are shown separately.

```
{prompt!-alias} EPS1 : tolerance! for! negligible! elements

{key!-alias} EPS1 : tne

{prompt!-alias} MATV : Are! eigenvectors! required

{set!-type} MATV : "(Y or N)"

{local} MATV [TRUE] : y

{local} MATV [FALSE] : n

{scalar} matv!-key

{qkeyword} matv!-key [1,1,1,2,2,2] : eigenvectors_required, vectors, v,
                                     no_eigenvectors_required, novectors, nov
```

Figure 11-1: F02BJF jazz file – input jazz commands

Explanation of input jazzing entries:

`{prompt!-alias} EPS1 : tolerance! for! negligible! elements`

This exemplifies the standard command used in IRENA to define the name to be used in prompting for a parameter. As described in section 9.2.3, IRENA itself prefixes the name with a type, so that the prompt (including IRENA's usual preliminary display) appears as

```
Please supply values for the following variables using the usual key-line
syntax.  '@' may be used instead of the name of the current variable.
Each input should be terminated with a ';' character.
Replying '!!;<cr>' to any prompt will abort the call.
To use the default value, simply reply ';<cr>'.

(Real scalar) tolerance for negligible elements? @=
```

However, since a default value is defined for this parameter, this prompt will appear only if the **promptall** switch is **on**.

The value of this parameter may be supplied in the keyline using the prompt-alias as a key. To avoid the need for repeated use of the REDUCE escape character (!), the present author added an automatic alias feature, which allows the underline character to be used in place of the spaces which occur in the prompt-alias, so that an alternative key is **tolerance_for_negligible_elements**. Automatic aliases are documented only at a general level, in the IRENA User Guide, [33].

```
{key!-alias} EPS1 : tne
```

This defines an abbreviated key, based on the initials of the significant words in the prompt-alias. For the benefit of those already familiar with the NAG routine, IRENA also accepts the NAG name **EPS1** as a key.

```
{prompt!-alias} MATV : Are! eigenvectors! required
{set!-type} MATV : "(Y or N)"
{local} MATV [TRUE] : y
{local} MATV [FALSE] : n
{scalar} matv!-key
{qkeyword} matv!-key [1,1,1,2,2,2] : eigenvectors_required, vectors, v,
                                     no_eigenvectors_required,
                                     novectors, nov
```

This block of instructions was originally generated – together with the **MATV** and **matv-key** defaults discussed in section 10.1.1 – by the program mentioned in section 15.2 and was later modified to use the **set-type** command, rather than **boolean**.

The first of these commands sets the prompt used by IRENA, as already discussed. The second overrides the type with which IRENA would prefix this prompt, so that the whole appears as

(Y or N) Are eigenvectors required?

The third and fourth allow the use of the "very local constants" **y** and **n**, to represent the Fortran logical constants **.TRUE.** and **.FALSE.** as values of the parameter **MATV**, thus permitting the user to respond naturally to the prompt for this parameter. The fifth of this set of commands defines an IRENA scalar, for communication

with the defaults system, and the last provides for the representation of the two possible values of this scalar by various keywords, usable in the function call. The defaults file translates the two values into the actual values (**.TRUE.** and **.FALSE.**) required for **MATV**. This instruction uses **qkeyword** rather than **keyword** in order to bypass IRENA's normal prompting mechanism for keywords, which offers the user a choice of the possible keywords; here we wish to use the prompt **Are eigenvectors required?** instead.

```
{output} A : !*noname!*a

{output} B : !*noname!*b

{cmplxquots} ALFR, ALFI, BETA : '((getval 'N)
                                  eigenvalues
                                  infinite_eigenvalue_warning
"'Eigenvalues' includes one or more infinite values, denoted '*'."
                                  indeterminate_eigenvalue_warning
"'Eigenvalues' includes ratios of small numbers, denoted '%', which may
represent indeterminate values.  The presence of any indeterminate value
casts doubt on the validity of all calculated values.  Please inspect
'eigenvalue_numerators' and 'eigenvalue_denominators' for small values."
                                  indeterminate_eigenvalue_warning
"'Eigenvalues' includes ratios of small numbers, denoted '%', which may
represent indeterminate values.  The presence of any indeterminate value
casts doubt on the validity of all calculated values.  Please inspect
'eigenvalue_numerators' and 'eigenvalue_denominators' for small values.

'Eigenvalues' also includes one or more infinite values, denoted '*'."
                                  eigenvalue_numerators
                                  eigenvalue_denominators)

{outputconj} ALFI, V : '((or (equal (getval 'MATV) 'TRUE)
                             (equal (caaar (caadr (getval 'MATV))) 'y))
                         normalized_eigenvectors_as_columns)

{output} ITER : iterations_for_each_eigenvalue

{output!-order} eigenvalues,
                infinite_eigenvalue_warning,
                indeterminate_eigenvalue_warning,
                eigenvalue_numerators,
                eigenvalue_denominators,
                normalized_eigenvectors_as_columns,
                iterations_for_each_eigenvalue
```

Figure 11-2: **F02BJF** jazz file – output jazz commands

Explanation of output jazzing entries:

```
{output} A : !*noname!*a
{output} B : !*noname!*b
```

The parameters **A** and **B** are described in the NAG Library manual as *Input/Output* but the only description of their output rôle is "the array is overwritten".

These commands prevent the appearance of **A** and **B** in the output list generated by IRENA, as described in section 9.3.2. An alternative approach would have been to hand tailor the specfile, as described in chapter 3, so that **A** and **B** were treated as purely input parameters when the various system files were generated. However, the present approach requires less manual intervention, especially if files are regenerated at a later release of the NAG Library, and allows for any possible later definition of information held here on output to be handled by a minimal change to the jazz file only.

The *Input/Output* description in the NAG documentation was probably introduced by the skeleton document generator[1] used by NAG to partially automate the production of routine documents by processing Fortran sources.

```
{cmplxquots} ALFR, ALFI, BETA : '((getval 'N)
                                  eigenvalues
                                  infinite_eigenvalue_warning
"'Eigenvalues' includes one or more infinite values, denoted '*'."
                                  ⋮
'Eigenvalues' also includes one or more infinite values, denoted '*'."
                                  eigenvalue_numerators
                                  eigenvalue_denominators)
```

This command takes the three NAG output arrays, **ALFR**, **ALFI** and **BETA** and builds an IRENA output vector of extended complex numbers, called **eigenvalues**, as a REDUCE column matrix, whose length is given by the input value of the NAG parameter **N**. The vector **eigenvalues** is calculated as (**ALFR** + i **ALFI**)/**BETA**, with the point at infinity represented by * and possibly indeterminate values by %. Values

[1] A less sophisticated version of this generator is a component of NAG's "NAGWare f77 Tools", described in, for instance, [23].

are regarded as "possibly indeterminate" if the corresponding elements of all three of the arrays **ALFR**, **ALFI** and **BETA** are less than a threshold value[2], set to 10^{-10}.

If infinite values but no possibly indeterminate values occur, an extra IRENA output parameter called **infinite_eigenvalue_warning** is generated, containing the first text string shown:

```
'Eigenvalues' includes one or more infinite values, denoted '*'.
```

The presence of possibly indeterminate values similarly generates an extra IRENA output parameter called **indeterminate_eigenvalue_warning**: in the absence of infinities, this contains the second of the text strings:

```
'Eigenvalues' includes ratios of small numbers, denoted '%',
which may represent indeterminate values.  The presence of
any indeterminate value casts doubt on the validity of all
calculated values.  Please inspect 'eigenvalue_numerators'
and 'eigenvalue_denominators' for small values.
```

in their presence, it contains the third text string, which repeats the second but also contains a separate paragraph regarding the infinities:

```
'Eigenvalues' also includes one or more infinite values,
denoted '*'.
```

When possibly indeterminate values are present, two further output vectors, called **eigenvalue_numerators** and **eigenvalue_denominators**, are generated. The intention here, as implied by the message texts, is that the user should inspect the small values in these structures, to decide what further action is required.

```
{outputconj} ALFI, V : '((or (equal (getval 'MATV) 'TRUE)
                             (equal (caaar (caadr (getval 'MATV))) 'y))
                         normalized_eigenvectors_as_columns)
```

This command unpacks a special, compressed NAG representation for complex eigenvectors, stored in the columns of the two-dimensional array **V**, in which

[2]Ideally, users calling the NAG Library from Fortran programs would perform similar checks on their results from this routine. The inclusion of special symbols for possibly troublesome results and of output parameters which draw attention to the presence of these should increase the likelihood of users taking appropriate action in doubtful cases.

real-valued vectors are represented as themselves but conjugate pairs of complex-valued vectors are represented by the real part and the imaginary part of one of the vectors. Here, **ALFI** is acting as an indicator: a zero entry signals the presence of a real eigenvector in the corresponding column.

The NAG parameters are processed to produce a complex-valued output matrix, whose name is given by the second element of the Lisp list on the right, only if the first element of list evaluates to **T**. This corresponds to the NAG input parameter **MATV** having the value **.TRUE.**, which indicates that eigenvectors are to be produced, and is normally recognised by testing for equality between the REDUCE copy of this parameter and the IRENA constant **TRUE**; this test is carried out in the first disjunct of the **or**. However, **MATV** may be set to **.TRUE.** via the "very local constant" **y**. As implemented by Dewar, this does not cause the REDUCE copy of the parameter to be set to **TRUE** but is, instead, detected and interpreted directly at the Fortran generation stage; consequently, we also test (in the second disjunct) for **y**, which REDUCE stores as a "standard quotient", **(*sq ((((y . 1) . 1)) . 1))**.

{output} ITER : iterations_for_each_eigenvalue

Here, **output** is used in its simplest mode, renaming a NAG output parameter to be more descriptive.

```
{output!-order} eigenvalues,
                infinite_eigenvalue_warning,
                indeterminate_eigenvalue_warning,
                eigenvalue_numerators,
                eigenvalue_denominators,
                normalized_eigenvectors_as_columns,
                iterations_for_each_eigenvalue
```

The **output-order** command determines the order in which the names of actual IRENA output parameters are displayed in the output list. The eventual display consists of those of the listed names which have actually been used for output objects, followed by the names of any other output objects (although, in fact, the **output-order** lists in the IRENA-1 jazz files are believed to be exhaustive).

11.1.2 E04GCF

This jazz-file illustrates two new output commands, **i2o** and **reshape-output**, and a new use of **output**.

```
% e04gcf jazz file

{prompt!-alias} M : number! of! residuals

{key!-alias} M : noresids

{prompt!-alias} X : starting! point

{key!-alias} X : start

{output} X : location_of_minimum

{output} FSUMSQ : minimum_sum_of_squares

{prompt!-alias} LSFUN2 : residual

{key!-alias} LSFUN2 : f

{i2o} M : !*noname!*number_of_residuals

{scalar} ns

{output} W[ns : ns + N - 1] : singular_values_of_estimated_jacobian_of_f

{reshape!-output} W : '((iname .
                         right_singular_vectors_of_estimated_jacobian_of_f)
                        (rowtrim . (((plus (getval 'ns) (getval 'N)) .
                                     (plus (getval 'ns)
                                           (getval 'N)
                                           (times (getval 'N) (getval 'N))
                                           (minus 1)))))
                        (dims . ((getval 'N) . (getval 'N))))

{output!-order} location_of_minimum, minimum_sum_of_squares,
                singular_values_of_estimated_jacobian_of_f,
                right_singular_vectors_of_estimated_jacobian_of_f

end; % of e04gcf jazz file
```

Figure 11-3: **E04GCF** jazz file

```
{i2o} M : !*noname!*number_of_residuals
```

The i2o command makes a copy of a NAG input parameter available as an IRENA output parameter – that is, as a REDUCE object. It is used in this case because a call to **E04GCF** may be followed by one to **E04YCF**, to obtain the variance-covariance matrix of the regression coefficients used by **E04GCF**; this requires the input value of **M** used by **E04GCF** as one of its parameters. Since this is not of interest to potential users as an output parameter, it is given a ***noname*** prefix to inhibit its appearance in the output list.

```
{output} W[ns : ns + N - 1] : singular_values_of_estimated_jacobian_of_f
```

This form of **output** takes the indicated section of the one-dimensional "workspace" array **W** and converts it into an IRENA output object called **singular_values_of_estimated_jacobian_of_f**. This and the output parameter generated by the next entry are also required for **E04YCF** but since, unlike **M**, their value is not trivially apparent to the user they are also made available as normal output parameters.

The value of the IRENA scalar **ns**, used in defining the required section of **W**, was calculated in the corresponding defaults file (an example of using an interaction between the defaults and jazzing systems, to be discussed in section 15.2).

```
{reshape!-output} W : '((iname .
                         right_singular_vectors_of_estimated_jacobian_of_f)
                        (rowtrim . (((plus (getval 'ns) (getval 'N)) .
                                     (plus (getval 'ns)
                                           (getval 'N)
                                           (times (getval 'N) (getval 'N))
                                           (minus 1)))))
                        (dims . ((getval 'N) . (getval 'N))))
```

This command extracts a matrix of column vectors from a section of the one-dimensional array **W**. It was written by the present author as a "mock-up" for a possible second generation general IRENA output jazzing facility and, as such, simulates a "key and value" syntax by using dotted key and value pairs in the Lisp list on the right. (In other words, this is an association list.) The "key" **iname** specifies the name of the IRENA output matrix, **rowtrim** the range of rows (or

elements, in this case) of **W** to be used and **dims** the dimensions of the output matrix. Other "keys" are described in appendix D.

The fact that the last two items contain useful information is documented in the NAG manual only under **E04YCF**.

11.1.3 E02ADF and E02AEF

New features in these files are the input jazzing command **newscalar**, the output command **lower** and the use of a selection of commands to extract and pass the information, needed to normalise **X** in the **E02AEF** defaults file, from **E02ADF** to **E02AEF**.

Dealing first with the **E02ADF** jazz file:

```
{newscalar} KPLUS1 [degree+1] : degree
{prompt!-alias} degree : maximum! degree! of! polynomial! fit! required
{key!-alias} degree : k
```

The NAG routine requires the user to supply the parameter **KPLUS1** which is one more than the maximum degree of polynomial approximation required. IRENA instead works with the maximum degree and uses the prompt **maximum degree of polynomial fit required**. The names **degree**, defined by the **newscalar** command, and **k**, defined by **key-alias**, are available as alternative keys.

```
{lower} A : chebyshev_coefficient_sets
```

This command extracts from the two-dimensional NAG array **A** the lower triangular matrix **chebyshev_coefficient_sets**.

```
{build!-rectangle} : '((iname . domain_of_definition)
                       (lower . ((in xmin)))
                       (upper . ((in xmax))))
```

This output command takes the values determined in the **E02ADF** defaults file for the IRENA scalars **xmin** and **xmax** and combines them into a "rectangle", the natural IRENA structure for storing interval information in any number of dimensions. It uses the same syntactic style as **reshape-output**. The values defined by the **lower** and **upper** keys are specified to be input parameter values, since IRENA scalars mimic NAG input parameters.

```
% e02adf jazz file

{newscalar} KPLUS1 [degree+1] : degree

{prompt!-alias} degree : maximum! degree! of! polynomial! fit! required

{key!-alias} degree : k

{prompt!-alias} X : old! points

{key!-alias} X : points

{prompt!-alias} Y : old! values

{key!-alias} Y : values

{scalar} xmin, xmax

{build!-rectangle} : '((iname . domain_of_definition)
                       (lower . ((in xmin)))
                       (upper . ((in xmax))))

{prompt!-alias} W : weights

{lower} A : chebyshev_coefficient_sets

{output} S : root_mean_square_residuals

{output!-order} chebyshev_coefficient_sets, root_mean_square_residuals

end; % of e02adf jazz file
```

Figure 11-4: **E02ADF** jazz file

In the **E02AEF** jazz file:

{rectangle} xmin, xmax : x_range

This input jazz command defines a rectangle to represent the two IRENA scalars. The name defined here is effectively a key-alias; the prompt-alias is defined by the next command:

{prompt!-alias} x_range : domain! of! definition

to be the same as the name used for this structure by the **E02ADF** jazz file.

```
% e02aef jazz file

{newscalar} NPLUS1 [degree+1] : degree

{key!-alias} degree : n

{prompt!-alias} A : chebyshev! coefficients

{key!-alias} A : coefficients

{key!-alias} A : coefs

{scalar} xmin, xmax, x

{rectangle} xmin, xmax : x_range

{prompt!-alias} x_range : domain! of! definition

{prompt!-alias} x : new! point

{prompt!-alias} XCAP : normalized! x

{output} P : new_value

end; % of e02aef jazz file
```

Figure 11-5: **E02AEF** jazz file

Of course, **xmin** and **xmax** are defined as **scalars** in both jazz files. The use of a rectangle to represent these means that, if their values are required and not supplied to **e02aef**, it will prompt for this natural structure.

11.1.4 D01BBF

This jazz file illustrates the use of the input jazz command **keyword** and the use of **case** in the **output** command.

```
{keyword} formula [1,1,2,2] : gauss_laguerre, gla,
                              gauss_rational, gra
{qkeyword} formula [1,2] : laguerre, rational
```

The IRENA scalar **formula** is used in the defaults file to select one of the two quadrature formulae available for semi-infinite ranges. (For finite and doubly-infinite ranges, **D01BBF** offers no choice of quadrature formula.)

```
% d01bbf jazz file

{scalar} lowerlimit, upperlimit, formula, parameter_a, parameter_b

{keyword} formula [1,1,2,2] : gauss_laguerre, gla,
                              gauss_rational, gra

{qkeyword} formula [1,2] : laguerre, rational

{rectangle} lowerlimit, upperlimit : range

{keyword} ITYPE [0,0,1,1] : normal_weights, nw, adjusted_weights, aw

{i2o} ITYPE : itype

{output} itype : case ITYPE (0,1) type_of_weighting_used,
                                  type_of_weighting_used,
                                  !*noname!*itype

{prompt!-alias} N : number! of! points! to! be! used

{output} WEIGHT : weights

{output} ABSCIS : abscissae

{output!-order} weights, abscissae

end; % of d01bbf jazz file
```

Figure 11-6: D01BBF jazz file

The **keyword** command, as well as allowing the specified strings to be used as keywords in the IRENA keyline, also causes them to be offered as alternatives if IRENA prompts for **formula**; the prompt is produced with two options per line, to emphasise that these form pairs with a common meaning. In this case, the abbreviated form uses three letters, to avoid confusion between Gauss-Laguerre quadrature and Gauss-Legendre, used for finite ranges.

The **qkeyword** command here provides a third choice of form for each keyword, which users may employ in the keyline, omitting the redundant **gauss_** component. As previously noted, **qkeyword** does not affect the generated prompt.

```
{i2o} ITYPE : itype
{output} itype : case ITYPE (0,1) type_of_weighting_used,
                                  type_of_weighting_used,
                                  !*noname!*itype
```

As **D01BBF** offers a choice of weighting strategies on semi-infinite ranges, of which one is chosen as the IRENA default, the input parameter **ITYPE** is re-output by IRENA, to inform the user of the strategy used. The **case** construct in the **output** command will give the output parameter **itype** the name **type_of_weighting_used** when the input parameter **ITYPE** has the value **0** or **1** (the possible values when there is a choice of strategy); the name ***noname*itype** will be used to hide this output parameter otherwise. Except when there is a choice of strategy, the defaults file gives **ITYPE** the value **unset**.

In fact, a better interface could be provided, by implementing a scalar version of the **interpret** output jazz command, which replaces numeric codes in an output array with strings interpreting these codes. **D01BBF** was processed at quite an early stage of the IRENA project, before the **interpret** command was added; when **interpret** was added, its potential applicability to this routine was apparently overlooked. The **D01BBF** jazz file has been commented to suggest the use of this strategy in a future release.

11.1.5 F04MAF

Here again, for convenience of display, the jazz file has been split between two figures, respectively showing input and output commands.

Input

```
{ragged!-in} INFORM : '((iname . details_of_factorization) 1)
{ragged!-in} inform1 : '((iname . details_of_factorization) scalar 1 1)
                                  ⋮
{ragged!-in} IKEEP : '((iname . details_of_factorization) 7)
```

Most of the input information required by this routine is output by **F01MAF**, which is used to pre-factor the coefficient matrix. As this information is not designed to be readily meaningful to human readers, it is packed into a single ragged array by

```
{prompt!-alias} N : order! of! matrix! A

{key!-alias} N : aorder

{scalar} inform1, inform2

{ragged!-in} INFORM : '((iname . details_of_factorization) 1)

{ragged!-in} inform1 : '((iname . details_of_factorization) scalar 1 1)

{ragged!-in} inform2 : '((iname . details_of_factorization) scalar 1 2)

{ragged!-in} NZ : '((iname . details_of_factorization) scalar 2 1)

{ragged!-in} AVALS : '((iname . details_of_factorization) 3)

{ragged!-in} IRN : '((iname . details_of_factorization) 4)

{ragged!-in} ICN : '((iname . details_of_factorization) 5)

{ragged!-in} WKEEP : '((iname . details_of_factorization) 6)

{ragged!-in} IKEEP : '((iname . details_of_factorization) 7)

{prompt!-alias} B : right!-hand! side

{key!-alias} B : rhs

{silent!-alias} B : right_hand_sides

{silent!-alias} B : rhss

{scalar} cc, mni

{prompt!-alias} cc : convergence! criterion

{prompt!-alias} mni : maximum! number! of! iterations

{scalar} ifailb, ifailc

{keyword} ifailb [1,1,0,0] : error_messages, em, no_error_messages, noem

{keyword} ifailc [1,1,0,0] : monitoring, mon, no_monitoring, nomon
```

Figure 11-7: **F04MAF** jazz file – input jazz commands

F01MAF's jazz file, using the **ragged-out** command, and unpacked here using **ragged-in**; this allows a single structure, **details_of_factorization**, to be passed between the two IRENA-functions. The RLISP functions which provide this functionality are designed to be easily generalised to allow higher dimensional arrays as components.

Entire rows of the ragged array may be extracted by **ragged-in** to provide NAG one-dimensional input arrays; individual elements of rows can be used to provide NAG scalars. In the scalar case, the word **scalar** precedes the address of the scalar in the ragged array. The information that a scalar is being processed is required so that a scalar assignment may be generated as part of the Fortran code: otherwise a sequence of array element assignments must be generated. As **ragged-in** is used here, the presence of a scalar could be detected automatically by means of its two-component address; however, in principle, the elements of the list which forms the ragged array could be lists nested to any depth, so that this technique would preclude later generalisation.

As the first two elements of **INFORM** are needed in calculations in the defaults file, **ragged-in** is used to extract these separately into the IRENA scalars **inform1** and **inform2**, as well as to extract the entire array **INFORM**.

```
{silent!-alias} B : right_hand_sides
{silent!-alias} B : rhss
```

The **silent-alias** command is a minor feature of IRENA which allows an alias to be used without being explicitly documented by the IRENA skeleton document generator – it is otherwise completely equivalent to the **key-alias** command. It is used mainly where singular and plural forms of the same word form the natural parameter name in related routines, to allow the alternative form, for consistency, and in quadrature routines, where the names **range** and **region** are appropriate for one- and multi-dimensional quadrature, respectively, again to allow the alternative. The possibility of such forms is documented at a general level in [33].

```
{i2o} cc : convergence_criterion_used

{output} ACC(2) : rms_residual_of_normalized_equations

{output} NOITS(2) : number_of_iterations

{output} WORK(1) : lower_bound_on_condition_number_of_a

{output!-order} solution,
                convergence_criterion_used,
                rms_residual_of_normalized_equations,
                lower_bound_on_condition_number_of_a,
                number_of_iterations
```

Figure 11-8: **F04MAF** jazz file – output jazz commands

Output

`{output} ACC(2) : rms_residual_of_normalized_equations`

This demonstrates the use of **output** to extract a NAG array element into a REDUCE scalar.

11.2 Further jazz files

11.2.1 E04MBF

E04MBF is described in [27] as "an easy-to-use routine for solving linear programming problems or for finding a feasible point ... not intended for large sparse problems."

```
% e04mbf jazz file

{prompt!-alias} ITMAX : maximum! number! of! iterations

{key!-alias} ITMAX : maxits

{key!-alias} ITMAX : mni

{set!-type} MSGLVL : "Please select monitoring level"

{keyword} MSGLVL [-1,-1,0,0,1,1,2,2] : no_printing, np,
                                       error_printout, ep,
                                       solution_printout, sp,
                                       full_diagnostics, fd

{phased!-prompt} A : Are! there! general! linear! constraints
                     (n > unset) linear! constraint! coefficients

{qkeyword} A [unset,unset] : no_linear_constraints, nlc

{key!-alias} A : lcc

% Separating "bounds on variables" from "bounds on general linear constraints"

{vector} lbv, lblc, ubv, ublc

{concatenate} BL : '(lbv lblc)

{local} BL [-fphuge] : times

{prompt!-alias} lbv : lower! bounds! on! variables

{prompt!-alias} lblc : lower! bounds! for! linear! constraints
```

Figure 11-9: E04MBF jazz file – part 1

```
{concatenate} BU : '(ubv ublc)

{local} BU [fphuge] : times

{prompt!-alias} ubv : upper! bounds! on! variables

{prompt!-alias} ublc : upper! bounds! for! linear! constraints

{rectangle} lbv,ubv : bv

{prompt!-alias} bv : bounds! on! variables

{rectangle} lblc,ublc : blc

{prompt!-alias} blc : bounds! for! linear! constraints

{phased!-prompt} CVEC : Do! you! only! want! a! feasible! point
                    (y > unset) objective! function! coefficients

{key!-alias} CVEC : ofc

{key!-alias} CVEC : c    % Used in the mathematical description of this.

{qkeyword} CVEC [unset, unset] : feasible_point_only, fpo

{prompt!-alias} X : starting! point

{key!-alias} X : start

{output} X : case LINOBJ (TRUE) x!*location_of_minimum,
                                x!*feasible_point

{output} x!*location_of_minimum : case IFAIL out(2,3,4) final_point,
                                                        final_point,
                                                        final_point,
                                                        location_of_minimum

{output} x!*feasible_point : case IFAIL out(1)
                                        point_of_least_infeasibility,
                                        feasible_point
```

Figure 11-10: **E04MBF** jazz file – part 2

```
{interpret} ISTATE : '((iname status_of_constraints_on_variables
                              status_of_linear_constraints)
                       (retain . !*noname!*istate)
                       (trim ((1 . (getval 'N)))
                             (((plus 1 (getval 'N)) . (getval 'NCTOTL))))
                       (keys (-2 . lower! violation)
                             (-1 . upper! violation)
                             ( 0 . free)
                             ( 1 . lower! limit)
                             ( 2 . upper! limit)
                             ( 3 . equality! held)
                             ( else . error!:! please! inform! nag)))

{output} OBJLP : case LINOBJ (TRUE) objlp!*of,
                                    objlp!*suminf

{output} objlp!*of : case IFAIL out(2,3,4)
                                     objective_function_at_final_point,
                                     objective_function_at_final_point,
                                     objective_function_at_final_point,
                                     minimum_value

{output} objlp!*suminf : case IFAIL out(1) sum_of_infeasibilities,
                                           !*noname!*objlp!*suminf

{output} CLAMDA[1:N] : lagrange_multipliers_for_constraints_on_variables

{output} CLAMDA[N+1:NCTOTL] :
                     lagrange_multipliers_for_general_linear_constraints

{output!-order} X, OBJLP,
                status_of_constraints_on_variables,
                status_of_linear_constraints,
                lagrange_multipliers_for_constraints_on_variables,
                lagrange_multipliers_for_general_linear_constraints

end; % of e04mbf jazz file
```

Figure 11-11: **E04MBF** jazz file – part 3

As well as providing more examples of various commands already covered, some of them illustrating additional features or use in more complex situations, the jazz file for **E04MBF** also illustrates the use of **phased-prompt** and **interpret**:

```
{set!-type} MSGLVL : "Please select monitoring level"
{keyword} MSGLVL [-1,-1,0,0,1,1,2,2] : no_printing, np,
                                       error_printout, ep,
                                       solution_printout, sp,
                                       full_diagnostics, fd
```

Here, **set-type** is used to insert a preamble to an IRENA prompt which is not, in itself, a type (and so not enclosed in parentheses). When used in this way in conjunction with a **keyword** instruction, **set-type** causes the defined preamble to be displayed, followed by a variant of the usual keyword prompt:

```
Please select monitoring level
(one of the following)  no_printing, np,
                        error_printout, ep,
                        solution_printout, sp,
                        full_diagnostics, fd?
```

```
{phased!-prompt} A : Are! there! general! linear! constraints
                     (n > unset) linear! constraint! coefficients
{qkeyword} A [unset,unset] : no_linear_constraints, nlc
{key!-alias} A : lcc
```

If no value has been supplied for **A**, the **phased-prompt** command first causes the initial prompt

(Y or N) Are there general linear constraints?

to be issued. If the response to this is **n**, **A** is given the value **unset**, otherwise the prompt-alias **linear constraint coefficients** is used in the normal prompting mechanism. A **qkeyword** command is used to provide keys corresponding to the **n** response and a key-alias defines an alternative key using the initials of the prompt-alias.

The next section of the jazz file deals with replacing the NAG parameterisation of various bounds. **E04MBF** requires two input arrays **BL** and **BU**, respectively holding lower and upper

bounds; in each case, bounds on variables are followed by bounds on linear constraints. The number of variables (and of bounds on these in each array) is given in a separate parameter **N**.

In the parameterisation chosen for IRENA, the bounds on variables are represented separately from the bounds on linear constraints; in each case, the lower and upper pairs are supplied in a rectangle. (The value of **N**, needed by the NAG routine, can then be automatically determined from the length of the rectangle **bounds on variables**, in the defaults file.)

{vector} lbv, lblc, ubv, ublc

This introduces the non-scalar local variables which will serve as intermediaries between the IRENA and NAG parameterisations.

{concatenate} BL : '(lbv lblc)
{concatenate} BU : '(ubv ublc)

The NAG arrays are formed by concatenating the entries in the appropriate intermediaries.

{local} BL [-fphuge] : times
{local} BU [fphuge] : times

E04MBF recognises entries less than -10^{20} and greater than 10^{20}, respectively, as representing the absence of a lower or upper bound. Other NAG routines use a different "cut-off" point in this context; in some cases, this is determined by the user. For simplicity, all IRENA jazz files use the largest "safely" representable floating point number, **fphuge**[3], (or its negative) here; this is certainly greater than 10^{20}. Specifying * (which REDUCE interprets as **times**) as a "very local constant" meaning **fphuge** therefore allows * to represent "unbounded" in the user's specification of the bounds.

{rectangle} lbv,ubv : bv
{rectangle} lblc,ublc : blc

The intermediaries are obtained as the components of the IRENA "rectangles" whose key-aliases are **bv** and **blc**.

[3] See **fphuge** in the glossary.

```
{prompt!-alias} bv : bounds! on! variables
{prompt!-alias} blc : bounds! for! linear! constraints
```

The names used by IRENA in prompting for the rectangles are defined here. The other prompt-aliases, for the quantities **lbv**, **lblc**, **ubv** and **ublc**, are only used in prompts in the (unlikely) event that the user has supplied upper or lower bounds alone; they also serve a minor documentation rôle in the jazz file.

The next block of **output** commands shows how compound conditionals may be handled in the renaming of output parameters:

```
{output} X : case LINOBJ (TRUE) x!*location_of_minimum,
                                x!*feasible_point
{output} x!*location_of_minimum : case IFAIL out(2,3,4) final_point,
                                                        final_point,
                                                        final_point,
                                                        location_of_minimum
{output} x!*feasible_point : case IFAIL out(1)
                                          point_of_least_infeasibility,
                                          feasible_point
```

E04MBF may be used either to minimise the specified objective function or to find a feasible point, depending on the input value of **LINOBJ**. The first **output** command distinguishes between these two cases, (temporarily) renaming **X** on output to an appropriate identifier in each case.

If the routine returns a non-zero **IFAIL** value, **X** will contain information indicating how far the solution process has progressed, otherwise it contains the solution to the chosen problem. The second and third **output** commands select names appropriate to unsuccessful and successful solution of the two types of problem, conditional on the output value of **IFAIL**.

IRENA provides the means of replacing coded values in an output array with descriptive strings, by means of the **interpret** command:

```
{interpret} ISTATE : '((iname status_of_constraints_on_variables
                              status_of_linear_constraints)
                       (retain . !*noname!*istate)
                       (trim ((1 . (getval 'N)))
                             (((plus 1 (getval 'N)) . (getval 'NCTOTL))))
                       (keys (-2 . lower! violation)
                             (-1 . upper! violation)
                             ( 0 . free)
                             ( 1 . lower! limit)
                             ( 2 . upper! limit)
                             ( 3 . equality! held)
                             ( else . error!:! please! inform! nag)))
```

The NAG output parameter **ISTATE** is split into two REDUCE column matrices, **status_of_constraints_on_variables** and **status_of_linear_constraints**, using the information specified in the **trim** option. The **retain** option specifies that the REDUCE equivalent of the NAG output parameter should also be formed, but, because of the ***noname*** prefix, omitted from the output list; this provides for the parameter's use on input by other routines. The actual text strings to be used to replace the various output values are specified by the **keys** option: each of the six possible values given in the NAG documentation is transformed into a descriptive string; a seventh string indicates that an undocumented value has been detected.

11.2.2 E01SEF and E01SFF

E01SEF generates a C^1, piecewise polynomial, two-dimensional surface interpolating a set of scattered data points; **E01SFF** evaluates the interpolant at a given point.

The IRENA function **e01sff** extends **E01SFF** by using an outer Fortran subprogram (**IE01SF**) – otherwise referred to as a *jacket* – which makes multiple calls to the NAG routine and so evaluates the interpolant at a grid of points. This serves both to provide a simpler user interface and to limit the time taken in processing a sequence of interpolations. (This is discussed further in section 12.1.)

The function **e01sff** is an exception to the general IRENA rule that the NAG form of input parameters should remain acceptable, in that a single interpolation point must also be represented as a grid, not by the two parameters **PX** and **PY**.

```
% e01sef jazz file

{tuples1} X : 'data_set

{tuples2} Y : 'data_set

{tuples3} F : 'data_set

{prompt!-alias} data_set : original! data! set

{key!-alias} data_set : ds

{prompt!-alias} RNW : radius! for! zero! weights

{key!-alias} RNW : rzw

{output} RNW : radius_for_zero_weights

{prompt!-alias} RNQ : local! data! radius

{key!-alias} RNQ : ldr

{output} RNQ : local_data_radius

{prompt!-alias} NW :
         average! number! of! points! within! radius! for! zero! weights

{key!-alias} NW : nrzw

{prompt!-alias} NQ :
         average! number! of! points! within! local! data! radius

{key!-alias} NQ : nldr

{output} FNODES : quadratic_nodal_function_coefficients

{message} MINNQ : '(data_density_warning (evallessp (fortran!-value 'MINNQ) 5)

"Fewer than 5 data points are local to some node(s), in whose
vicinity linear, not quadratic, interpolation has been used."

                              fewest_local_data_points)

{output!-order} quadratic_nodal_function_coefficients,
                local_data_radius, radius_for_zero_weights,
                fewest_local_data_points, data_density_warning

end; % of e01sef jazz file
```

Figure 11-12: **E01SEF** jazz file

```
% e01sff jazz file

{tuples1} X : 'data_set

{tuples2} Y : 'data_set

{tuples3} F : 'data_set

{prompt!-alias} data_set : original! data! set

{prompt!-alias} RNW : radius! for! zero! weights

{key!-alias} RNW : rzw

{prompt!-alias} FNODES : quadratic! nodal! function! coefficients

{template} ie01sf : grid
{template} ie01sf : ~grid

{gridfirst} GX : 'new_points

{gridsecond} GY : 'new_points

{prompt!-alias} new_points : new! points

{output} PF : !*noname!*pf

{output} GF : new_values

end; % of e01sff jazz file
```

Figure 11-13: **E01SFF** jazz file

These jazz files illustrate the use of **template**, **grid** and **tuples** in input jazzing and of **message** in output jazzing.

```
{tuples1} X : 'data_set
{tuples2} Y : 'data_set
{tuples3} F : 'data_set
{prompt!-alias} data_set : original! data! set
{key!-alias} data_set : ds
```

The NAG parameters **X**, **Y** and **F**, representing the coordinates of the data points, may be supplied to the IRENA functions as a "list of n-tuples" – that is, a list of lists of the same length – with the prompt-alias **original data set** and key-aliases

data_set and ds. (The key-alias ods has since been added to the jazz files of this and similar routines, to reflect the policy that an abbreviated key-alias should correspond, where feasible, to the initials of the prompt-alias.)

```
{message} MINNQ : '(data_density_warning (evallessp (fortran!-value 'MINNQ) 5)
"Fewer than 5 data points are local to some node(s), in whose
vicinity linear, not quadratic, interpolation has been used."
                                    fewest_local_data_points)
```

The IRENA output parameter **data_density_warning** is generated if the output value of the NAG parameter **MINNQ** is less than 5; the fact that linear interpolation is used locally in this case is mentioned in the "Description" section of the NAG **E01SEF** routine document but does not generate a warning **IFAIL** value. The parameter **MINNQ** itself is renamed as **fewest_local_data_points**.

The **message** mechanism provides a useful two-level error mechanism, in that the variable name which appears in the output list, in the event of a warning being necessary, is suggestive of the cause of the problem and the content of the variable is a fuller description of this. Extending this mechanism to replace the present method of dealing with non-zero **IFAIL** returns could usefully be considered for a future project, although, for a system as comprehensive as IRENA, generating the message names and contents might be rather labour-intensive.

```
{template} ie01sf : grid
{template} ie01sf : ~grid
```

The first form of the **template** command indicates that templates corresponding to the jacketed routine **IE01SF** should be used in place of those for **E01SFF** if the key **grid** appears in the keyline; the second form indicates that these templates should be used if the key **grid** does not appear; together, they ensure that the **IE01SF** material is used unconditionally.

IE01SF provides the interface which accepts a grid of evaluation points, rather than a single point; for this routine, the effort required to allow either a single point or a grid, and thus maintain the NAG interface, was not considered worthwhile, as a very simple representation of a single point as a grid is provided (see below).

```
{gridfirst} GX : 'new_points
{gridsecond} GY : 'new_points
```

The two one-dimensional NAG input arrays **GX** and **GY** may be obtained from the IRENA grid **new_points**. This is simply a list with two entries, each of which may be a single number or a list of numbers; thus, a single point is represented as a pair of values, a transect parallel to an axis by a value and a list, and a general grid by two lists. To emphasise the manner in which the two lists define a grid, the examples supplied with IRENA display the second list vertically, as in

```
e01sff(new_points={{      3,  6,  9, 12, 15, 18, 21},
                   { 2,
                     5,
                     8,
                    11,
                    14,
                    17                              }})$
```

11.2.3 D01ALF

D01ALF evaluates a definite integral over a finite range; the integrand may have singularities at a finite number of points, which the user is expected to specify.

Its jazz file illustrates the use of the **fort-dims** input command and the **precedence** and **out-dims** output commands and of a pair of **output case** constructs to generate different forms of output in different circumstances.

```
{fort!-dims} POINTS : '((getval 'NPTS) . 1)
```

The **fort-dims** input command provides a means of specifying the dimensions to be used for an array in the generated Fortran, where the correct values cannot be determined automatically by IRENA. In this case, **POINTS** is shown in the specification section of the **D01ALF** routine document, from which its dimension would normally originally be deduced, as an assumed-size array – that is, with dimension *. As for the output use of **A** and **B** in **F02BJF**, this could be rectified in the specfile but processing it in the jazz file requires less manual intervention and is more permanent.

```
{prompt!-alias} F : integrand

{rectangle} A,B : range

{silent!-alias} range : region

{prompt!-alias}  POINTS : break! points

{key!-alias}  POINTS : bp

{fort!-dims} POINTS : '((getval 'NPTS) . 1)

{prompt!-alias} EPSABS : absolute! accuracy! required

{key!-alias} EPSABS : absacc

{prompt!-alias} EPSREL : relative! accuracy! required

{key!-alias} EPSREL : relacc

{output} RESULT : integral

{output} ABSERR : absolute_error_estimate

{precedence} IW

{output} IW(1) : iw1

{output} iw1 : case IFAIL out(0) number_of_subintervals_used, !*noname!*iw1

{output} IW(1) : !*noname!*iw1a

{out!-dims} W : '((aeval '!*noname!*iw1a) . 2)

{output} W : case IFAIL out(0) !*noname!*W, subintervals

{output} W[2*!*noname!*iw1a+1 : 3*!*noname!*iw1a] : e_list

{output} e_list : case IFAIL out(0) !*noname!*e_list,
                                error_estimates_for_subinterval_approximations

{output} W[3*!*noname!*iw1a+1 : 4*!*noname!*iw1a] : r_list

{output} r_list : case IFAIL out(0) !*noname!*r_list,
                                    integral_approximations_on_subintervals

{prompt!-alias} LW : main! workspace! length! !(restricts! subdivision!)

{key!-alias} LW : workspace
```

Figure 11-14: Body of **D01ALF** jazz file

{precedence} IW

The NAG output parameters mentioned in **precedence** commands are moved to the head of the list of output variables for which IRENA equivalents are to be generated; this is necessary if the output jazzing for other parameters depends on the values of these. In this case, the processing of W depends on that of IW, so that the latter must be processed first.

{out!-dims} W : '((aeval '!*noname!*iw1a) . 2)

The value stored in IW(1), which is transferred into *noname*iw1a, gives the number of subintervals used in the quadrature. The first section of W of this length contains the lower endpoints of the subintervals, the next section (of the same length) the upper endpoints; the **out-dims** command effectively converts these elements of W into a REDUCE matrix whose dimensions are given by this value and 2. (The entire NAG array remains available to other output jazzing commands.)

The value of IW(1) must be stored in *noname*iw1a as well as in one of number_of_subintervals_used and *noname*iw1, since the **aeval** in the **out-dims** command will be applied unconditionally to its argument.

{output} iw1 : case IFAIL out(0) number_of_subintervals_used, !*noname!*iw1
{output} W : case IFAIL out(0) !*noname!*W, subintervals

The respective positions of the *noname* components in these complementary **output** commands ensure that, when the routine terminates successfully, with IFAIL having the output value 0, the user is informed of the number of subintervals used but, in case of failure, the diagnostic information stored in the restricted W – that is, the actual subintervals – is made available instead.

11.3 Cond-out

The output jazzing command `cond-out`, as used for the routine **F02XEF**, provides an example of how complex individual jazzing commands can become.

F02XEF performs the singular value decomposition of a complex $m \times n$ matrix; it can return various components of the decomposition, depending on the settings of NAG input parameters: the singular values are always returned; if **WANTQ** is **.TRUE.** then the matrix of left-hand singular vectors is returned; if **WANTP** is **.TRUE.**, the conjugate transpose of the matrix of right-hand singular vectors is returned.

The main difficulty in providing IRENA output lies in the fact that the singular vectors are stored in different locations, depending on the characteristics of the problem; what is stored where is detailed in the NAG manual in the *On exit* sections of the descriptions of the parameters **A**, **Q** and **PH** – it is, perhaps, most easily understood as a contingency table, as in figure 11-15.

WANTQ	WANTP	$m \geq n$	Location of singular vectors	
			left	right (conjugate transpose)
T	T	T	A	PH
		F	Q	A
	F	T	A	-
		F	Q	-
F	T		-	A
	F		-	-

Figure 11-15: Location of singular vectors in **F02XEF** output

When the design of the original jazz functionality was undertaken, the possibility of the same information being output in different locations was not considered; however, the output-function facility provided the means to add the required functionality, through **cond-out**, which includes conditional output, trimming the NAG arrays and optionally forming the conjugate transpose. For wider applicability, the ability to form upper and lower triangular matrices and diagonal matrices was included. The **cond-out** command is used in four IRENA-1 jazz files; its use for **F02XEF** is shown in figure 11-16.

```
{cond!-out} A, Q, PH : '((left_hand_singular_vectors_as_columns
                          (cond ((and (evalgeq (getval 'M) (getval 'N))
                                      (evalequal (getval 'WANTQ) 'TRUE))
                                 '((A (1 . 1) ((getval 'M) . (getval 'N)))))
                                ((and (evallessp (getval 'M) (getval 'N))
                                      (evalequal (getval 'WANTQ) 'TRUE))
                                 '(Q))
                                (t nil)))
                         ((right_hand_singular_vectors_as_columns
                           . (conjugate transpose))
                          (cond ((and (evallessp (getval 'M) (getval 'N))
                                      (evalequal (getval 'WANTP) 'TRUE))
                                 '((A (1 . 1) ((getval 'M) . (getval 'N)))))
                                ((and (evalgeq (getval 'M) (getval 'N))
                                      (evalequal (getval 'WANTQ) 'FALSE)
                                      (evalequal (getval 'WANTP) 'TRUE))
                                 '((A (1 . 1) ((getval 'N) . (getval 'N)))))
                                ((and (evalgeq (getval 'M) (getval 'N))
                                      (evalequal (getval 'WANTQ) 'TRUE)
                                      (evalequal (getval 'WANTP) 'TRUE))
                                 '(PH))
                                (t nil))))
```

Figure 11-16: Cond-out usage in the **F02XEF** jazz file

11.4 Conclusion

A range of examples of the commoner input and output jazzing commands and of a number of the less common commands has been presented.

With the availability of the output-function mechanism, there is no intrinsic problem with output jazzing – output functions can be written to transform NAG output into whatever form is required. There remains, of course, the question of defining a sufficiently compact notation to handle this at an appropriate level of generality and of implementing this in an integrated system; **reshape-output** represents a prototype for some of the facilities required.

Input jazzing can be more complicated, since different representations of the same information are possible. Such cases are, in general, handled by possibly introducing extra parameters as IRENA **scalars** or **vectors** and setting the parameters which are not required to **unset**. In such cases, it may be necessary to control the order of prompting if prompts for redundant parameters are to be avoided; unfortunately, IRENA did not provide such a facility – the need for this was, in large measure, handled by means of **phased-prompt** and to some extent by reordering entries in the defaults file. However, in complicated cases, unwanted prompts may still occasionally be induced by certain combinations of keyline entries. Should this occur, users may respond with the value *, meaning unset; however, it would be preferable in a future design if this whole area could be simplified by including a general prompting dependency control for input jazzing.

Chapter 12

Jacketed routines

As mentioned briefly in section 11.2.1, a Fortran *jacket* is a Fortran subprogram which provides an alternative interface for one or more calls to other (usually pre-existing) Fortran subprograms. In this context, the pre-existing subprograms are NAG Library routines.

Initially, Fortran jackets were provided for a number of NAG routines as a "last resort", to provide additional functionality which could not easily be obtained through "standard" IRENA features such as the jazz and defaults systems. However, this proved a very powerful tool and its use will certainly be extended in future versions of IRENA and similar systems.

12.1 Reasons for developing jackets

The typical elapsed time for IRENA to go through its code generation, compilation and loading phases, when loading from the complete Mark 15 NAG Library, was about about half a minute (see section 9.1). The execution time, of course, is problem dependent but is usually negligible for small problems.

Whilst a half minute turnaround of a function call, with progress reports to the user, is just acceptable for a single call, for problems requiring multiple calls of NAG routines it soon becomes extremely tedious for the user and is inefficient in utilising computing resources for multiple compilations and loading operations. (The further inefficiency of IRENA's repeatedly

interpreting the same control files and converting the same data structures, whilst aesthetically unattractive, usually accounts for few of the resources required to produce a solution, as is exemplified in figure 9-1.) Therefore, in a number of cases, Fortran "jackets" were developed, to handle multiple NAG routine calls in a single IRENA-function call.

C06EAF	C06FPF	C06FUF	E01BHF	E02DAF
C06EBF	C06FQF	E01BFF	E01SBF	F04MCF
C06ECF	C06FRF	E01BGF	E01SFF	M01EAF

Table 12.1: NAG routines jacketed in IRENA 1.0

The jacketed routines in IRENA 1.0 are shown in table 12.1. The detailed reasons for developing jackets varied with the NAG Library "chapter".

C06 The C06 routines calculate finite Fourier transforms. For each routine there is a "complementary" routine (sometimes itself) which, if prefaced or followed or both by an appropriate complex conjugate finding routine (for general, Hermitian or multiple Hermitian sequences), calculates the inverse transform. For these, the alternative, jacketed procedure, invoked by the keyword **inverse**, consists of the sequence of routines required to calculate the inverse transform. An example is provided in figure 12-1.

```
      SUBROUTINE IC06EA(X,N,IFAIL)
      INTEGER           N, IFAIL
      DOUBLE PRECISION  X(N)
      EXTERNAL          C06EAF, C06GBF
C
      IFAIL = -1
      CALL C06EAF(X,N,IFAIL)
      IF (IFAIL.NE.0) RETURN
      IFAIL = -1
      CALL C06GBF(X,N,IFAIL)
      IF (IFAIL.NE.0) IFAIL = IFAIL + 100000
C           above should never happen!
      RETURN
      END
```

Figure 12-1: "Inverse" jacket for C06EAF

E01 One set of E01 routines calculates interpolating functions, another evaluates these at a set of points (one-dimensional interpolation) or a single point (two-dimensional interpolation). The E01 jackets, used in IRENA calls corresponding to routines in the second (evaluation) set, include the corresponding routines from both sets and are invoked by the keyword **setup** (or, to be more precise, by the absence of **no_setup**).

To allow setup information generated by jackets to be output for later reuse, the input parameters which normally carry this information are redefined as input/output in the NAG routines' specfiles and the infofile is regenerated, as described in section 12.3.

The jackets for the routines in the E01S subchapter replace a single call of the evaluating routine by calls over a rectangular grid of points, specified in the Fortran by a pair of one-dimensional arrays. The output, in this case, is a rectangular array of values, as opposed to the scalar value output by the NAG routine. An example is provided in figure 12-2.

```
      SUBROUTINE IE01SB(M,X,Y,F,TRIANG,GRADS,MGX,MGY,GX,GY,GF,IFAIL)
      INTEGER           M, TRIANG(7*M), MGX, MGY, IFAIL
      DOUBLE PRECISION  X(M), Y(M), F(M), GRADS(2,M), GX(MGX), GY(MGY),
     *                  GF(MGX,MGY)
      EXTERNAL          E01SBF
C     Local scalar
      INTEGER           IF3CNT
C
      IF3CNT = 0
      DO 40 I = 1, MGX
         DO 20 J = 1, MGY
            IFAIL = -1
            CALL E01SBF(M,X,Y,F,TRIANG,GRADS,GX(I),GY(J),GF(I,J),IFAIL)
            IF (IFAIL.EQ.1 .OR. IFAIL.EQ.2) RETURN
            IF (IFAIL.EQ.3) IF3CNT = IF3CNT + 1
   20    CONTINUE
   40 CONTINUE
      IF (IF3CNT.GT.0) IFAIL = 3
      RETURN
      END
```

Figure 12-2: Jacket for E01SBF

The distinction between the handling of the various IFAIL values is due to the fact that the values 1 and 2 signal an error in an E01SBF call but 3 is merely a warning (that extrapolation has been necessary). Whilst the occurrence of an error should terminate the execution of the jacket, a warning should certainly not do so. A more ambitious jacket might also return values indicating at which boundaries of the grid extrapolation was used. This example shows how, although the

jackets produced for IRENA were generally as modest as the particular need allowed, even here some human understanding could be required: this requirement would be increased for more ambitious jackets.

For reasons discussed in section 12.3, the jacketed **E01S** routines are called unconditionally, unlike some others for which, as we have seen, IRENA keyword parameters determine whether the jacket or the original NAG routine should be used.

E02DAF This routine, which generates a bicubic spline, requires indexing information, normally provided by **E02ZAF**, specifying an advantageous ordering of an internal matrix. The jacket performs an automatic call to **E02ZAF**, signalled by the **setup** keyword. It is also advantageous to interchange the rôles of X and Y if the length of the grid of knots is greater in the Y than in the X direction. This is also done by the jacket, if allowed by the keyword **swap**. Finally, if requested with the keyword **sort**, the jacket uses **M01EAF** and **M01ZAF** to sort the input data into the "panel order" determined by the knot set. This is by far the most comprehensive of the IRENA-1 jackets and is used unconditionally.

E04 For the two routines **E04DGF** and **E04UCF** only, the NAG Library provides its own "default parameter" mechanism, for a large selection of "optional parameters" which users occasionally wish to reset. The ability to reset the default values is provided for each of these routines by means of two auxiliary routines; for instance, in the case of **E04DGF**, the auxiliary **E04DJF** allows the user to specify the Fortran channel to which a file of parameter values is connected whilst **E04DKF** allows individual parameter values to be respecified by means of a character string argument; a corresponding pair of routines exists for **E04UCF**. These auxiliaries communicate the redefined parameter values to their principals by means of **COMMON** blocks. If users are to be provided with access to the full functionality of these routines – in particular, to optional parameters – jackets become essential. This contrasts with earlier examples, where jackets were simply a means of enhancing the user interface.

In fact, jackets were not provided for these two routines in IRENA-1, so the NAG defaults could not be overridden, there. However, a jacket for **E04DGF** was developed by the author shortly after that release and formed the basis of similar jackets, for both routines, in the Axiom-NAG interface, "NAGlink", (see section 17.1). These jackets allow the optional parameters to be treated in the same manner as all other parameters, with default values provided by the normal IRENA (or NAGlink) mechanism.

In the jackets, additional Fortran parameters corresponding to the "optional parameters" are introduced; the IRENA default values for these are empty character strings. A non-blank string provided as a value for any of these parameters produces a call to the auxiliary which resets individual parameter values. A further, additional parameter allows the user to specify the name of a file of values, mimicking the **E04DJF** style of operation; a non-blank value for this parameter causes the named file to be opened on a channel number not normally used in a NAG context and calls the other auxiliary with this channel number as a parameter.

F04MCF **F04MCF** solves a system of linear equations whose coefficient matrix has previously been factored by **F01MCF**. This is analogous to the **E01** case.

M01EAF **M01EAF** sorts a one-dimensional array into the order determined by a predefined set of ranks. If a vector of ranks is not provided, the jacket allows this to be determined, using **M01DEF**, from columns of a matrix, represented as a two-dimensional array. It then sorts the two-dimensional array by applying **M01EAF** to a sequence of its sections. The jacket is used unconditionally.

12.2 Other potential uses for jackets

In chapters 10 and 11, we encountered a number of pairs of NAG routines which are naturally used together and which would be obvious candidates for inclusion in jackets in a later release of IRENA or any similar system: **D01BBF** and **D01FBF**, **E02ADF** and **E02AEF**, **E04GCF** and **E04GYF**; these are, in general, typical of sections of their respective chapters in which routines are naturally used in pairs. As well as these, there are several other areas where an enhanced user interface could be produced for NAG Library material by developing an appropriate jacket.

12.2.1 Modular routines

The full NAG Library includes a number of routines for the solution of stiff ordinary differential equations; however, none of these is included in the Foundation Library. The appropriate sub-chapter of the Library (**D02M-D02N**) takes a modular approach to the solution of these equations – the introduction to this sub-chapter in the manual [26] indicates that the general form of a program calling a NAG stiff solver should include the steps:

call linear algebra setup routine
call integrator setup routine
call integrator
call integrator diagnostic routine (if required)
call linear algebra diagnostic routine (if appropriate and if required).

In an extended IRENA, covering this material, the user interfaces would clearly benefit from the provision of jackets, potentially handling these five calls in each case. Interfaces for the individual routines would be awkward to use and would provide little advantage over a user-written Fortran program.

12.2.2 User control of error tolerance

In a number of routines, mainly in the **D02** chapter, the user has only indirect control of the accuracy achieved: for example, in several of the **D02** routines, the documentation indicates that, for any particular problem, the error in the integrand should be proportional to the input parameter **TOL** and suggests that the accuracy achieved may be estimated by comparing the results of a call in which **TOL** equals the desired accuracy and one in which **TOL**'s value is an order of magnitude smaller.

It would be straightforward to program such a double call and accuracy determination in a jacket. It would then be a short additional step to compare the accuracy achieved with that specified and, if necessary, to make a further call, with a setting of **TOL** which should achieve the desired accuracy.

If the relationship between **TOL** and the accuracy achieved is genuinely linear, no further processing should be required; if not, it might in principle be necessary to iterate this process.

12.2.3 Reverse communication

Two of the routines in the **D02M-D02N** sub-chapter, **D02NMF** and **D02NNF**, use reverse communication – that is, they are designed to be embedded in a loop of code which takes appropriate action (perhaps performing some subsidiary calculation) following the previous call to the routine, then makes another call, if necessary. Like the other reverse communication routines in the NAG Library, they are not appropriate for immediate use in an interactive system such as IRENA. However, such routines could, in principle, be included in IRENA, if embedded in appropriate jackets.

12.2.4 Error recovery

In some NAG routines' documentation, the explanation of non-zero **IFAIL** values which indicate an unsuccessful call includes advice on how to try to circumvent the problem: the suggested strategies could often be incorporated into a jacket, removing from the user the odium of recasting the problem.

For example, in **E02GAF**, which calculates an l_1 solution for an over-determined system of equations, the description of **IFAIL** = 2 is "The calculations have terminated prematurely due to rounding errors. Experiment with larger values of TOLER or try scaling the columns of the matrix (see Section 8)." Section 8 describes how the matrix columns should be scaled and the eventual solutions rescaled to take account for this – "This should ...enable the parameter TOLER to perform its correct function" (which is to determine when small numbers can be regarded as "essentially zero").

In this case, a reasonable strategy to encode in a jacket would be, after detecting **IFAIL** = 2, to first rescale each column of the matrix and then, if **IFAIL** = 2 recurred, repeatedly double **TOLER** until either a successful call occurred or some predefined limit on the number of doublings was reached, finally rescaling the solution. As well as the normal NAG parameters, the jacket would have an extra input parameter by which the user could set this limit (with **-1** meaning "no limit") and an extra input-output parameter, in which the user could indicate whether rescaling should be allowed and the jacket could signal whether it had occurred; it would also treat **TOLER** as an input-output parameter, in which the final value used could be returned. IRENA could then use its standard conditional output and **message** jazz facilities, respectively, to produce output quantities called, say, **reset_zero_tolerance** and **warning_rescaling_used** on detecting that TOLER had changed or that rescaling had occurred.

This example illustrates a significant, additional advantage of using jackets: it is easy to arrange that the extended interface, provided by the jacket, itself interfaces cleanly with standard IRENA jazz facilities. This greatly facilitates the provision of a user interface to the underlying NAG routine which meets such IRENA design objectives as ease of use and being informative to the user; in the case of routines with unusual features, the ease of this approach contrasts with the considerable effort which may otherwise be needed to modify the jazzing system itself; such modification, in turn, is likely to reduce the uniformity and ease of use of the jazzing system.

In other routines, a particular IFAIL value often indicates that a particular solution tolerance could not be met. In these cases, a similar strategy to that described above would allow users to indicate whether a larger tolerance should be tried and to learn whether it had been.

12.2.5 Handling special features

With hindsight, greater use of jackets could have considerably reduced some of the problems encountered in trying to accommodate unusual features of NAG routines.

To take a single example, as we saw in section 10.1.4, the routine D01BBF, used in quadrature calculations, has a highly non-orthogonal representation of the range of integration, the quadrature formula to be used and the parameters of this formula. Although the final versions of the jazz and defaults files for this routine may appear fairly straightforward, the process of arriving at them was quite involved. In particular, various aspects of the defaults sytem had to be modified to take account of the fact that the names of various NAG auxiliary routines could form the value of the parameter D01XXX; literal values occur in no other IRENA-1 defaults file. At the simplest level, a jacket which used a numeric coding for these names would have avoided this necessity.

In a more comprehensive jacket, separate parameters could have been used to flag whether the lower and upper endpoints of the range were infinite and to specify the choice of quadrature formula, its parameters and the finite endpoints, wherever any of these was appropriate. Although the logic of this jacket would be essentially that of the defaults file, choosing to develop such a jacket would have isolated the problem of orthogonalising the parameters: a clear protocol for decomposing complicated tasks (such as rationalising the interface of this routine) will usually lead to the desired end result with less effort. Once the orthogonal parameterisation was achieved, simple jazzing facilities could be used to give the final form desired for the user interface.

12.3 Effectiveness of jackets

For the purposes listed in section 12.1, a Fortran jacket could be implemented quickly and easily. The 15 jackets used in IRENA 1.0 ranged in size from 12 to 79 lines of Fortran (the median was 20 and the mean 22 lines).

Once a jacket was written, it was compiled and copied to a separate library, searched on loading after the main NAG Library. The mechanism provided by Dewar for using alternative routines – in particular, jackets – required that C and Fortran templates (see, for instance, figures 3-1 and 3-2) be provided for these routines. Furthermore, the NAG routine's infofile had to include information on any parameters of the jacket which were not parameters of the original routine.

When the jacket completely replaced the original NAG routine, the procedure was comparatively straightforward: a specfile appropriate to the jacket was produced (usually by editing the NAG routine's own specfile) and templates and an infofile were generated from this by the usual automatic mechanism. The NAG routine's infofile was then replaced by the jacket's newly generated infofile.

When either the original routine or the jacket could be called the situation was slightly more complicated, since it was considered advisable to provide additional functionality within the context of the established NAG name, in order to allow users to arrive at an appropriate choice of routine from normal NAG sources and to avoid possible conflicts with the names of future NAG routines. In this case, the same three files could be generated automatically but the infofile had to be merged with that of the NAG routine and would have information about input parameters of both the NAG routine and the jacket, since, of course, the infofile is read at the start of the IRENA process, before it decides, using criteria in the jazz file, whether a jacket will be used (see figure 3-2). Those input parameters which were not applicable to the particular routine in use were defaulted to `unset` (so that no Fortran assignments were generated for them), conditional on the presence or absence of the keyword used to invoke the jacket.

No corresponding mechanism is available when the output parameters of the jacket did not match those of the NAG routine; in these cases, complete replacement of the NAG routine by the jacket was necessary. This lead to the only failures of the original design objective that the NAG parameterisation of non-housekeeping input parameters should remain available (with some essential renaming to avoid REDUCE's reserved names). However, by this point in the project, it had become apparent that the amount of effort required to meet this objective was probably not justified.

12.3.1 Utility to users

Jackets constitute one of the tools used in IRENA to provide a simpler interface to NAG routines, suitable for users who are not, necessarily, themselves Fortran programmers. The question naturally arises whether, since the jackets are Fortran routines, they might usefully be provided as part of the NAG Library, to provide similar simplification for Fortran programmers.

In some cases, the jackets would provide significantly simpler interfaces for Fortran users, at the cost of some further increase in the size of NAG Library, and would form a natural part of the existing NAG trend to supply alternative "fully comprehensive" and "easy to use" versions of routines. Examples here could include the existing **C06** and **E02DAF** jackets and the potential jackets described for **E02GAF** and the **D02** routines.

On the other hand, where the main function of the jacket is to avoid multiple IRENA cycles, there is less reason to expect it to be beneficial to Fortran programmers. A case in point would be the **E01S** routines, where the jacket replaces a single evaluation of an interpolant with evaluation at the points of a rectangular grid. Whilst this is a common requirement, it is certainly not exhaustive and writing the loops required to handle it in Fortran is straightforward.

Chapter 13

REDUCE-like interfaces

Some NAG routines, notably in the S (Special Functions) chapter, return a single value. In these cases (and as a template for users wishing to define their own NAG-based functions) it seemed appropriate to attempt to provide a more REDUCE-like interface, with IRENA functions which return the value in question, rather than a list of one value. A list of all of these functions may be found in appendix B of [33]. For consistency with the rest of IRENA, the "standard" functions were also retained.

The code required to produce these extra functions was basically very straightforward. (They were, in fact, programmed as REDUCE "algebraic" or user-level functions – rather than at the "symbolic" or system level – at a quite early stage of IRENA development.) For example, for the NAG routine `S01EAF`, which requires a single complex argument and returns its exponential[1], the following was sufficient:

[1] This particular IRENA function is clearly superfluous, since REDUCE itself can compute complex exponentials with arbitrary accuracy; it was, however, included in line with the policy that interfaces should be provided for all or none of the routines in any given chapter of the NAG Library.

```
procedure nagexp !*nag!-mnemon!-param1!*$
begin scalar !*verbose;
      !*verbose := nil;
      return first s01eaf(z=!*nag!-mnemon!-param1!*) end$
```

Those routines which can only take a real argument need to be protected against attempts to use a complex argument. For instance, for **S11ACF**, which computes the inverse hyperbolic cosine of a real value, we have:

```
procedure nagarccosh !*nag!-mnemon!-param1!*$
begin scalar !*verbose;
      !*verbose := nil;
      if impart !*nag!-mnemon!-param1!* neq 0
      then write "Real argument required for nagarccosh"
      else return first s11acf(x=!*nag!-mnemon!-param1!*) end$
```

For routines with more than one real parameter, similar tests are carried out for each. (The author was not, at this point, aware of the REDUCE **typerr** function.)

In some cases, NAG offers one routine for real arguments and another for complex arguments. As these generally return real and complex results, respectively, the argument of the IRENA function was tested (in the REDUCE code of the function) and the appropriate NAG routine called. In this way, we avoided the possibility of obtaining a result in the real case with a very small imaginary component, due to Fortran rounding errors, and possibly gained slightly more efficiency (whilst adhering to our goals of regularity and minimality).

For example, the Airy function Ai is calculated by **S17AGF** in the real case and **S17DGF** in the imaginary case. The code for this:

```
procedure nagai !*nag!-mnemon!-param1!*$
begin scalar !*verbose;
      !*verbose := nil;
      if impart !*nag!-mnemon!-param1!* = 0
      then return first s17agf(x=!*nag!-mnemon!-param1!*)
      else return first s17dgf(z=!*nag!-mnemon!-param1!*) end$
```

demonstrates a typical test for a real parameter, in the REDUCE code.

It may also happen that users have a choice of what object should be returned by a particular NAG routine, the choice being signalled by the setting of an auxiliary input parameter. For example, `S17DGF` may be used to calculate either the function Ai or its derivative. In the above example, we relied on the fact that the IRENA default is to calculate the function. A further complication in this case is that, for real arguments, NAG adopts a different strategy and provides a separate routine to calculate the derivative of Ai. IRENA enhances the uniformity of this situation by hiding the different forms required:

```
procedure nagdfai !*nag!-mnemon!-param1!*$
begin scalar !*verbose;
      !*verbose := nil;
      if impart !*nag!-mnemon!-param1!* = 0
      then return first s17ajf(x=!*nag!-mnemon!-param1!*)
      else return first s17dgf(z=!*nag!-mnemon!-param1!*,derivative) end$
```

Probably the most demanding part of this exercise was to find a set of names which were both reasonably mnemonic and reasonably short. To avoid conflicts with built-in REDUCE function names, the names of the IRENA functions were prefixed with **nag**.

13.1 An experimental higher level interface

Some time after producing the set of functions just described, the author decided to code a unified interface for the whole of the `C02` chapter, which calculates zeroes of polynomials, in the form of the function **nagpolysolve**. In part, this was an early investigation of the feasibility of generating a higher level and more mathematical interface. Higher level, because the chapter provides four separate routines, for real and imaginary polynomials, with quadratics as special cases, and more mathematical in that polynomials could be supplied in the standard REDUCE form, such as `x^5 - 3*x^2 + 2*x - 1`, rather than as, for example, the dense arrays of coefficients (possibly including many zeroes), required by the general NAG routines.

The representation of polynomials here demonstrates the general improvement in regularity which is found in IRENA. Each of the four NAG `C02` routines uses a different parameterisation of the set of coefficients. The quadratic solvers express these as individual real scalars (three in the real case, six in the complex), the general real solver has a one-dimensional real array and the

general complex solver uses an unusual representation with a real array of dimension $(n+1) \times 2$ (where n is the degree of the polynomial). As a consistent representation is inherited from the underlying individual `c02` IRENA-functions, no special code is required in **`nagpolysolve`** to provide regularity here.

The **`nagpolysolve`** function also includes code to handle degenerate cases and is written so that special REDUCE constants, such as `PI`, occurring in coefficients are converted to numeric form. This, rather unnecessarily, saves and restores various REDUCE switch settings – which could have been done more simply by having local copies of the switch variables, declared as `fluid`.

Like the functions described in the previous section, the top level function here was written in algebraic mode; however, the lower level function to convert the coefficients required rather more complex manipulations and so, like the remainder of IRENA, was written in symbolic mode.

The full version of the code for this function is given in appendix G. In the released version of IRENA-1, since the special routines for the quadratic case are not present in the Foundation Library, the code for handling quadratics as a special case was removed.

As has already been seen, the regularity of the standard IRENA-functions is a considerable advantage in writing higher level functions such as **`nagpolysolve`** and should encourage the production of other such high level interfaces.

The most complicated aspect of this exercise was the transformation of the representation of polynomials, which, nevertheless, only required about a hundred lines of code (and would, no doubt, have needed less if coded by a more experienced RLISP programmer). Although this was also quite encouraging for the prospects of producing higher level interfaces for other sets of related routines, no further work was devoted to that end as part of the present project, due to the very heavy effort required to produce the basic IRENA interfaces for other NAG routines.

The nonlinearity found in section 8.3, in the relationship between the number of parameters of a routine being processed and the amount of code to provide its interface, is suggestive of a possible more general nonlinearity between the size of a body of software to be interfaced and the interface code required and, at the very least, suggests caution in attempting to provide higher level interfaces more widely. A sensible approach would be to treat the problem incrementally, choosing individual problem areas for interfacing and gradually increasing the size of the sets of routines for which unified interfaces are developed.

One disadvantage of the approach adopted here is that the code developed is only applicable to REDUCE (and could possibly require significant maintenance as later releases of REDUCE occur). A different approach, using Fortran 90 jackets to provide interfaces, is discussed in section 15.3. As Fortran 90 is a stable, generally applicable language, this approach should overcome the problems associated with using a package specific language such as RLISP. Although the discussion in section 15.3 will concentrate on interfaces for individual routines, the same advantages would apply to higher level interfaces: this approach may well be adopted in future to provide higher level interfaces for the Axiom-NAG link.

Chapter 14

Documentation of IRENA-1

One of the present author's principal dissatisfactions with the earlier Naglink system, described in section 1.2, was the low utility of the documentation for users not already familiar with the underlying Macsyma system and, especially, with the routines of the NAG Library which it utilised. Consequently, the design objectives formulated for IRENA included making the software self-documenting, where possible, and otherwise making the documentation as self-contained as possible. Some of the design decisions taken for IRENA itself reflect these concerns – as noted earlier:

- the system explains how to respond to its prompts and to access its results, at appropriate points in its use;
- it produces error messages automatically, rather than requiring users to interpret error codes by consulting the documentation;
- the help system relates these error messages to an IRENA context;
- "self-documenting" naming conventions were adopted.

However, separate documentation was still required, to describe general points relating to the use of the system, to explain the functionality of the various IRENA-functions, to document the various parameters in more detail, in particular those parameters for which defaults were

provided, and to present examples of the usage of each IRENA-function. In fact, about 10% of the total effort expended on IRENA was devoted to producing its documentation.[1]

Since IRENA was developed as a REDUCE "package", it was natural to adopt the same LaTeX based approach to documentation as is used in the rest of REDUCE; this is straightforward and appropriate to a command-driven system such as REDUCE (or IRENA), as well as being compatible with the basic documentation style of other NAG products.

The documentation for IRENA-1 consists of two main components: a User Guide and individual description documents for each IRENA-function. Except where otherwise noted, the documentation was developed by the present author.

14.1 The User Guide

This publication [33] was designed to provide sufficient background information to allow users to make effective use of IRENA; it consists of (iv + 70) pages, whose LaTeX source occupies 139378 bytes. In addition to a Preface and Bibliography, it contains fifteen chapters and seven appendices; the content of each of these is described briefly here.

Production of the User Guide occupied several weeks.

14.1.1 The chapters

Introduction

This chapter describes how IRENA simplifies NAG routine use and provides basic information essential for its use or not covered elsewhere:

- how IRENA is accessed through REDUCE,
- how to recover from failures,
- IRENA-function nomenclature *and*
- (for advanced REDUCE users) how to incorporate IRENA-functions in compiled REDUCE procedures.

[1] In a personal communication, R. W. Brankin, Deputy Divisional Manager of NAG's Numerical Libraries Division, estimates that the corresponding figure for new Fortran Library material is in the range 5% to 20%, so, in this respect, IRENA seems reasonably comparable to the NAG Library.

Simple IRENA usage

This chapter explains how to call IRENA-functions using the keyline, how to access the results and how to respond to prompts for parameters omitted from the keyline, when **envsearch** is on.

Vectors and matrices

This chapter explains how vectors and matrices may be represented using a special collection of functions in IRENA, pointing to appendix A for full details of these; it shows how structures defined using this representation may be converted to REDUCE matrices, whilst pointing out that this is not normally necessary; it also points out that, in general, either representation may be used for IRENA input parameters but that output vectors and matrices are represented as REDUCE matrices.

Defaults

This chapter introduces the IRENA defaults system, distinguishing between defaults for housekeeping and control parameters, and explains that, of these, only the control parameters are documented (as "Optional Parameters") in the function description documents. It also introduces the second level default variables ***userabserr***, ***userrelerr***, ***usermixerr*** and ***userinputerr*** and explains that these may be reset during an interactive session or in the startup file.

ENVSEARCH and PROMPTVAL

This chapter explains the effect of the **envsearch** and **promptval** switches – respectively to take the values of unspecified parameters from the REDUCE environment and to prompt for these if default are not provided – and explains how they may be reset. It also introduces the **promptall** switch – which extends **promptval**'s effect to non-housekeeping parameters with defaults – and mentions that **promptall** and the associated identification of housekeeping parameters are still under development.

Input jazzing

This chapter briefly describes the philosophy behind jazzing and describes the main types of input jazzing under the headings

- Aliases,
- Keywords,
- Trimmed arrays *and*
- "Rectangular" regions.

It also mentions that some additional, specialised forms give rise to the data types described in appendix F.

Output jazzing

This chapter explains how the jazzing philosophy relates to output parameters, explains that the IRENA output parameter names are obligatory, introduces the concepts of long and short forms of these names and mentions the user alias facility.

Data input from files

This chapter explains how data stored in files may be read into an IRENA-function.

Argument subprograms (ASPs)

This chapter points out that, in many cases, users need not be aware of the use of ASPs by IRENA-functions – but that an important case where the use of an ASP affects the IRENA user interface is user-defined functions and families of functions, which may be supplied as REDUCE expressions or IRENA `fsets`. It describes the replacement of the `OUTPUT` subroutine, which is present in many NAG routines to print out values at intermediate points (for example, in the solution of differential equations) by an IRENA input vector **`output points`** and an output matrix `solution_at_output_points`. The chapter also mentions that users have the option of writing the code for ASPs in Fortran (or modifying the Fortran generated for these by IRENA) and specifying this in the IRENA call, as well as describing the use of **`fortinclude`**.

Setup files

This chapter describes how the REDUCE setup file may be used as a convenient means of setting IRENA switches and second level defaults and of specifying the location of various system directories.

Personal alias files

This chapter describes how the user can specify new input and output names for parameters and change the location of the directory which contains the alias files.

Personal defaults files

This chapter describes the structure of defaults files and how to specify and cancel defaults; it points out that a `housekeeping` entry in a defaults file (like any other entry) overrides the system's setting, explains how to change the location of the user's defaults directory and mentions the formal defaults syntax in appendix E.

HELP in IRENA

This chapter explains why a help system is necessary, to aid in the interpretation of some error messages, and describes the four functions

- `jazzing`,
- `default`,
- `details` *and*
- `explain`.

Accessing the Fortran

This chapter explains how to retain the Fortran code generated by IRENA, how to arrange for this to be produced in the form of a self-contained program and how to change the directory in which it is stored. It also points out that the system can be used to cross-generate Fortran programs to be run on different computers and mentions how, for some machines, single precision

Fortran variables may be appropriate and can be produced by turning off the REDUCE switch `double`.

Source code optimisation

This chapter describes the effect of the GENTRAN switch **gentranopt**, which performs source code optimisation in the generated Fortran (as described in section 3.2.1), and discusses some advantages and disadvantages of this, in particular the occasional generation of illegal Fortran.

14.1.2 The appendices

Most of the appendices are adequately described by their titles:

A Vector and matrix facilities in IRENA

B Mnemonically named functions based on NAG routines

C NAG constants available in defaults files

D Functions available in defaults files

E Formal defaults syntax

F IRENA function descriptions

G Keyline switches

Appendix F

This describes how the individual function descriptions are organised, provides a glossary of input data types, lists the IRENA-functions in IRENA-1 and concludes with a ficticious function description, to illustrate all of the components of such descriptions in brief and without the need to understand the details of a particular function.

Appendix G

This describes an additional feature, added by Dewar shortly before the release of IRENA-1, whereby IRENA switches may be reset in the keyline for the duration of an IRENA-function call.

14.2 Function descriptions

Following the style of NAG Library documentation, a separate *function description document* was provided for each IRENA-function, with the exception of `a00aaf`, which only serves to identify the version of the NAG Library being used and which is documented in the User Guide.

The information needed to document the functions is, in principle, present in the NAG Library documentation and the various IRENA system files, such as the jazz and defaults files and ASP sources. However, the conversion from NAG to IRENA parameterisations of the user interfaces means that much of the NAG documentation cannot be used directly, as there is far from a one-to-one mapping between parameters: cross-references to other parameters in a parameter description are particularly troublesome to resolve automatically. Since there are many IRENA ASP types and jazz-functions which are of limited applicability, the effort likely to be involved in writing a nearly automatic document convertor would have been considerable. Moreover, since a rationalisation of jazzing would be a priority in any future revision of IRENA, such a convertor would have little chance of reuse. For these reasons, a mixed automatic and manual approach to documentation was adopted, with "skeleton" function documents being produced automatically, as previously mentioned in chapter 11, but with considerable manual modification and extension of these being required to resolve difficult points and to deal with those areas which were not considered to be worth automating.

The automatic skeleton document generator was written by Dewar and G. Nolan, a Teaching Company Associate at NAG. It took as its starting point the NAG Concise Reference manual [22], to provide descriptions of the purpose of each routine and of the NAG parameters. Information from the routine's specfile was used to eliminate workspace and dummy parameters, to identify the types of ASPs, to determine whether the NAG routine was a function (so needing an extra return parameter in its IRENA representation) and to flag other NAG parameters as input, output or both. The mapping from NAG to IRENA parameters was obtained from the jazz file and an attempt was made to analyse whether or not input parameters were "essential" or "optional", using information from the defaults file, which also provided default values. A facility was provided to mark parameters in the defaults file as "essential" or "optional" but this was little used – especially as, at this level, it referred to NAG rather than IRENA parameters. Functionality was included to read IRENA parameter descriptions embedded in the local version of the jazz file[2] – this allowed recurrent parameter descriptions to be provided through a set

[2] These descriptions were stripped out in the released version of IRENA.

of LaTeX macros developed by the present author. A similar system allowed descriptions of ASPs, in IRENA terms, to be read from the REDUCE source code definitions of the ASP types. Finally, the IRENA test example was included, preceded by the standard description of the corresponding NAG example program, on which it was based.

Each function document consisted of five sections

1. Purpose
2. Essential Input Parameters
3. Optional Input Parameters
4. Output Parameters
5. Example

Although the skeleton documents provided a useful starting point, little in them could actually be relied on as IRENA documentation, especially as the NAG documentation was, at that time, written in the typesetting language TSSD [16] and the NAG TSSD to LaTeX convertor, which was then under development, could not always handle mathematical typesetting correctly.

Probably the item requiring least modification was the Purpose section, although, even here, occasional mentions of the parameterisation used had to be amended. The text here was also occasionally revised for greater clarity, by including additional information from the Description section of the NAG routine document (which is considerably more comprehensive than the Purpose section of either that document or the Concise Reference entry), to correct mathematical typesetting and for greater consistency across documents. (The revised Purpose sections could, in future, be used for other products where more self-contained descriptions of incorporated routines are required; in the context of the NAG Library documentation, the present form may, perhaps, be preferred, since the additional information incorporated for IRENA is already present elsewhere in the same document. The mathematical typesetting produced by the TSSD convertor has since been corrected.)

The example description was, similarly, fairly reliable although the need to correct mathematical material was greater here. Names of NAG routines mentioned in the description had to be replaced (not necessarily on a one-to-one basis) by the IRENA-functions used in the IRENA test example. The example itself did not, generally, require modification, apart from occasional attention to its layout.

Due to the radical reparameterisation carried out in IRENA, the skeleton documents provided little more than a basic structure for the parameter description sections: except where the IRENA parameter corresponded to one of the NAG parameters, the text describing the parameters was often inapplicable in the IRENA context and was largely replaced. Even when there was an exact match between a NAG and an IRENA parameter, some revision could be required when other parameters were mentioned in the description.

The analysis of input parameters into "Essential" and "Optional" was, at best, tentative and did not take account of much of the reparameterisation.

The facility to provide general descriptions of ASPs was little used since, for the majority of these, the number of instances was very small and it proved simpler to deal with these as they arose in the document, especially as it was then possible to describe the function of the ASP in context. The skeleton document's handling of ASPs was, however, useful in stripping out mention of those ASPs for which no user input was required.

Development of the function description documents occupied the present author for several months – on average, each probably required nearly a day's work to complete (although a few complicated routines contributed disproportionately to this average).

14.3 Other documentation

In common with other NAG products, IRENA was provided with an Installers' Note [28], explaining how to mount the system.

14.4 Comparison with NAG documentation

14.4.1 The User Guide

The IRENA User Guide does not have a close equivalent in terms of NAG Library documentation – after all, most NAG Library users are already familiar with Fortran, whereas new IRENA users would certainly not be familiar with the use of IRENA and might well not even be familiar with REDUCE.

In terms of the Foundation Library, on which it is based, the nearest equivalent to the User Guide is perhaps the Foreword and Introduction, which together occupy 82 pages, slightly more than the User Guide. However, discounting the "List of Routines" and "Keywords in Context" section, this falls to 18 pages, considerably less than the User Guide.

A closer parallel may be found in the more recent NAG Fortran 90 Library manual [29], which, like IRENA, addresses the problems of a possibly unfamiliar language and of a number of conventions peculiar to itself. The first two sections of this, the Essential Introduction and Tutorial, which are reasonably comparable to the IRENA User Guide, together occupy 58 pages, quite close to the length of the User Guide (the body of which occupies 68 pages, with a similar print area).

14.4.2 The function documents

A feature of the NAG Library which tends to intimidate users is the extent of the documentation[3]. Because of the rationalised user interfaces and the consequently simplified examples, IRENA function documents are considerably reduced in size, compared to their NAG counterparts – a subjective estimate is "less than half the size".

This reduction is due to a number of factors:

- the Specification section of the NAG routine documents, which shows the Fortran declarations of the routine and its variables, is unnecessary for IRENA;
- the Error Indicators and Warnings section is also unnecessary, since the interpretation of these is built into IRENA;
- there are usually fewer parameters and, as their names are descriptive, they require less documentation;
- the IRENA example is usually considerably shorter than its NAG equivalent.

[3] For example, the Mark 16 manual, which describes 1134 routines, is supplied in 12 A4 manuals, each of about 500 pages, which in total occupy some 70 cm of shelf space.

However, the NAG documents also include other sections which have no counterpart in the IRENA function documents, namely

- Description
- References
- Accuracy *and*
- Further Comments

(although some of the information in these is incorporated in the IRENA interface, in the jazzing process). IRENA users requiring these details are, in fact, referred to the NAG documentation.

As a result of the relative reduction in the size of the function description documents, the majority (108 of 159) of these occupy less than two pages of A4 paper – it would be interesting to contrast this with the NAG "routine documents". In an attempt to quantify this reduction, at least approximately, the number of pages of documentation, excluding the sections mentioned above as having no IRENA counterpart, were measured for a small paged sample of NAG routines[4] and compared to the corresponding IRENA documents. The material in the NAG documents was found to occupy 29.9 pages, that in the IRENA documents 17.3 pages, giving a raw figure of just under 60% for the ratio of the sizes of IRENA and NAG documents.

Both forms of documentation use the same font size. However, the IRENA documents were originally designed for photo-reduction before printing and consequently have wider margins than their NAG equivalents, so that the height of the IRENA print area is about 90% of NAG's and the width about 87%, with the total available print area being about 79%. Adjusting the ratio of the document sizes by this factor gives a revised figure of 46%. There are a number of factors biasing these figures: for example, many lines in both sets of documentation are shorter than the width of the page, so the comparative line lengths are less influential than might be assumed; on the other hand, there appears to be more white space in the IRENA documents, especially in cases where vectors or a succession of results are printed in the example, since here the NAG version generally prints a table but the IRENA results occur on individual lines, with additional spacing inserted by REDUCE. Overall, the subjective estimate of a more than 50% reduction in the volume of documentation appears plausible.

[4] C06EAF, D01BBF, E01SEF, E04DGF, F02ADF, F04MAF, S14BAF and S18DCF. For E04DGF the sections of the NAG documentation concerned with optional parameters were excluded, as these were not incorporated in IRENA-1.

In an attempt to obtain a more accurate measure, the LaTeX sources of the two forms were compared. This allowed the sizes of the "descriptive" sections of the documentation (that is, with the example program removed) and the examples to be compared separately. The results are shown in tables 14.1 and 14.2.

Routine	Documentation (excluding example) LaTeX source		Example run				
			NAG Library			IRENA	
	A: NAG Library	B: IRENA	C: Program	D: Data	E: Results	F: Full example	G: Input (data-free)
C06EAF	3071	1185	2058	104	570	1088	266
D01BBF	5200	2478	962	0	289	752	195
E01SEF	5758	2771	2385	1055	573	2870	386
E04DGF	14688	1476	1879	108	2093	2743	196
F02ADF	3991	1199	1157	236	85	858	185
F04MAF	10431	4906	3269	0	468	2527	1561
S14BAF	3110	1302	857	88	332	836	402
S18DCF	5819	1605	1172	173	526	941	385
Total	52068	16922	13739	1764	4936	12615	3576

Table 14.1: Sizes in bytes of NAG and IRENA documents

Routine	Relative sizes (percentages) – see table 14.1		
	B / A	F / (C + D + E)	G / C
C06EAF	39	40	13
D01BBF	48	60	20
E01SEF	48	72	16
E04DGF	10	67	10
F02ADF	30	58	16
F04MAF	47	68	48
S14BAF	42	65	47
S18DCF	28	50	33
Overall	32	62	26

Table 14.2: Size ratios of IRENA and NAG documents

The descriptive text

In the NAG versions, the sections which have no IRENA equivalent were excised; in the IRENA versions, text (but not layout) macros developed for IRENA were expanded: as a result, the two forms should be fairly comparable, since the NAG documentation also uses tailored layout macros.

As can be seen from the first column of table 14.2, compared to the NAG material, the IRENA LaTeX source is reduced by between 52% (**E01SEF**) and 90% (**E04DGF**) with an overall figure of 68%. The exceptional reduction in the case of **E04DGF** is largely due to the extensive descriptions of error indicators in the NAG documentation; this and the highly skewed distribution of the ratios suggests that the median value (60%) may be a better indicator than the mean for these results. It is worthy of note that the reduction is, in every case, greater than 50%.

Examples

In the case of the examples, IRENA differs from the NAG Library in having its data embedded in the example, rather than in a separate file; the IRENA example runs, as documented, also display their results. For these reasons, the size of each IRENA example was compared to the total size of the NAG example program, data and results.

As shown in the second column of table 14.2, in this case the range of reductions in size is from 28% to 60% but the values were much more uniformly distributed, giving an overall mean of 38% and a median of 37%.

As much of the volume of the examples is due to the data and results, it is interesting to compare the sizes of the code alone, although this is not strictly a matter of documentation. Copies of the IRENA example inputs were prepared in which, for those cases where a NAG data file existed, the data values were stripped out. The final column of table 14.2 compares the sizes of these with the NAG programs, showing reductions ranging from 52% to 90%, with a mean of 74%. Once more the distribution is highly skewed – perhaps even bimodal[5] – with a median of 82%. Here again, the reduction is, in every case, more than 50%.

[5] In the S chapter examples, the NAG program takes the form of a read-call-write loop, whereas, for clarity, a separate IRENA-function call was defined for each data point. The **F04MAF** situation resembles this in that, for greater transparency in the IRENA example, the data matrix was generated directly from the problem description, with each non-zero element specified separately; in the NAG example, identical values were assigned in loops. (To have adopted the NAG approach, in IRENA, would have required the use of an intermediate structure to hold the data values and locations, prior to the definition of a sparse matrix using these.)

Part III

Conclusions and recommendations

Chapter 15

Considerations for the design of future IRENA-like systems

15.1 Precedence of non-housekeeping defaults

15.1.1 Parameter evaluation strategy in IRENA-1

In IRENA, if both `envsearch` and `promptval` are `on`, the value of a parameter is determined according to the precedence:

1. keyline specification,
2. user defaults file specification,
3. system defaults file specification,
4. REDUCE value (global, or a loop control variable),
5. response to an IRENA prompt.

Whilst this is certainly the correct order for housekeeping parameters (for which entries 4 and 5 should in any case be irrelevant), the situation for other parameters is more problematical.

15.1.2 Control parameters

The author envisages that most users will be happy either to use the system defaults for control parameters or to specify permanent defaults of their own. Occasionally, however, some users may wish to experiment with varying control parameters to explore the behaviour of a NAG routine or the underlying algorithm. Whilst this can be done by varying a keyline value in a loop, doing so adds a level of indirection to the code, which the user may prefer to avoid. Certainly, the meaning of

```
for each error_control in {0.01, 0.001, 0.0001} do
<< d02bbf() $ ... >>;
```

is more immediate than that of

```
for each ec in {0.01, 0.001, 0.0001} do
<< d02bbf(error_control=ec) $ ... >>;
```

Admittedly, this is a matter of personal preference; the point, however, is that users should not be constrained to use the second form.

15.1.3 Data parameters

Normally, there is no need to provide defaults for data parameters; the commonest exception is that, where a routine allows the weighting of data values, there is an obvious default setting of equal weighting; other examples include the use of equispaced grids in interpolation etc.

Dewar, in a personal communication, argues that such parameters should be classified as control; the present author feels that, at most, they could be said to be data parameters with a particular value serving a control function. Although the distinction has no practical effect in IRENA-1, if some form of extended user control, as described in section 15.1.4, is introduced in a later system, it would seem an unfortunate side effect if, say, a user electing not to be prompted for defaulted control parameters thereby inhibited prompting for some parameters with a possible data rôle.

In any case, it is unlikely that a user working in **envsearch** mode would want a default value to override a data value in the REDUCE environment. This was prevented in IRENA-1 by making such defaults conditional on the absence of any REDUCE object of the appropriate type with a name corresponding to the parameter in question. This is not altogether satisfactory, since with

several possible aliases representing the parameter the defaults entry can become quite lengthy. Further, there is a danger of redefining an alias (in the jazz file) and omitting to adjust the conditional default to take account of this. (In particular, this could apply in the case where a user chooses to rename a parameter.)

15.1.4 Recommendation for future enhancements

The switch `promptall` (described in Section 9.2.6) was added to IRENA to allow prompting for defaulted, non-housekeeping parameters. To accomplish this, it was necessary to flag which parameters were to be considered housekeeping. An obvious extension to this would be to allow parameters to be flagged as control or data.

Once this was done, it would be possible to replace the **`envsearch`**, **`promptval`** and **`promptall`** switches with a unified system, allowing users to specify the precedences for evaluating the three categories of parameters separately and to redefine to which category any parameter should be assigned. The author's own preference, for both control and data parameters, would be for REDUCE values to take precedence over defaults and this is suggested as a possible default setting.

Should such a system be introduced, there is clearly a need to separate the control and data rôles of some parameters. If this is done at an IRENA, rather than a Fortran, level, then the ability to classify parameters ought to apply to the resulting IRENA parameters, rather than to the NAG parameters which they replace. (In general, users should be able to remain unaware of the underlying NAG parameters of a routine and should only be concerned with the parameters of the IRENA-function.) However, a strong case exists for such interface redefinition to be implemented in Fortran jackets; this will be discussed in section 15.3.

It can be seen from the above discussion that the set of controls, used in IRENA-1 to handle the precedence of parameters values from different sources, evolved into a form which lacked a cohesive design. This is characteristic of the way in which the design of experimental software systems can be subverted by considerations which were not initially apparent. *With hindsight*, the original design of this area of IRENA should probably have been discarded and replaced by a more unified approach earlier in the project; although system developers are naturally aware of their investment in any system and so may be reluctant to change the *status quo*, failure to make such a change when necessary will result in an even greater investment eventually being abandoned.

15.2 Are defaults distinct from jazzing?

In IRENA-1 there are separate mechanisms for "jazzing" and setting defaults – at first sight, this seems eminently sensible, since jazzing is used to redefine the user interface to a function which appears to be quite a different activity to defining a default value for a parameter. However, closer inspection reveals that both are aspects of the same activity – defining NAG parameter values in terms of a different (not necessarily disjoint) set of parameters. It might be thought that the default setting activity is distinguished by the subset of parameters required to define a default value being empty – however, this is by no means always the case as may be seen from the default for **LWORK** in the routine **E04UCF**:

```
LWORK : if NCLIN = 0 and NCNLN = 0 then
           20*N
        else if NCNLN = 0 then
           2*N*N + 20*N + 11*NCLIN
        else
           2*N*N + N*(NCLIN + 2*NCNLN + 20) + 11*NCLIN + 21*NCNLN
```

in which the values of three other NAG parameters are involved in the calculation. Thus, simple numerical defaults such as

```
JOB : 1
```

(for **C06EKF**) are simply a limiting case (and rather the exception).

It might be argued that defaults only involve the values of scalar parameters – this is true of IRENA-1 but, in fact, represents a deficiency in that system: for instance, in the (now superseded) optimisation routine **E04JBF**, the NAG Library Manual's suggested value for **ETA** is 0.5, except when **N** is 1 or when "for all except one of the variables the lower and upper bounds are equal", in which case it is 0.0. The calculation of this default (if it were possible in IRENA-1) would involve the values stored in the arrays **BL** and **BU**, which contain the bounds. IRENA-1 does, however, admit tests for the *existence* of non-scalar parameters in defaults so, even there, this distinction is blurred.

A perhaps more valid distinction is that jazzing commands tend to describe "structural" relationships between NAG and IRENA parameters, in the sense of mappings between different data types, whereas default specifications describe arithmetic relationships. However, some

arithmetic relationships are handled by the input jazzing command **newscalar** and a general arithmetic ability is provided by the output jazzing command **calculate**. Thus, there appears to be no intrinsic reason that input jazzing should not also provide a general facility.

A further argument for considering jazzing and defaults setting to be intimately connected is that, in IRENA-1, it is often necessary to use an interaction of the two systems to obtain a desired effect. Several examples of this were encountered in chapters 10 and 11.

- **MATV** in **F02BJF** is jazzed so that it may be specified either by means of keywords or in response to a "Yes or no" question. This requires an IRENA scalar **matv-key** to be defined in the jazz file; defaults for this scalar and for **MATV** itself, in terms of the scalar, are also required.

- **W** in **E04GCF** is restructured on output. An intermediate variable **ns** is introduced in the jazz file, given a value in terms of the NAG input parameters **M** and **N** in the defaults file and finally used in the jazz file to help define the dimensions of the restructured components of **W**.

- **XCAP** in **E02AEF** is reparameterised in terms of more natural quantities **x**, **xmin** and **xmax**, easily obtainable from the output of **E02ADF**. The reparameterisation, being a scalar calculation, is carried out in the defaults file but refined (by converting **xmin** and **xmax** into a "rectangle") in the jazz file.

- **D01XXX**, **A** and **B** in **D01BBF** are completely reparameterised in terms of the rectangle **range**, the IRENA scalars **parameter_a** and **parameter_b** and the keywords **gauss_laguerre** and **gauss_rational**. This parameterisation is defined in the jazz file but the redefinition of the NAG parameters in terms of the scalars underlying these structures is carried out in the defaults file.

- **ACC**, **NOITS** and **IFAIL** in **F04MAF** all represent combinations of logically distinct items. The separate items which they represent are introduced in the jazz file but mapped onto the NAG parameters in the defaults file; additionally, keywords corresponding to the meanings of the components of **IFAIL** are defined in the jazz file.

- **N** in **E04MBF** can only be given a default because the NAG parameters **BL** and **BU** are split into their logical components in the jazz file.

The case of **MATV** in **F02BJF**, mentioned above, exemplifies a frequently occurring situation in IRENA, in which we wish to handle a NAG parameter functioning as a switch by issuing a

prompt in the form of a question to which the answer is **Y** or **N**. Further, as in this case, we may wish to allow the user to include keywords in the IRENA-function call, equivalent to these **Y/N** responses and we may require one value to be the default. To allow both the keyword and prompting approaches, we must define an IRENA "scalar" and associate with two (arbitrary but distinct) numerical values of this scalar the desired keywords, define the prompt, indicating that it should request **Y/N** responses, and associate **Y** and **N** with the appropriate NAG values. (So far, these are all jazzing operations.) Finally, we must give a "default" value to the NAG parameter which associates the correct values with those chosen to represent the scalar via keywords and, if a genuine default is to be established, set an appropriate default value for the scalar to accomplish this.

As a further example, the relevant section of **CO2AFF**'s jazz file, dealing with the parameter **SCALE**, is

```
{prompt!-alias} SCALE : scale! the! polynomial

{set!-type} SCALE : "(Y or N)"

{local} SCALE [!.true!.] : y

{local} SCALE [!.false!.] : n

{scalar} scale!-key

{qkeyword} scale!-key [1,1,2,2] : scaled, s, unscaled, u
```

and the entries in the defaults file are

```
scale!-key : 1

SCALE : if scale!-key = 1 then TRUE else if scale!-key = 2 then FALSE
```

If IRENA is used with the switch **promptall** on and **SCALE** has not been set by the user, the resulting prompt is

```
(Y or N) scale the polynomial?
```

This type of situation occurs sufficiently frequently[1] – and the interplay of jazzing and defaults is sufficiently confusing – that a small (270 line) C program was written to allow the more common cases to be specified interactively. This interrogates the programmer to determine which features are required in a particular case and then generates appropriate fragments of jazz and defaults files.

In conclusion, there appears to be no clear dividing line between the activities of setting default values and jazzing – and the whole task of redefining routine interfaces would be simplified if a uniform system for setting "derived parameter values" were introduced. One possible such system is described in section 15.3. Similarly, the facilities currently available to users, to redefine input and output names and reset default values, could be duplicated in a unified system in which three commands, `input`, `output` and `default`, were used in a single file, although the resulting subsystem might feel rather more natural if a fourth, `canceldefault`, were added, to replace the use of `canceldefault` as a *value* in default setting. However, a new, combined system, suggested in section 15.4, is sufficiently straightforward that allowing users access to its full functionality might be considered practicable.

15.2.1 Impact on specfiles

A further question which naturally arises is whether the remaining point of human intervention in setting up an IRENA-function, namely, revision of the automatically generated specfile, should also be incorporated in a unified system.

There are several advantages in carrying out this further integration:

- all programming activities in the definition of IRENA-functions would then be carried out in one location, which is simpler and should reduce opportunities for the introduction of errors;
- the introduction of additional "quasi-NAG" parameters becomes much more natural, since these can be specified analogously to the actual NAG parameters; this would eliminate a fertile source of IRENA system errors, attributable to attempting to introduce such parameters in the jazz file, as part of the `scalar` and `vector` mechanism;
- similarly, the elimination of an unnecessary output rôle for some input/output parameters could be accomplished more naturally than with the `*noname*` facility; as discussed in

[1] The approach described here is used for 28 of the 160 routines in the IRENA-1 subset and, quite often, for more than one parameter in a routine.

section 11.1.1, changing the specfile to accomplish this was avoided, as the specfile was regarded as fixed and allowing rare changes to it could have resulted in maintenance problems if these were overlooked at a later date;

- necessary changes to input array dimensions – for example, when these are "assumed-size" in the NAG routine, or differ there from those of the underlying object – may also then be made to the automatically generated specfile settings, with less hesitation; this has been avoided previously, for the reasons just given;
- modification to the NAG error messages, to reflect the IRENA parameterisation, is more likely to be carried out if these messages are visible in the same file in which the reparameterisation is defined; making such modifications would be facilitated by not attempting to maintain the NAG interface as an alternative in the IRENA-function.

As in the case of parameter precedence in section 15.1.4, this discussion illustrates the need at times to abandon an early design and replace it with a more unified version. In this case, activities which were initially conceived of as having very distinct rôles – setting defaults, jazzing and to, some extent, creating specfiles – prove to be instances of a more general function, that of reparameterising a user interface. Recognition of such unities can bring about considerable simplification in the design of a system.

15.3 Generalising interface design

To generalise interface design for compatibility with a variety of host systems, it becomes essential to avoid, as far as possible, the system dependence described in section 8.2. This implies that, whilst it may be necessary to produce some code in the host system's language to enable connection to the interface, this should be kept to the minimum practicable. The remainder of the interface should be coded in a "neutral" language. What this language should be is open to debate: given the objective of providing links from host systems to NAG numerical software, which is written in Fortran, there is a strong a priori case for using Fortran (and, in particular, Fortran 90) for at least the "NAG end" of the interface.

To some extent, this occurred in IRENA. As described more fully in section 12.1, Fortran jackets were provided for a number of routines, to enable common combinations of NAG routines or multiple calls of the same routine to be handled in a call to a single IRENA-function. When the desired transformations could be carried out in Fortran, this technique proved very efficient, in

terms of the coding effort required. What is not immediately obvious is whether Fortran would be a suitable tool for building most of the interface.

A complete interface between a host system and a numerical library must handle a number of distinct processes. One possible order for these would be:

1. elicit data;
2. select appropriate routine(s);
3. transform data[2] to routine's format;
4. transmit to Library machine;
5. handle multiple calls;
6. transmit output data;
7. transform output data;
8. display results.

(After process 2 there may need to be a further interaction with the user, to determine essential, routine specific control parameters. This will be dealt with in section 15.3.2; it does not influence the present discussion.)

Processes 1 and 8 are obviously dependent on the host system but none of the others needs be.

Processes 3 and 4 and, respectively, 6 and 7, could be reversed in order - in other words, the data transformations could be carried out on either the host machine or the Library machine. The better choice here depends on the chosen language of implementation – as suggested above, this should probably not be the language of the host system, for portability considerations, in which case it is likely to be either Fortran or a general systems language such as C. Fortran would certainly be available on the Library machine, C would probably be present on both, other languages might be more problematical.

If the data transformation modules were provided in Fortran or a Fortran callable form, they could be distributed as an adjunct to the NAG Library and might find more general use. A sensible first approach, then, seems to be to investigate the extent to which Fortran compatible data transformation could be provided.

[2]Note that the data being transformed may include functions and other items required by ASPs; the impact of this on the recommended solution is examined in section 15.3.3.

15.3.1 Analysis of IRENA jazz command usage

A summary of the use of input jazz commands in IRENA is provided in tables 15.1 and 15.2 with the corresponding output jazzing analysis in tables 15.3 and 15.4. (See appendix D

Command	FL	FL+	Command	FL	FL+
prompt-alias	402	612	sparsecolumn	3	3
key-alias	272	413	sparserow	3	3
local	105	137	sparsevalues	3	3
set-type	80	88	column-mat	2	5
scalar	75	95	sbandlengths	2	3
fort-dims	67	74	sbandvalues	2	3
qkeyword	66	72	cmat2ivec	2	2
keyword	55	70	cmat2rvec	2	2
rectangle	43	80	diagonal	2	2
silent-alias	30	55	rowmat2vec	2	2
concatenate	19	28	trim-matrix	2	2
template	19	23	raggedlengths-1	1	2
phased-prompt	13	13	raggedvalues	1	2
vector	10	13	maxraggedlengths	1	1
gridfirst	10	10	row-mat	1	1
gridsecond	10	10	sumraggedlengths	1	1
ragged-in	9	9	trim-vector	1	1
tuples1	8	8	unpack	0	3
tuples2	8	8	hi-d-dims	0	2
complex-in	6	14	hi-d-im-vals	0	2
mat2vec	6	10	hi-d-re-vals	0	2
tuples3	6	6	exteriorpolygon *	0	1
fill-knots	4	4	polygonx *	0	1
newscalar	4	4	polygony *	0	1
rect2scalar	4	4			

Key: FL Frequency of usage in Foundation Library jazz files
FL+ Frequency of usage in all jazz files
* not yet implemented

Table 15.1: Frequencies of input jazz commands

for brief descriptions of the functionality of the jazz commands used in IRENA-1.) Totals are given for the frequencies of occurrence of commands in IRENA-1, as released, and for all NAG routines processed at the time of the IRENA-1 release. This second category includes about 100 further routines from the NAG Mark 15 Fortran Library. These routines were not subject to the final harmonisation process which the IRENA-1 contents underwent, so the frequencies may not reflect "mature" usage. They are included because they represent a wider range of structures than occurs in IRENA-1. Unless otherwise indicated, the rest of this discussion will refer to the IRENA-1 jazz files.

Category	Number of commands		Number of instances	
	FL	FL+	FL	FL+
Renaming parameters	3	3	704	1080
Naming special values	3	3	226	279
One IRENA ⇒ many NAG	21	28	127	188
Controlling prompting	2	2	93	101
Local objects	2	2	85	108
Reset Fortran dimensions	1	1	67	74
IRENA ⇒ part NAG	3	3	22	34
Alternative routine	1	1	19	23
More natural object	4	4	16	20
Accessing substructure	2	2	3	3
Key: FL Usage in Foundation Library jazz files FL+ Usage in all jazz files				

Table 15.2: General input jazzing classification (principal uses)

From table 15.2, it can be seen that the majority of instances of input jazzing are concerned with simply renaming variables and that this rises to more than two thirds when the naming of special values is included. These are not operations which could be easily carried out in Fortran 77, with its purely positional parameter list and fixed array sizes; however, in Fortran 90 they seem considerably more feasible, given the introduction of optional arguments, the intrinsic function `PRESENT` and automatic arrays. We should note, however, that the commands **`prompt-alias`** and **`set-type`** are involved in the data elicitation process; **`prompt-alias`** could also be involved in defining the name of an alternative Fortran 90 parameter but, although Fortran 90 allows names of up to 31 characters, some IRENA prompts are considerably longer than this. As some renaming must, therefore, be carried out before a Fortran program is invoked, the most natural solution is to treat renaming operations purely in the data elicitation module. With prompting control included, data elicitation then accounts for 80% of input jazzing instances (but only 19% of the types of commands).

Disregarding those jazz commands which are essentially type declarations for local objects, the next most common input jazz command is **`fort-dims`**, accounting for 26% of the remaining instances. This command is used to ensure that a NAG array has the correct dimensions, for instance, when the NAG routine uses an assumed-size array whose dimensions cannot be determined directly from those of input objects. Usually, the dimensions are maxima or minima of the values of objects' actual dimensions and arithmetic functions of other NAG parameters. Coding this in Fortran 90 would present no difficulty.

The next most common command is `rectangle`, which effectively translates a list of pairs, representing bounds in one or more dimensions, into two NAG scalars or one-dimensional arrays. This is just one instance of the type of input jazzing, categorised as mapping one IRENA object to many NAG objects, whose members together comprise 50% of all the remaining instances of input jazzing. These are obviously candidates for Fortran 90 derived types.

The next most frequent categories are input jazzing commands used to build a NAG array from IRENA components (9%) and `template`, which specifies the use of an alternative routine or jacket(7%). Commands for accessing a substructure of an IRENA object account for a further 1%. These are natural operations in Fortran 90.

Remaining are the 6% of commands which are used to allow parameters to be expressed as "more natural objects". These turn out to be concerned with converting IRENA matrices into one-dimensional Fortran arrays, performing simple arithmetic on scalars and padding out arrays with multiple copies of certain elements – all straightforward in Fortran[3].

Thus, it appears that, apart from its data elicitation rôle, all of the functionality of input jazzing could be provided more or less simply in Fortran 90 jackets. We have also seen that, although data elicitation accounts for 80% of all IRENA input jazzing, it involves less than 20% of the range of input jazz commands, suggesting that it is a more easily defined process than data transformation, for which the ratios are approximately reversed.

We shall now consider the tasks carried out by output jazzing. The commonest form of output jazzing, renaming, can again involve long, descriptive names, the choice of which may depend on the values of input and output parameters. As Fortran does not provide any facility for the optional generation of output parameters, at first sight, such renaming must again be carried out in the host system. However, the decision on the choice of name is easily performed in Fortran and this is where the actual values of all input parameters are most readily accessible – some of these may have existed only symbolically in the host system. A suitable mechanism for output naming might, then, be to associate with each NAG parameter two jacket parameters components, one containing the value and the other (a character string) the eventual name to

[3] The one exception is jazzing representing "high-dimensionality" arrays – structures which may represent sets of data in any number of dimensions, depending on the user's problem. These occur in the routine `C06FJF` – for multi-dimensional finite Fourier transforms – which does not form part of IRENA-1. However, a query on the NAG electronic bulletin board, asking for information from users on the types of data sets to which they applied this routine elicited a single response, from a user all of whose data are three-dimensional. Thus, it would seem that not much would be lost if this routine were provided with a special interface for the three-dimensional case, as a separate IRENA-function, whilst the general IRENA-function would retain this particular data set in its NAG form – that is, with the data points and dimensions provided separately. If there proved to be a user requirement for them, other low dimensional interfaces could also be provided.

Command	FL	FL+	Command	FL	FL+
output	395	594	upandslow	2	2
output-order	81	108	output-rectangle	1	2
precedence	19	22	cmplxquots	1	1
i2o	18	34	elements	1	1
out-dims	16	23	lower	1	1
message	16	19	matels2list	1	1
complex-out	13	20	matoverlay	1	1
ragged-out	6	6	outputconj	1	1
vec2rowmat	5	5	out-tuple	1	1
calculate	5	5	upandlow1diag	1	1
cond-out	4	4	append	0	2
up1diagandlow	3	6	sup+dinv2up	0	2
build-rectangle	3	4	cuhessandlow	0	1
reshape-output	2	4	interps	0	1
interpret	2	3			

Key: FL Frequency of usage in Foundation Library jazz files
FL+ Frequency of usage in all jazz files

Table 15.3: Frequencies of output jazz commands

Category	Number of commands		Number of instances	
	FL	FL+	FL	FL+
Renaming or subsetting	3	3	412	618
Control output display	1	1	81	108
More natural object	7	7	32	38
Many NAG ⇒ one IRENA	7	9	26	39
Controlling evaluation order	1	1	19	22
Reflect input	1	1	18	34
One NAG ⇒ many IRENA	4	6	7	12
Conditional output	1	1	4	4

Key: FL Usage in Foundation Library jazz files
FL+ Usage in all jazz files

Table 15.4: General output jazzing classification (principal uses)

be used. An empty string could be used to indicate that the parameter should be suppressed. For ease of handling, the two components should probably form a single parameter of derived type. The other function of the jazz command **output**, namely subsetting, is, here as for input, a natural candidate for Fortran processing.

Of the remaining output jazzing categories, controlling the output display (**output-order**) must be carried out in the host system.

The generation of "more natural objects" largely reflects the input situation. The least obvious cases are perhaps the commands **message** and **explain**. **Explain** replaces an array of coded information (usually integers) with an array of text strings, containing the same information in decoded form; in fact, this could also be easily achieved in Fortran. **Message** generates an extra IRENA output parameter, with an appropriate name, such as **error_control_warning**, when some output parameter has an abnormal value. The extra parameter contains the text of a message, interpreting the meaning of the abnormal value. In fact, this forms a two level warning mechanism: the appearance of the warning parameter in the output list (usually as its first element) being enough to alert experienced users of a routine to the problem; those less experienced can obtain a detailed explanation by examining the value of the parameter, by simply typing **@1;**. If the message text is placed in a Fortran string, the situation here becomes similar to that of the **output** command, except that an additional parameter is generated.

For the many-one (and one-many) mappings between IRENA and NAG objects, the same arguments hold as for input jazzing. Controlling the evaluation order occurs naturally where the evaluation is performed. To reflect input values again requires access to their numerical, rather than symbolic, values and so takes place naturally at the Fortran level. The final category, conditional output, represents the situation where a mathematical object may be stored in one of a variety of locations (or may not be generated at all), depending on a combination of values of input and output parameters. In principle this introduces no new complications for Fortran level implementation – in fact, however, this approach was probably adopted to optimise memory usage in solving very large problems, so duplicating storage for output arrays might be considered unfortunate. However, if the Fortran were carefully generated to exhibit an input-processing-output modularity, the rare occasions when it was essential to work at the limits of memory could be handled by excising the processing module and running this as a free-standing program.

15.3.2 Routine selection

Generally, for the solution of a particular class of problem, NAG provides several routines, each applicable to particular instances of the class.

As we saw in section 13.1, the choice of routine in some instances may easily be made on the basis of simple, objective criteria such as the degree of a polynomial or whether a parameter is real or complex. In such a case, a "super-jacket" in, say, Fortran 90, could be written to include the selection process, implemented as a simple sequence of logical decisions; parameters would need to be of the most general type – for instance, complex rather than real – with coercion to a less general type where needed for a call to a specific NAG routine.

In other cases, as discussed by Dewar [5] and Davenport and Dupée [10], the criteria for routine selection may be subjective ("Is the function fairly smooth?") or may involve manipulations which are more easily performed analytically than numerically[4], such as determining whether the derivative of a function has singularities in a given range.

Determining a more precise meaning for subjective criteria is sometimes feasible, given a sufficiently detailed reading of the NAG documentation and, possibly, access to the author of the routine in question; for example, Dewar introduces 10 different concepts describing varying forms of smoothness (or its lack) which may be used in routine selection. In other instances, or even, as Dewar points out, when the information on which to base a decision could in principle be obtained analytically, the most efficient approach may be to try one routine and, if this either fails or does not converge in an acceptable length of time, to try another.

In summary, although some decisions can be made at the Fortran level, others need the power of a symbolic manipulation package; some problems are most efficiently handled by a combination of routines (including, of course, some for which multi-routine jackets already exist). A sensible "division of labour", then, might be to handle cases which require multiple routine calls (but no intervening symbolic calculation) in Fortran jackets but to leave the actual choice of (possibly jacketed) routine to the host package.

An additional advantage of keeping the routine selection in the host system, together with data elicitation, is that the case where different routines require different parameters can be dealt with more tidily, without the need for additional communication between running Fortran code

[4] Of course, not all calculations required in routine selection are best performed symbolically. The choice of a NAG routine to solve a system of ordinary differential equations depends on the stiffness of the system; Dupée handles this by initiating a separate numerical calculation which uses a NAG routine to obtain the eigenvalues of the Jacobian of the system, since the stiffness may be defined in terms of ratios of these eigenvalues.

and the host system. This even applies to instances where additional information is required by some calls of a particular routine but not others: for example, D01BBF offers the user a choice of two quadrature formulae for semi-infinite ranges but not otherwise. If this user choice were to be maintained in the overall interface, the decision on whether the range was semi-infinite would have to be made as part of the data elicitation process.

15.3.3 Compilation and ASPs

We have seen that much of the processing carried out in IRENA jazzing could be performed in a Fortran 90 jacket in which the appropriate NAG routine calls were embedded. As with the IRENA-1 jackets, these "jazzing jackets" would be precompiled; in fact, there seems to be no reason in principle why the Fortran programs to run these jackets should not, themselves, be precompiled, reading their data, rather than having it embedded in the program as in IRENA-1. In this way, the significant proportion of IRENA processing time spent in program compilation and, especially, loading from the NAG Library could be eliminated, at the cost of requiring storage space for these precompiled programs.

We have not yet considered whether ASPs have any impact on the scheme suggested above; the question of precompilation brings us naturally back to this subject, since at least some ASPs represent input information which could not easily be represented as Fortran 90 data.

About 20% of ASP types can be represented by fixed bodies of code (which could be obtained from IRENA); about another 10% require a matrix, vector or rectangle to be specified and could probably be rewritten as predefined subprograms having a corresponding array as an argument. However, 55% of all ASP types require a function or function family for their definition and, for these, Fortran code corresponding to function definitions in the host system will have to be generated by GENTRAN or some similar system, at run time. The remaining ASP types are concerned with supplying derivatives of known functions: since the determination of the derivatives is an obvious area for the application of symbolic techniques, this will naturally be carried out in the host system and the Fortran must again be generated at run time.

Thus, for about 30% of cases, a permanent body of Fortran 90 code could be produced, and this would enable a worthwhile saving of effort to be made in implementing ASPs for future systems. (The code would not, of course, be a jacket as it would not call any pre-existing routine.) However, for the remaining 70% of cases, a GENTRAN-like generator will be required in (or accessible to) the host system.

The direct impact of ASPs on the "jazzing jackets" is that, where the ASPs have data requirements, these must be propagated up, to integrate with those of the parent routine.

15.3.4 Conclusion

Returning to the main routines, once the more "deterministic" aspects of routine selection have been coded in a jacket, there is still a need for analytic functionality in choosing the appropriate jacket. The sequence of processes outlined at the start of section 15.3 and their recommended location is thus modified as follows:

- pre-processing on host system:
  1. elicit data;
  2. select appropriate jacketed routine(s);
  3. generate code of variable ASPs;
  4. transmit code and data to "numerical server";
- processing on numerical server:
  5. compile ASPs;
  6. link complete F90 program;
  7. run this program
     (which incorporates a jacket to
     - transforms data to routine's format,
     - handle multiple routine calls by calling F90 jacket *and*
     - transform output data);
  8. transmit output data;
- post-processing on host system:
  9. display results.

Fortran 90 compilers, produced by both NAG and other vendors, already exists for the majority of machines on which the NAG Library is mounted and the language will undoubtedly become even more widely available in the future. There would seem to be a strong case for developing any future "jazzing" in Fortran 90, as an extension of the NAG Library.

Where jazzing is to be used as part of the interface to some host package, data acquisition and display modules appropriate to that package must be generated. These might be in a "neutral" language, such as C, in the package's own system language or in a combination of the two.

Obviously, the programmer defining an interface to a NAG routine does not want to be concerned with generating code in a variety of languages and locations. A better solution is to write a control file in a customised control language and process this to derive the necessary components automatically. One of these components would be a Fortran 90 jacket, the others should probably be coded descriptions of the input and output requirements of the jacket which could be automatically processed by package specific programs to generate the appropriate data acquisition and display functionality (either interpretively or in a fully integrated form).

15.3.5 Example

To demonstrate the feasibility of the approach described in section 15.3.1, a Fortran 90 jacket for the (comparatively simple) NAG routine **D01AJF** was generated by hand, together with a module defining the "name and value" output structures suggested in that section. (For clarity, the "second level" defaults for error control parameters were also coded in a separate module; as these quantities may be changed by users at any time, if such a module were included in the final scheme, it could only be compiled at run time. However, it seems more sensible to resolve any use of them as part of the "data elicitation" stage, in the host system.) As suggested in section 15.2, jazzing and default specification were naturally handled together in the jacket.

This particular routine's only ASP represents a (mathematical) function as a Fortran **FUNCTION** – as already pointed out, in a live application this would have been generated at run time. Similarly, if run time compilation is to be kept to a minimum, the non-ASP data (specifying the range of integration, in this case) would be read, rather than assigned.

The code described here, together with a simple test program and its output, is displayed in appendix H. Note that the choice of parameter names for the jacket is arbitrary: the names seen by the user in an eventual product would be determined in the data elicitation and display modules. In an automatically generated jacket, some regular choice of names, such as **jacket_essential_input_***n*, **jacket_output_***n* and **jacket_optional_input_***n*, would presumably be made.

15.4 Considerations for the design of a revised jazzing system

The present jazz system exhibits a number of deficiencies:

- there has been a proliferation of commands,
- the syntax is irregular,
- commands are not, generally, composable,
- complicated manipulations – often requiring interaction with the defaults system – can be required to achieve comparatively simple results,
- control of the order of prompts in the user interface is not available,
- the dimensionality of arrays (for example, to distinguish vectors from matrices) is not readily available,
- the system is not suitable for non-specialist use, especially as it relies on an ability to program in REDUCE symbolic mode for functionality outside its rather limited original core,
- when a parameter of an IRENA-function is itself a function, there is no means of indicating its parameter requirements in a prompt[5].

In addition, as discussed in section 15.1.3, in the present defaults system, the detection of values set in the REDUCE environment is unduly complex and potentially error prone, both for the jazz programmer and for users of the alias system.

Any new, combined jazzing and defaults system should provide a general, uniform, structure manipulating language. Ideally, this should define mappings, rather than objects, to avoid producing large intermediate structures.

The decision to attempt to retain NAG parameterisations and parameter names in IRENA, in addition to the IRENA parameterisation, although reasonable when taken (to address the needs of existing NAG users) was, with hindsight, a misjudgement which added considerably

[5] To do so fully, information on the numbers of various types of parameters is required – these numbers are generally deducible from the value of some parameter of the NAG routine so, here again, control over the order of prompting for parameters is required.

to the complexity of the task of IRENA's authors. As will be seen in what follows, the system now suggested makes no such attempt – rather, it is assumed that, generally, even the names of parameters will change (although the mechanism of defining IRENA parameters does not prevent reuse of a NAG name, should this be appropriate). In the later Axiom NAGlink facility, separate functions are supplied, providing NAG-like and higher-level interfaces; see sections 7.2 and 17.1.

As we saw in section 15.2.1, the most efficient way of achieving a desired change of interface is sometimes to modify the specfile but, as this was seen to be exceptional and consequently error-prone for maintenance purposes, it was avoided as far as possible and jazzing functionality provided instead.

One area which was seen to be particularly troublesome was the introduction of "NAG-like" additional parameters; the jazzing system attempted to provide these through the `scalar` and `vector` commands, which also specified parameters local to just the jazz and defaults systems; the conflict between these two uses was a frequent source of system errors. If adjustments to the specfile had been accepted as normal, these problems could have been avoided – any extra parameters required could have simply been declared there in the same manner as NAG parameters.

As the specfile is designed to be human readable and as the new system being proposed would generate host system code and Fortran 90 jackets automatically from a preliminary file, it seems reasonable to combine the functionality of this proposed new file with that of the specfile, thus gaining the extra flexibility that freer modification of the specfile would introduce. As the present functionality of the specfile would be retained but that of the jazzing and defaults systems replaced, a natural approach is to subsume the system jazz and defaults files in the specfile. Specimens of an IRENA-1 specfile and the corresponding proposed new specfile, as potentially generated automatically and in its final form, are provided in appendix I.

The suggested syntax borrows from both Fortran-90 and Axiom conventions. Although the following description is couched in the present tense, to avoid repetitious use of conditionals, this should not be taken to imply that such a system exists – it is as yet only a proposal. The suggested functionality appears to overcome the deficiencies mentioned above, whilst being sufficiently versatile to duplicate the capabilities of the present jazz and defaults systems.

15.4.1 General features

In addition to the sections at present labelled **TYPE**, **SPECIFICATION**, (NAG) **PARAMETERS** and **IFAIL VALUES**, an **IRENA PARAMETERS** section is added, before **IFAIL VALUES**. Descriptions of any ASPs are provided in a final section and mirror the structure previously used for the main routine. It may be possible to extend the parameter redefinition system defined below to cover the coding of ASPs: in particular, this would require the inclusion of a differentiation operator (relying on the functionality of the underlying symbolic system), since many ASPs are concerned with the forming of Langrangians, Hessians and other derivatives. However, this has not yet been investigated in detail and no attempt has been made to include this in the examples.

The **IRENA PARAMETERS** section would consist of subsections for

Input parameters,

Intermediate input objects,

Input redefinition (including defaults),

Output parameters,

Intermediate output objects,

Output redefinition.

For clarity, declaration of objects' types are kept separate from their definition, in the **parameters** subsections.

End of line is taken as a terminator, unless a bracket is open there. Brackets are **(** ... **)** and **if** ... **endif** (see below). For simplicity of processing, Fortran continuation characters are retained in the routine specification. Spacing is otherwise cosmetic throughout.

Comments are introduced by any percent sign (%) which does not occur within a character string and extend from % to the end of the line, only.

15.4.2 Type declarations

Type declarations for NAG parameters consist of a name, followed in the case of a non-scalar by a parenthesised list of dimensions, the operator : and a type specification with two components: a "general" Fortran type, chosen from

`integer`

`real`

`complex`

`logical` *and*

`character(`n`)` (where n is the length of the character string).

and an indicator of dimensionality, chosen from

`scalar`

`vector`

`matrix`

`3array`

`4array`

⋮

The first component is used directly in generating Fortran code; the second is clearly redundant but its presence unifies the form of NAG and IRENA parameter declarations and removes the need to analyse the dimensions more than once.

IRENA input declarations consist of the name of an object, the operator : and a two or three component specification. The first component may be one of:

`scalar`

`vector`

`matrix` (host system type)

matrices of the various IRENA types

rectangle(*n*) (the (*n*), which is optional, specifies the number of pairs defining the rectangle; if present, it allows additional verification of the user-supplied value)

tuple

list(*component type*)

grid (of 2 or more dimensions)

ragged (ragged array)

function

fset

or a choice of these, denoted as **a ? b ?** ...

– for example, where a sparse matrix **A** is required, we might indicate that any of the IRENA sparse matrix types should be provided by a declaration of the form:

```
A : sparse-mat ? lsparse-mat ? ssparse-mat ? slsparse-mat...
```

The second component is one of the words

data

control

housekeeping

and, where a parameter can be recognised automatically as **housekeeping**, this is inserted in the specification; otherwise, a comment is inserted that the programmer should choose one option.

The optional third component – which has the form **suppliedAs**(*prompt, list of allowed aliases*) – specifies the prompt to be used for the object to which this is assigned and a list of allowable aliases (in addition to those implied by the prompt); *prompt* may be qualified by *:type*, which is used as in the present **set-type** command. Where the third component is omitted, the name of the IRENA object is used in prompting. Parameter lists for function aliases in **suppliedAs** are assumed to be the same as for the prompt. Users may choose any dummy parameters in supplying the function but those specified here will appear in prompting[6].

[6]This contrasts with the present system in which, as was noted in the footnote on page 166, there is no mechanism in to allow information concerning the parameters required in a data function to be provided to the

The `Input parameters` section includes those ASP data requirements which can be (tentatively) inferred automatically.

IRENA output declarations are similar to input declarations without the final component and with choices disallowed. The dimensions of IRENA output matrices etc. should be specified; an asterisk (`*`) may be used for "obvious" dimensions (for example, the second dimension of a matrix formed from a vector, when the first is given, or the dimensions of a matrix formed by concatenating vectors). IRENA output names may coincide with NAG names.

"Output order" is defined implicitly by the order in which IRENA output parameters are specified.

It is anticipated that output matrices will commonly be specified as native host system structures but the possibility of the IRENA representation is retained to deal with, for example, sparse matrices.

15.4.3 Redefinitions

The code in the redefinition sections is obeyed sequentially. The operator `:=` is used in redefinition.

Any name on the right hand side of an assignment is taken as an IRENA name unless it is the argument of one of the functions **`in`** or **`out`** (see below).

Any redefinition may be regarded as a parallel assignment of elements on its right to those on its left; any structure is considered to have an implied order attached to its elements, in general, this is a row-major order; details of the implied orders for all structures are not given here but may be summarised as last index changing fastest in some "natural" representation of the structure (for example, a **`rectangle`** would be regarded as an $n \times 2$ array, so the elements would be taken in the order in which they appear in defining a rectangle). Structures consisting of nested lists are effectively flattened. A consequence of this is that more than one object may appear on either side of an assignment; commas (`,`) separate the individual objects.

---

user, only the name of a function being displayed when prompting. Here, when, for instance, the number of parameters is fixed, they may easily be displayed in a manner which suggests their meaning, as in prompting for a function to define a second derivative in terms of the coordinates and first derivative with **`f(x,y,y')`**? Even when the number of parameters is variable, the form used could be more more suggestive than at present, as in **`f(x1,..,xn,t)`**?

In the redefinitions, the Fortran 90 convention on array sections is assumed – that is, a pair $m:n$ in the place of an array dimension specifies the desired range of values for that dimension. In addition, certain functions (**diagonal**, **upperTriangle** and **lowerTriangle**, see below) are used to specify subsets of elements of structures. These conventions may be utilised on the left or the right of an assignment, in each case specifying the elements to be processed and the order of their processing.

The input redefinition section is used to supply definitions for all NAG parameters in terms of declared IRENA input and intermediate parameters and constants. Where parameters can be automatically recognised as housekeeping, values may be provided automatically. Default values are introduced by = as in `N := irena_n = dim(A,1)`.

An **if** ... **then** ... **elseif** ... **then** ... **else** ... **endif** construct is provided, with the standard numerical comparison operators.

The usual arithmetic operators operate componentwise on conformable structures; when one operand is a scalar the operation is applied to the lowest level components of the other.

The imaginary element i is represented as **%i**.

The following functions are available; unless otherwise indicated, they may be used in both input and output redefinition:

absent(*p*) (input only) returns **true** if the irena parameter *p* has not been supplied by the user in the keyline (or its host system equivalent) – or in the global environment if the user requires this to be searched for this class (**data** or **control**) of parameter.

concatenate(*list*) concatenates the flattened elements of *list*; if the last element of *list* is **unset**, all trailing elements on the left hand side are **unset**. If the object on the left hand side of the assignment has dimensionality greater than zero, the objects on the right are flattened, from the inside out, until objects of dimension one less than the left hand's are produced.

copies(x,n) provides n copies of x.

diagonal(*name*,r,c) supplies a list of the elements of *name*, on the diagonal through (r,c).

dim(*name*,n) (input only) the nth dimension of the vector, matrix, array or fset *name*.

global(*host system variable name*) supplies the value of the host system global variable.

imag(x) supplies x with each bottom-level component replaced by its imaginary part.

`in(`*nagname*`)` (output only) the input value of the named NAG parameter.

`keyword(`*k*`)` (input only) true if the keyword *k* is present in the keyline or its host system equivalent.

`length(`*x*`)` supplies the top-level length of *x*.

`lowerTriangle(`*name*`,`*r*`,`*c*`)` supplies a list of the elements of *name*, on and below the diagonal through (*r*,*c*), in row-major order.

`map(`*fun*`,`*object*`)` applies *fun* to each top-level component of *object*.

`max(`*l*`)` (input only) supplies the maximum element of the numeric-valued structure *l*.

`min(`*l*`)` (input only) supplies the minimum element of the numeric-valued structure *l*.

`out(`*nagname*`)` (output only) the output value of the named NAG parameter.

`present(`*P*`)` (input only) returns `true` if the irena parameter *P* has been supplied by the user in the keyline (or its host system equivalent) – or in the global environment if the user requires this to be searched for this class (`data` or `control`) of parameter.

`promptIfUndefined()` (input only) causes a prompt for the variable to which it is assigned to be generated if that variable does not already have a value.

`real(`*x*`)` supplies *x* with each bottom-level component replaced by its real part.

`sum(`*l*`)` (input only) sums the elements of the numeric-valued structure *l*.

`transpose(`*m*`)` the transpose of the matrix *m*.

`upperTriangle(`*name*`,`*r*`,`*c*`)` supplies a list of the elements of *name*, above and on the diagonal through (*r*,*c*), in row-major order.

Note that, in applying these functions, intermediate objects need not be explicitly calculated – for example,

```
transpose(upperTriangle(transpose(X),1,1))
```

which gives the lower triangle of `X` in column-major order should not generate the object `transpose(X)` but should simply redefine the order of a subset of `X` as part of an overall transformation. However, as a considerable additional programming effort would probably be required to avoid the creation of intermediate objects in a general manner and a set of *ad hoc* manipulations is undesirable, an initial system would probably accept the overhead of creating these objects, so that their impact on efficiency and utility could be determined.

A table of IRENA input parameter values is built up by

1. taking keyline values,
2. taking environmental or default values for those parameters in the classes (`control` and/or `data`), for which the user has specified that this source has precedence over prompting, according to the specified precedence,
3. performing a single pass through the redefinition code and prompting for any required IRENA parameters, as these are encountered.

At stage (3), Fortran code for NAG parameter values will be generated. Thus, representations for all IRENA input parameters but no NAG parameters will exist in the host system; initially, the revised specfile generator provides a one-to-one correspondence between those NAG and IRENA input parameters which cannot be automatically classified as housekeeping – the jazzing programmer will remove those which are unnecessary – and includes values for those which can be so classified.

As an example of how the old jazzing commands map onto the new, `phased-prompt` would require a new IRENA input variable which would be tested before the main variable was processed, so, to mimic the effect of the present command

```
{phased!-prompt} INIT : do! you! wish! to! supply! an! initial! approximation
                        (n > unset) initial! approximation! to! solution
```

with keywords `no_initial_approximation` and `noia` also being allowed, we would have:

under `Input parameters`:

```
irena_init_flag : scalar control suppliedAs("Do you wish to supply an
                                    initial approximation":"(Y or N)",
                                    << >>)

irena_init : vector data suppliedAs("Initial approximation", << ip >>)
```

under **Input redefinition**:

```
irena_init_flag :=
   if present(irena_init)                       then unset
      elseif keyword(no_initial_approximation) then 'N
      elseif keyword(noia)                      then 'N
      else                                           promptIfUndefined()
   endif

irena_init := if irena_init_flag = 'Y then promptIfUndefined()
              else                         unset
              endif

INIT := irena_init
```

This allows the various possibilities – keys, prompts, necessary and unnecessary parameters – to be thought through in a straightforward, sequential way and programmed accordingly.

The assignment of **unset** to **irena_init_flag** could equally well have been **'N** but this would be less transparent for those reading the code.

In a data-reading Fortran based system, such as that suggested in section 15.3.3, **unset** would probably be handled in the Fortran by having an auxiliary variable read first, with actual data values only read when the auxiliary variable has an appropriate value.

Chapter 16

Recommendations for NAG Library development

The development of IRENA led to the identification of a number of areas in the design of NAG software and documentation which could usefully be revised, to enable the material to be incorporated more simply into packages.

Section 16.1 is a slight paraphrase of a paper, given at the end of 1992 to members of NAG's Numerical Libraries Division, who are the principal developers of the NAG Library.

16.1 Software guidelines for library development

The main requirement in NAG Library software, revealed by the IRENA project, can be summed up briefly as maximum consistency in the interfaces of the Library routines.

Some of the more notable problems attributable to the Library itself, experienced in developing IRENA, are described below.

16.1.1 Argument SubPrograms

The large (and growing) variety of ASPs was a major component in slowing the development of IRENA. Each variant requires detailed individual programming at the RLISP level, to allow a symbolic description of its functionality to be mapped into Fortran code. This is an area in which strict adherence to a limited set of models would provide substantial benefits for derived products.

16.1.2 Option setting

Those **E04** routines which use option setting had to be omitted from the first release of IRENA, since the mechanisms required to handle their parameters differ completely from those built into IRENA for normal routines.

16.1.3 Uncontrollable termination

The non-standard error reporting mechanism in the **F07** chapter breaks IRENA if it is invoked. Library routines should never terminate a program without allowing a user override.

16.1.4 Natural data representations

Are far as is practicable, NAG parameters should correspond to individual mathematical objects. The most troublesome departures from this precept are the cases where the location of a particular item of data is variable. Examples are the array **W** in **D02YAF** and the location of the singular vectors in **F02WEF**.

A notable instance of an unnatural parameter representation is that, in a number of **D01** routines, the length of a workspace array, which would normally be considered a housekeeping parameter, has a control rôle, in that it restricts the degree of subdivision allowed in the quadrature. A more natural approach would be to provide a parameter which specified the degree of subdivision allowed and define the length of workspace array required in terms of this.

(Another example, which does not in itself affect package development but which does present users with an unnatural interface, is the parameter **D01XXX** in **D01BBF**, where the choice of quadrature formula must be specified by supplying the name of a NAG auxiliary routine – a

technique perhaps adopted to avoid increasing the size of the load module, at a time when memory efficiency was an important consideration. This choice could now be specified much more naturally by a string-valued parameter indicating the name of the required formula.)

16.1.5 Matrix representations

The many alternative matrix representations in the Library caused us to decide, at an early stage, largely to ignore these and build a set of IRENA matrix types in their place. For instance, our symmetric matrix type allows the specification of either triangle and automatically supplies the other. If a minimal set of representations had existed in the Library, this would not have been necessary. For the benefit of future package builders, such a set should be adopted, building on the existing commitment to a consistent representation of symmetric, skew-symmetric and Hermitian matrices on input (and, preferably, extending this to allow either triangle to be supplied in every case).

16.1.6 Complex quantities

There is a variety of representations of complex-valued structures in the Library – as complex valued arrays, pairs of real (usually `DOUBLE PRECISION`) arrays and real arrays with an extra dimension, 2. Ideally, a single representation should be chosen and since, for Fortran 77, the requirements for higher precision rule out the use of `COMPLEX`, the present most common solution – to use a pair of real arrays – should be adopted uniformly in the Fortran 77 Library. As a special case, complex scalars should be represented as a pair of real scalars. (This problem does not, of course, arise in the case of Fortran 90 developments, where developers have sufficient control of the precision.)

16.1.7 Naming conventions

One of the aims of IRENA is to achieve a high degree of consistency in the naming of equivalent objects throughout the package. The use of alternative names for similar objects is a minor problem, provided that it is recognised. More serious is the use of the same name for different objects: for instance, matrices to be processed by NAG routines are almost always called `A`. However, this name is sometimes used for a single component of the description of a matrix – usually a set of values, without full information on their location.

In IRENA, we reserve A for the name of the matrix itself – thus risking possible confusion by users familiar with the Library. (We do, of course, draw special attention to these cases in the documentation.) Ideally, in the case described, an alternative name such as **AVALS** should be adopted in the Library.

16.1.8 Misclassification and misplaced information

In a number of instances, parameters have been described as *Input* in the routine document when they have, in fact, been *Input/Output*. This, of course, causes problems in the automatic processing of the information. This type of error is probably best avoided by coding the parameter descriptions into the source of the routines, as they are written – effectively "declaring" parameters as *input* etc. This information could then be automatically extracted in generating the documentation.

Harder to detect, and therefore more troublesome, are cases where there are errors in the specification section – such as missing commas after parameter names at line ends or (in **F04AXF**) `IKEEP(N,5)` instead of `IKEEP(N*5)`. Some of these errors could be detected by automatically parsing any Fortran fragments included in the routine documents.

In some routines, parameters are described as *Workspace* – but a different routine document reveals that this workspace contains useful information. For instance, the information in the "workspace" parameter W of **E04FDF** is described in **E04YCF**'s routine document but not in its own.

16.1.9 Special data representations

The frequency of such special representations is a major problem – in fact, this must compete with the variety of ASP types as the single factor which most delayed the appearance of IRENA. The representations in question range from the case of **C02AJF**, which, unlike the other C02s, requires the coefficients to be specified as scalars and which does not return all the roots in the same array, through HMAX in **D02KEF**, in which the first row contains data and the second is workspace (most other multi-part input matrices are partitioned by columns), to the various three-dimensional arrays of coefficients in **D03ECF**.

Also worthy of note are routines with non-standard documentation (in particular, the **F06s**[1]).

[1] See section 16.4.1.

16.1.10 Assumed-size arrays

Where assumed-size (*) dimensions occur, the presence of constraint information is very helpful in processing them (but is not always present).

16.1.11 Summary

The task of any package developer is simplified by the extent to which the underlying components' parameters meet the desiderata which were recognised in the design of IRENA parameters – namely, that they should be:

- Informative

  parameters should have meaningful names

  routine usage should be easily learnt

- Regular

  different routines should be similarly parameterised

- Orthogonal

  distinct items of information should be kept separate

and

- Minimal

  information should be simple to input

  information should only be obtained when required

  the proliferation of parameters is to be avoided.

16.2 Material for direct inclusion in symbolic packages

Many symbolic packages already offer some internal numeric capability; where NAG material is being considered for incorporation in such a package, there are, at least, two possible approaches – the existing NAG Library routines could be interfaced in some way to the package, as has been done for IRENA and the Axiom-NAG link, or the NAG code could be translated into the package's own system language and incorporated directly.

In the longer term, the second approach has much to commend it, since a more integrated product is likely to result. However, such an approach also has implications for the longer term development of NAG numerical software. Since the philosophy behind symbolic computation does not, generally, admit to fixed precision computation and, even less, to computation where the actual precision of the result is known only approximately, there is a need to concentrate on the development of techniques for which precise error bounds can be calculated and which, ideally, are capable of being applied with arbitrary precision.

In reality, the requirement for arbitrary precision may, in some situations, be usefully relaxed. A distinction can be made between "mathematical" and "engineering" style applications. The former, which would include pure mathematics and some areas of theoretical science, may genuinely require arbitrary precision – but, of the areas traditionally covered by the NAG Library, this is likely to affect only a relatively small number, for example, the calculation of standard transcendental functions and, possibly, some linear algebra. For "engineering" applications (which would include most other areas, such as applied science and finance) it is usually adequate to obtain results which are known to be correct to a few significant figures; however, given the premise that the results of symbolic packages' computations should be, in some sense, provably correct, even for these applications only algorithms which produce exact error bounds should be incorporated.

In the symbolic package it is, of course, essential to distinguish between the two types of result: for example, in Axiom they could belong to different (formal) types.

16.3 Packages as sources of library material

In sections 12.2.4 and 12.3.1, mention was made of how IRENA jackets could simplify the use of NAG routines. Future IRENA-like projects are likely to make increasing use of such jackets, probably written in Fortran 90, to provide simple user interfaces to existing NAG routines; these should not be overlooked as a possible source of code for the NAG Fortran 90 Library. Duplication of effort may be avoided if NAG library and package developers collaborate in specifying such jackets.

16.4 Structure of NAG documentation

The documentation of the NAG Library was the single most important external component in the production of IRENA, providing information both for automatic generation of IRENA components and for the manual activities such as jazzing and defaults definition. The ease or difficulty with which this information can be used is critically dependent on how well it is structured: the nature and extent of each distinct item of information should be clearly marked (facilitating both machine and human analysis of its content) and a consistent structure is needed throughout.

One area where IRENA has led to an improved structuring of the Library manual has already been remarked on in section 7.1, namely the specification of "suggested values". However, there remain instances where this information is embedded in general text, rather than being explicitly flagged. For example, in the **F01BRF** routine document, the description of **LICN** states that it "should ordinarily be 2 to 4 times as large as NZ" but no formal suggested value is given. For IRENA, the default value was set as

```
LICN : 4*NZ
```

– in such cases, safe values should be made explicit in the NAG documentation.

Another, similar regularisation in the Library manual, due to IRENA, is the presentation of constraints on parameter values. Although this information is now better structured, instances remain where it is not explicitly presented in the appropriate location.

At times, the constraint information is presented as an English language description, rather than as a mathematical inequality. For example, the constraint given for the parameter **H** in **C05AVF** is "either X + H or X - H must lie inside the closed interval [BOUNDL,BOUNDU] (see below)". This could be used much more easily in automatic processing if expressed as "BL - X $\leq$ H $\leq$ BU - X or X - BU $\leq$ H $\leq$ X - BL".

Further, "obvious" constraints (such as requiring a step size to be positive) are sometimes omitted. Although these may be considered to be apparent to users, this is a subjective judgement; in any case, they should be certainly made explicit for automatic processing systems.

The NAG documentation's "Error Indicators and Warnings" sections are another area where improved structuring would be helpful to package developers who need to base error messages for derived packages on those of the incorporated NAG routines. Here, related headings are sometimes grouped – for instance, for the routine **C05AGF**, the **IFAIL** values 5 and 6 are described together: they "Indicate that a serious error has occurred in C05AVF or C05AZF respectively".

Fairly extensive internal cross-referencing also occurs here; for example, **C05AJF**'s description of **IFAIL** = 4 includes the remark that "This error exit can occur because NFMAX is too small ...or for either of the reasons given under IFAIL = 3 above". Other cross references may also need to be expanded, for instance, the common "see Section ..." at very least needs to become "see Section ...of the appropriate NAG routine document" if it is to be used as the basis of a package's error message.

In recent marks of the Library, a facility has been added for many routines to print their own error messages; as this develops, the messages may be expected to become much more self-contained and, provided that the internal message forms are also utilised in the documentation, they should provide a better basis for error messages in derived packages. For direct use to be made of such messages in packages, they should describe what has occurred in terms of mathematical objects, rather than Fortran parameters; it may be that this could best be achieved by allowing an additional input **IFAIL** value to indicate that this style of output was required.

A further improvement, which would be useful in processing exceptions, would be to make an explicit distinction among **IFAIL** values, classifying them as "structural errors" (that is, due to the detection of the violation of a stated constraint), "execution errors" (in effect, all other errors – for example, a failure to converge) and "warnings". In a system which is set up automatically, structural errors are very unlikely to occur in housekeeping parameters and it is probable that no special action would be needed to process these; furthermore, it is unlikely that the facility to express error messages mathematically, mentioned in the previous paragraph, would be required for such cases. If a package uses a technique, such as IRENA's jackets, which may involve multiple NAG routine calls from a single command, the action required when an error occurs will almost certainly be different to that taken in response to a warning; distinguishing between these would increase the ease of automatic processing.

16.4.1 F06 documentation

In section 16.1, mention was made of the documentation of the F06 (linear algebra support) chapter. This is arranged quite differently to the documentation of other chapters, probably because the material here is meant mainly to provide an efficient linear algebra underpinning for use in other NAG routines and is expected to be only secondarily of interest to end users. However, the material is of some interest to end users and may also be required for internal incorporation in packages, for instance, in the provision of ASPs. Unfortunately, this non-standard organisation completely frustrates any attempt at automatic processing which is not designed specifically for this chapter (and also makes the task of users, including package developers, who need to incorporate this material considerably more difficult).

In outline, the F06 documentation is organised as follows.

Section 1 consists of a single sentence, outlining the scope of the chapter.

Section 2 gives the background to the problems in the entire chapter.

Section 3 gives the purpose of each routine: here, the routines are arranged into four categories:

scalar

vector

matrix-vector and matrix

matrix-matrix

each with a subsection of "Basic Linear Algebra Subprograms"[2] (BLAS) and, except for the last, another of "other routines".

Section 4 contains the routine descriptions, now arranged in *five* categories, again with the "BLAS" and "others" subdivision. Descriptions of uniformly named parameters are usually given at the beginning of the section describing each category, although the correspondence between similarly named variables in section 2 and parameters here tends to be implicit and the description may consist, wholly or partly, of a cross-reference to a

[2]The Basic Linear Algebra Subprograms are an agreed, standard set of numerical linear algebra utilities, designed for efficient implementation in both sequential and parallel computing environments, to serve as the basis on which efficient user-orientated numerical linear algebra software may be built. See [7], [8], [9] and [18].

similarly named parameter in an earlier section; descriptions of parameters applicable to a single routine or closely related group of routines are given in the description of those routines.

Section 5 lists routines which have been withdrawn or are scheduled for withdrawal.

Within any subsection, routines are arranged alphabetically by the fifth letter of their names, which has the effect that equivalent routines with real or complex parameters appear together – there may be more than two versions, since more than one parameter may be involved; the fourth letter of the names appears to depend only on the sequence in which the routines were introduced and so does not aid in determining their positions within the chapter.

The style of organisation of the information in this chapter has the effect of making the task of understanding a routine's usage particularly time-consuming, since the information is so scattered.

Chapter 17

Impact of IRENA on other NAG software

17.1 Impact on the Axiom-NAG link

The experience gained from the development of IRENA influenced the development of the Axiom-NAG link in a number of ways, affecting both its overall design and detailed implementation.

For the overall design of the new link, a less ambitious and more incremental approach was adopted. This resulted in a first release of the link with the following characteristics.

- The set of routines was based on that adopted for IRENA-1, but was slightly smaller, since the `A00` (Library identification), `M01` (sorting) and `X01` (mathematical constants) routines were omitted (although the three Foundation Library `D03` – partial differential equation – routines were included).
- A more Fortran-like interface was accepted for this first release: in this, all non-housekeeping input parameters were visible (as well as certain housekeeping parameters, as described in the next item).
- Axiom includes a facility for visual templates to be provided for command construction, with each parameter having its own input area in the template. At the suggestion of the

present author, user interfaces were developed for the linked routines, based on this facility, so that default values could be displayed as initial settings in the corresponding input areas. (In fact, this idea was taken further by the link's developers and initial settings equivalent to the standard NAG example program were displayed for all visible parameters.) As a means of controlling the sizes of the arrays' input areas, those housekeeping parameters corresponding to array sizes were made visible in this interface.

- The execution of the NAG routines was decoupled from Axiom, the two processes communicating through sockets, using the **xdr** protocol. This avoided the use of proprietary software items such as **oload**, (whose interaction with the Sun loader caused considerable difficulty in IRENA) and allows Axiom and the NAG routines to be run on different machines, if desired. As suggested in section 15.3.3, precompiled programs calling the NAG routines were used, where feasible, to reduce the time required to process calls (in particular, that taken for loading from the NAG Library which, as we have seen, usually dominates the processing time). However, the main programs were produced (automatically from the IRENA specfiles) in C, to allow communication through **xdr** and for control of error handling (since no general error recovery mechanism is available in Fortran for run-time failures). To reduce the volume of material stored, these programs were not retained permanently but were compiled on the first occasion they were required in any session and retained throughout that session.

IRENA components were reused wherever possible. In particular, the specfiles were available as a reliable source of routine-specific information, from which, as has already been mentioned, C main programs for the routines could be produced. In addition, default values, some jackets and the classification and logic of the ASPs were utilised.

Drawing on the experience of IRENA, additional jackets were written (by Dewar) to avoid some of the irregularities which IRENA had revealed, for instance, allowing the choice of quadrature formula in **D01BBF** to be specified as a number rather than as the name of a NAG routine.

Where the interface was changed, no attempt was made to provide a completely NAG-like interface as an alternative. This decision was encouraged by the ease with which function names can be overloaded in Axiom, where functions with the same name but different signatures – that is, lists of the types of the parameters – are regarded as distinct. This meant that, if required later, more and less NAG-like versions of routine interfaces could be produced as separate functions which could, if desired, both retain the NAG name. In fact, the NAG name was only retained for the NAG-like interfaces, produced for the first release of the link.

The present author has only recently become involved in the detailed development of the link: in particular in developing Axiom-like interfaces for the second release of the link. The functions providing these are given names which indicate their purpose, prefixed with nag to indicate their use of NAG routines (and consequent limitations on their accuracy, implicit in the numerical algorithms which these routines implement). These interface functions call their NAG-like counterparts: where different NAG routines provide the same functionality for distinguishably (in Axiom) different data types, separate Axiom functions with identical names are developed; where the distinction in the input data is more subtle – for example, between symmetric and asymmetric matrices – branches to the appropriate NAG-like function calls are coded in a single interface function.

This approach allows the NAG-like and Axiom-like interfaces to be kept separate, unlike the situation in IRENA, which attempted to provide the equivalents of these as alternatives in each IRENA-function (see section 15.4).

The IRENA experience has already proved useful in this exercise, in a number of ways. For example, in planning the (Axiom-like) command line interface, the present author was able to characterise this in about one working day, mainly by drawing on the classification of input parameters into essential and optional in the IRENA function description documents, adopting a strategy based largely on the model of the IRENA "mnemonically-named functions", discussed in chapter 13.

Rather than attempting to provide natural interfaces to a large fragment of the NAG Library, he has chosen to identify areas in which Axiom's numerical capabilities need extension and to concentrate on providing uniform interfaces in those areas. At the time of writing, the areas which have been covered are the NAG special functions chapter, quadrature for functions approximated by polygons, discrete Fourier transforms and matrix eigenvalues and eigenvectors; other areas (such as interpolation and optimisation) will be identified and interface packages written, as time permits, until the release of the new version.

The NAG Fourier transform code makes frequent use of "Hermitian sequences"; the IRENA (RLISP) code for handling these, described in section 9.6, proved to be almost directly translatable into "Axiom extension language" (Axiom-XL) code as part of this exercise.

17.2 Impact on other software

17.2.1 Fortran 90 Library

At the time that the interfaces for the routines of the Fortran 90 library were being designed, a complete set of IRENA function description documents was made available to the designers, as an indication of the minimum logical requirements for input to the existing Fortran 77 routines.

It is worth remarking that the facilities in Fortran 90 for keywords and optional parameters bear a closer resemblance to the design of IRENA than to that of Fortran 77.

17.2.2 The NAG Product Information Database

One by-product of IRENA was the abstraction from the NAG documentation of information on the routines in the Library, in a machine processable form (the specfiles). The need for such information to be available on a wider basis, in a form easily analysable for incorporation in derived products such as IRENA, was a major consideration behind the development of the NAG Product Information Database (NPID), as the basic repository of information on NAG products, including (but not limited to) the Library documentation.

The development of the NPID could also be regarded as the logical extension of the revision of the manual already described in section 7.1 and the further revision suggested in section 16.4, in that the information on the routines was further decoupled from their documentation requirements. In particular, information generated in the IRENA project, such as versions of error descriptions with cross-references resolved and reliable default values for parameters (as opposed to "usually adequate" suggestions), was incorporated.

17.2.3 The MATLAB gateway generators

This collection of software was developed, after IRENA, to provide access to NAG routines from the MATLAB numerical computation environment. It was able to draw on the information in the NPID (and so, indirectly, on IRENA) as the basis of its development.

Chapter 18

Final overview

A number of the principal conclusions drawn in the body of this thesis are reiterated here and further conclusions are presented.

18.1 Effort required with more complex interfaces

Perhaps the most interesting of the earlier conclusions is the lack of scalability, discussed in sections 8.2 and, especially, 8.3, of experience drawn from using or processing simpler Fortran routines to more complex cases. In particular, there is good evidence for a marked nonlinearity in the relationship between the complexity of the interface of a Fortran routine and the amount of effort required to redefine that interface. In practical terms, this may be summarised by saying that Fortran routines with few parameters give a deceptively encouraging picture of the ease with which the redefinition may be carried out: the slope of the graph of effort (measured as the amount of code required) to redefine an interface against parameter count increases rapidly for the first few parameters and only then becomes reasonably constant.

It seems reasonable to suggest that this experience may be generalised to imply that the effort required to use a library procedure also increases rapidly with the number of parameters of that procedure and that there is a rational basis for the preference of library users for interfaces with few parameters. To understand and use a pair of procedures, each with four input parameters, may well be expected to be a considerably easier task than to do the same for a single procedure with eight: provided that the transfer of data between the two is hidden from the user, there

seems no reason why splitting a complicated procedure into smaller units should result in an increase in the total number of input parameters required.

Of course, IRENA also produced an absolute decrease in the number of parameters required in using many routines; this is discussed further in the next section.

A related issue is the additional effort required to produce a single interface to a number of related routines. Appendix G illustrates a fairly simple example of this, in which a REDUCE-like interface is provided to a set of four IRENA-functions for the solution of polynomial equations (two of which correspond to routines in the full NAG Library which are absent from the Fortran Foundation Library).

The entire interface in this case required only 150 lines of RLISP code (including blank lines). This may be contrasted with the individual interfaces provided for the four NAG routines – these required jazz and defaults files totalling 102 lines. It seems that the task of providing a common interface may require an additional effort similar to that of providing the original, individual interfaces; the somewhat greater amount of code for the common interface may be attributed to RLISP being a rather "lower-level" language than the IRENA jazz and defaults languages.

However (especially in view of the lack of scalability mentioned above) no strong conclusion should be drawn from this single example. It is worth noting that most of the RLISP code (111 lines) was concerned with converting sparse REDUCE coefficient sets to the dense representation required for NAG – essentially, defining an additional jazzing operation; in view of the small amount of code required for the rest of the interface it is certainly worth investigating common interfaces further at an early stage of any future project in this area.

18.2 Ease of use

Reported experience of IRENA users suggests that it represents a considerable advance in ease of use on the original NAG Library routines. The analysis in section 14.4 gives an objective basis for this impression, in that, for the routines examined, both the individual documentation and the program required to use the routine were reduced in size by more than 50% in every case. A factor in this is the reduction in the number of input parameters required by an IRENA-function, compared to its underlying NAG routine. In addition, users of NAG routines must define workspace parameters and output arrays of the correct dimensions for the routines' use.

The relative numbers of parameters requiring user-supplied information are listed in table 18.1, for the routines considered in section 14.4. For this table, **EXTERNAL** subprograms have been included as input parameters, although they may, of course, require considerably more specification than other input parameters; the NAG error parameter **IFAIL** has also been included as an input-output parameter.

Routine	Numbers of user-supplied parameters				
	NAG Library			IRENA	
	Input and I/O	Workspace	O/P arrays	Essential	Optional
C06EAF	3	1	0	1	1
D01BBF	6	0	2	4	2
E01SEF	9	1	1	1	4
E04DGF	4	4	1	2	4
F02ADF	6	1	1	2	0
F04MAF	14	0	1	2	4
S14BAF	4	0	0	2	1
S18DCF	5	0	1	3	1
Overall	51	7	7	17	17

Table 18.1: Numbers of user-supplied parameters, NAG and IRENA

From this table it may be seen that, overall, for the routines in question, the number of parameters which must be specified in IRENA is between 26% and 52% of the number required in the corresponding NAG routine calls. Given that some of the NAG parameters are themselves user-supplied subprograms, which may require significant effort in their specification, there appears to be adequate justification for the reported simplification of usage, especially as the effect of the number of parameters may be expected to be amplified by the non-linear relationship seen in section 8.3.

A further factor in the simplification which is not apparent in this table is the increased consistency of parameterisation in IRENA, compared to the NAG routines.

As was noted in section 5, to maintain the standardisation of parameter names, it was necessary to periodically review the names in use. This same discipline was, of course, extended to the actual parameterisations used, although there was less likelihood of divergence here, due to our decision to match the parameters to mathematical objects as closely as possible.

In this way, divergences of approach, particularly in naming conventions, were detected and rectified in a number of areas[1]: similar consistency reviews are recommended in the development of any body of software presenting a number of interfaces to users.

To reiterate a final point concerning interface simplification, it is sometimes possible to achieve a considerable improvement in ease of use by accepting a slight loss of generality. An example of this was seen in section 6.6, with the user-supplied `OUTPUT` subroutine – required by some `D02` routines to specify when solution values should be printed – being replaced by a simple vector of output points. It is particularly worthwhile accepting such a trade-off when, as in this case, the means of "repairing" the loss of generality can also be provided.

Additionally, the development task itself may be considerably simplified by the decision to accept some loss of generality: this was particularly evident in the effort required to maintain the NAG parameterisations in IRENA-functions. As was pointed out in section 15.2.1, maintaining this dual interface inhibited the translation of NAG error messages to refer to IRENA parameters; this, in turn, led to the development of an entire additional subsystem, the IRENA help system, described in section 9.4.

18.3 Unification of control mechanisms

In section 15.2 we examined the interconnections of the defaults and jazzing mechanisms and saw how a more natural facility for redefining interfaces could result if these were merged; a further simplification could be achieved by subsuming this merged system in the "specfile", the first point at which human intervention becomes necessary in the production of an IRENA interface to a NAG routine.

A general lesson which may be drawn is that, where more than one control mechanism exists, consideration should be given to the possibility that these represent different aspects of the same process and could be merged. As this may not be apparent from the outset, the periodic reviews recommended in the previous section for the maintenance of consistent interfaces should be extended to consider, at times, the possibility of unifying the internal mechanisms used in system development.

[1] A by-product of this, the consistent naming of variables in the code of jazz- and output-functions, was found by the author to be helpful in the later translation of these into interface functions for the Axiom-NAG link.

18.4 Functionality and multiple numeric calls

One of the initial objectives given to the authors of IRENA was the creation of simplified interfaces to the entire NAG Fortran Library. However, as was seen in section 8.1, this was soon found to be impracticable and a considerably reduced target set of routines was adopted. In this context, the revised objective was phrased as supplying such interfaces to all the routines of entire Library chapters, albeit to the considerably smaller chapters of the Foundation Library. Even this now seems too broad a strategy – a more selective approach, based on identifying functionality missing from the host package and building interfaces to those NAG routines which can supply this functionality, is clearly much more effective as a means of enhancing the host package and has been adopted by the author in providing Axiom-like interfaces to NAG Foundation Library routines, on a very restricted time-scale of only a few months, prior to the next release of Axiom (the first release on PCs).

In contrast to the approach of supplying interfaces to particular NAG routines, section 12.2 of this thesis considered various areas in which enhanced user interfaces could result from using the potential of IRENA-functions to produce several calls to NAG routines – possibly calls to a number of distinct routines or multiple calls to the same routine. Similarly, calls to alternative routines, chosen according to characteristics of the problem, could result from the same IRENA-function. A particular example of the potential for multiple calls, in which a second routine call could be made after a failure, was discussed in section 12.2.4 – in the chosen example, the same routine would be re-called with adjusted parameters but, of course, in other instances an alternative routine might equally well be called.

In the context of section 12.2, multiple calls would have been achieved through a Fortran jacket. However, a similar approach may be taken in any sufficiently versatile language: for the Axiom-NAG link, the technique of using a second routine where the first fails was used, in interfaces written in the Axiom system language Axiom-XL, to provide user interfaces for the generalised eigenproblem, using routines from the NAG `F02` chapter. In this chapter, there are separate routines for solving the generalised real eigenproblem $\mathbf{Ax} = \lambda\mathbf{Bx}$ – efficiently when both the matrices $\mathbf{A}$ and $\mathbf{B}$ are symmetric and $\mathbf{B}$ is positive-definite – or less efficiently, otherwise.

Whilst it is possible in principle to test algebraically whether matrices are positive-definite, this is impracticable for large matrices. However, the NAG routines for the positive-definite case test this property (numerically) at an early stage of the processing and return an `IFAIL` value

of 1 if it does not apply. The author's strategy, in the NAGlink interfaces, is, if both matrices are symmetric, to call the routine for the positive definite case then test for **IFAIL = 1** and, if so, use the more general routine, **F02BJF**.

An example of the output from such a run, in which only eigenvalues are calculated, follows. (The appropriate NAG routine for eigenvalues only, in the positive-definite case, is **F02ADF**.) In this run, results of type **FormalFraction Complex DoubleFloat** are produced – **FormalFraction** is an Axiom type constructor introduced by the author to allow users to inspect quotients whose components are calculated separately, before evaluating them as **DoubleFloat** quantities, in case there are components small enough to cast doubt on the evaluated quotient[2].

```
(1) -> outputGeneral 5
                                                              Type: Void
(2) -> mA := matrix [[ 0.5 ,   1.5 ,   6.6 ,   4.8],  _
                     [ 1.5 ,   6.5 ,  16.2 ,   8.6],  _
                     [ 6.6 ,  16.2 ,  37.6 ,   9.8],  _
                     [ 4.8 ,   8.6 ,   9.8 , -17.1]];

                                                    Type: Matrix Float
(3) -> mB := matrix[[-1 ,  3 ,   4 ,  1],  _
                    [ 3 , 13 ,  16 , 11],  _
                    [ 4 , 16 ,  24 , 18],  _
                    [ 1 , 11 ,  18 , 27]];

                                                  Type: Matrix Integer
(4) -> eVals := nagEigenvalues(mA,mB)
nagman:acknowledging request for f02adf
nagman:connection successful to nags8.nag.co.uk
nagman:receiving results from nags8.nag.co.uk

 ** ABNORMAL EXIT from NAG Library routine F02ADF: IFAIL =      1
 ** NAG soft failure - control returned
nagman:acknowledging request for f02bjf
```

[2]This provides an alternative solution to that adopted in IRENA, described in section 11.1.1, in which the symbols * and % were used to denote infinite and possibly indeterminate eigenvalues, respectively, with the numerators and denominators made available for inspection in separate matrices.

```
nagman:connection successful to nags8.nag.co.uk
nagman:receiving results from nags8.nag.co.uk

   (5)
    18.733159837707458    16.822508301752684
   [-----------------, - -----------------,
    5.3498132860013889    10.873889792812092
    0.29040238760251624 + 1.5317884314528465%i
    ---------------------------------------,
                7.0465137397020676
    0.14557828081124119 - 0.76788324041837752%i
    ---------------------------------------]
                 3.5324067560446304
                 Type: Union(b: List FormalFraction Complex DoubleFloat,...)
```

(In this example there are no entries with components small enough to be questionable, so the output could next be coerced to type `List Complex DoubleFloat`.)

Thus, we have seen that enhanced user interfaces can result when the narrow view of matching interfaces to individual NAG-routines is abandoned in favour of a more user-centered approach, designed to provide a general mathematical capability.

18.5 Symbolic-numeric interaction

Another lesson, which is illustrated by the IRENA project and others, is that, when numeric functionality is incorporated in a symbolic system, there is scope for using both the symbolic and numeric capabilities of the combined system in providing appropriate interfaces. Examples of this can be seen, at an elementary level in the code for **nagpolysolve** in appendix G, where REDUCE functionality was used to recognise distinct cases and to transform the coefficient set from a sparse to a dense representation, and at a considerably more advanced level in the work of Dupée and Davenport, [10].

Conversely, the symbolic component of the combined system may need to be modified, to take account of properties of the numeric methods being used. This may be seen in the interfaces provided for **F02BJF** in both REDUCE and Axiom, discussed in sections 11.1.1 and 18.4

respectively. In the former, representations for infinite and potentially indeterminate numeric quotient values were introduced in the output matrix of eigenvalues; in the latter, a new Axiom type was introduced, to allow the quotients to be displayed without simplification.

In general, it is necessary to apply both numeric and symbolic expertise to the production of such interfaces.

18.6 Conclusion

Although the IRENA project was rather overambitious in its scope, it has had several useful outcomes.

For NAG, as well as developing in-house expertise in symbolic systems generally and symbolic-numeric interfaces in particular, it revealed a number of areas in both software and documentation which could be enhanced – the former, particularly by improvements in the consistency of interfaces, evidenced in the Fortran 90 Library, the latter by making additional aspects of the descriptions of routines explicit in the structure of the documentation.

More concretely, data and code from the IRENA project were reused, either directly or in translated form, in a commercial NAG product, the Axiom-NAG link, and have the potential for further use in this and future products.

From the users' point of view, IRENA represents a system which is significantly easier to use than the Fortran routines on which it is built, with considerable simplification having been achieved both in the software itself and its documentation. This simplification has already begun to be reflected elsewhere, in the new interfaces produced for the Axiom-NAG link.

The IRENA project has demonstrated that a considerable improvement in the ease of use of numerical software is possible, through the careful definition of user interfaces which take account of the mathematical structure of the problems being addressed and through adherence to the design objectives of informativeness, regularity, orthogonality and minimality in their parameterisation.

Part IV

Appendices

Appendix A

Code size and authorage

A.1 Code attribution

The table overleaf shows the contributions to IRENA, in terms of source code, of the principal authors, Dewar (MCD) and Richardson (MGR).

Most items have been classified as having a single principal author, although the other author may also have contributed code – for example, many of Richardson's enhancements to IRENA-0, described in chapter 9, took the form of additional code in Dewar's `interface.red` RLISP source. Since some ASP template files contain significant amounts of code from both authors, these are ascribed to "MGR + MCD". Contributions to the code by Atta (NA) and McGettrick (MMcG) (see sections 9.2.6 and 9.4) are also included in the table.

In the "language" column, "RLISP" is the REDUCE system language – see chapters 16 and 18 of [14]; "PSL" is Portable Standard Lisp, the language originally underlying RLISP – see [36]; "GENTRAN" is Gates's extension to REDUCE which provides automatic code generation in a variety of languages – see [12] and [13]; "IRENA" represents code written in the REDUCE "algebraic" (that is, user level) language, with the IRENA authors' extensions; "jazz" and "defaults" are the purpose-built languages defined for coding the correspondingly named IRENA files.

The defaults and jazzing languages are introduced in chapter 7 and their usage is exemplified in chapters 10 and 11. The problem of constructing ASP templates is introduced in section 3.2 and discussed further in various places in this thesis.

Hand coded material distributed with IRENA-1				
Component	Source language	Principal author(s)	Number of items	Size in bytes
System source	RLISP	MCD	22	406543
		MGR	3	124395
	PSL	MCD	2	8657
	C	MCD	1	1589
		NA + MMcG	1	4675
Jazzing	jazz	MGR	160	119604
Tests	IRENA	MGR	195	96728
Defaults	defaults	MGR	129	44504
ASP templates	GENTRAN	MCD	18	18687
		MGR	3	3429
		MCD + MGR	8	8415

Undistributed source material for IRENA-1				
Component	Source language	Principal author(s)	Number of items	Size in bytes
Setup programs	C	MCD	4	148097
		NA	1	40958
		MGR	1	7280
Jackets	Fortran	MGR	15	9962

Material for routines not included in IRENA-1				
Component	Source language	Principal author(s)	Number of items	Size in bytes
Jazzing	jazz	MGR	122	44064
Tests	IRENA	MGR	115	37441
Defaults	defaults	MGR	151	28869
Jackets	Fortran	MGR	8	6189

Table A.1: Principal contributions to IRENA code

The material described as "setup programs" includes the code to generate specfiles from NAG documentation, and infofiles and GENTRAN templates from specfiles, described in chapter 3, the code to generate skeleton default files, described in section 9.2.6 (but little used, in practice) – see figure 3-1 for all of these – and the code for generating jazz and default fragments appropriate to NAG parameters which function as switches, described in section 15.2.

The use of Fortran jackets is described in chapter 12.

Ascribing each file to its principal author and the jointly authored ASP templates to the two principal authors in the same proportions as the individually authored templates gives an approximate measure of total code contribution as 590,715 bytes by Dewar and 543,965 bytes by Richardson.

A.2 Distributed size of IRENA-1

The total size of REDUCE 3.5, with IRENA loaded, is 5.7 Mbytes, of which about 0.9 Mbyte represents compiled IRENA code. Interpreted elements of the IRENA system (jazz and defaults files, C and Fortran GENTRAN templates, and infofiles) distributed with IRENA-1 total 0.76 Mbyte, of which 89% was generated automatically, as described in chapter 3.

Other major components of the IRENA-1 distribution were the NAG Foundation (Fortran) Library, of 0.63 Mbyte, and the NAG Fortran 90 compiler, of 1.07 Mbytes.

As with REDUCE, system source, documentation source, test programs and results for IRENA were also distributed with the system, giving a total size for the distribution (including REDUCE material) of about 16 Mbytes.

Appendix B

NAG, Naglink and IRENA parameterisations of a specimen routine

As many of the routines in the NAG Foundation Library, on which IRENA-1 is based, had not been introduced at the time of the Naglink project, there is a rather limited choice of routines which can be used to contrast the parameterisations in these systems. However, the NAG (Fortran 77) routine `F02AXF`, which calculates the eigenvalues and eigenvectors of a Hermitian matrix, serves to illustrate some of the contrasts, without having an unmanageably large number of parameters.

B.1 Example calls

B.1.1 NAG (Fortran 77) call

```
      DOUBLE PRECISION AR(4,4), AI(4,4), VR(4,4), VI(4,4),
     +                 R(4), WK1(4), WK2(4), WK3(4)
      N = 4
      READ(NIN,*)  ((AR(I,J), AI(I,J), J=1,N), I=1,N)
*       Where the data file contains:
*       0.50  0.00  0.00  0.00  0.00  0.00  0.00  0.00
*       0.00  0.00  0.50  0.00  0.00  0.00  0.00  0.00
*       1.84 -1.38  1.12 -0.84  0.50  0.00  0.00  0.00
*       2.08  1.56 -0.56 -0.42  0.00  0.00  0.50  0.00
      IFAIL = 1
      CALL F02AXF(AR,N,AI,N,N,R,VR,N,VI,N,WK1,WK2,WK3,IFAIL)
```

B.1.2 Naglink call

```
ar : ([  0.50,  0.0 , 0.0 ,  0.0  ],
      [  0.0 ,  0.50, 0.0 ,  0.0  ],
      [  1.84,  1.12, 0.5 ,  0.0  ],
      [  2.08, -0.56, 0.0 ,  0.50 ]);

ai : ([  0.0 ,  0.0 , 0.0 ,  0.0  ],
      [  0.0 ,  0.0 , 0.0 ,  0.0  ],
      [ -1.38, -0.84, 0.0 ,  0.0  ],
      [  1.56, -0.42, 0.0 ,  0.0  ]);

result : f02axf(ar,ai);
```

B.1.3 IRENA call

```
% The layout here is cosmetic.

herm!-mat a {{  0.5,  0.0,  1.84 +1.38*i,  2.08 -1.56*i },
             {        0.5,  1.12 +0.84*i, -0.56 +0.42*i },
             {              0.5         ,  0.0          },
             {                             0.5          }}$

f02axf()$
```

B.2 Matrix defining parameters

To define the input matrix in the NAG call requires the first five parameters of the call which specify, respectively, the real parts of the lower triangle of the matrix (in a rectangular array), the first dimension of that array, the imaginary parts of the lower triangle (in a rectangular array), the first dimension of that array and the order of the matrix.

In Naglink, this is reduced to two matrices, which retain the NAG names **AR** and **AI**. The Naglink documentation did not specify what these represented: the user is left to infer that they are square matrices whose lower triangles contain the appropriate entries.

In IRENA this is further reduced to a single, complex matrix, described in the documentation as a Hermitian matrix; this is an IRENA datatype, which may be specified by either its upper or lower triangle, at the choice of the user.

B.3 Accessing the results

In the NAG call, the eigenvalues are returned in the one-dimensional array **R**; the real and imaginary parts of the eigenvectors are returned in the two-dimensional arrays **VR** and **VI** (whose leading dimensions must appear after the array names in the routine call).

Naglink returns a list, the documentation of whose components retains the names **R**, **VR** and **VI**, describing the first as "a list of real numbers" and the others as "matrices with real entries". The individual entries may be accessed using the Macsyma **get** function.

IRENA displays a list:

```
{EIGENVALUES,NORMALIZED_EIGENVECTORS_AS_COLUMNS}
```

naming its output objects, which exist in the REDUCE environment. These may be most conveniently accessed as `@1` and `@2`, respectively; unless the user has specified otherwise, the display of the list is preceded by general instructions explaining this mode of access.

(In the notional IRENA-0, the `@` output feature was not available and the output objects would have had shorter and rather less mnemonic names.)

B.4 Workspace parameters

The parameters `WK1`, `WK2` and `WK3` are required by the NAG routine for workspace – both Naglink and IRENA handle this automatically.

B.5 Other parameters

The remaining NAG parameter `IFAIL` indicates, on input, whether or not errors should cause the program to terminate; on output, its value either indicates that the run was error free or indexes the type of error which occurred.

In Naglink, the system is retained, with the input functionality being carried by the Macsyma "pre-variable" `softfail` and the output functionality by the "post-variable" `ifail`. Both of these quantities are present as Macsyma variables.

In IRENA, errors recognised by the NAG routine are never allowed to terminate the program, as the diagnostic information would then be lost. If a non-zero value is returned (indicating an error) IRENA prints a diagnostic, rather than returning an unexplained index to the user.

Appendix C

Example IRENA-function description

The command `print-precision`, *which occurs in the example in section 5, was provided in REDUCE by Professor J. H. Davenport, early in the IRENA project, in response to a request from the present author for a mechanism to allow the output of numeric results with a precision matching their expected accuracy.*

c02aff

1 Purpose

Finds all the roots of the complex polynomial equation $P(z) = a_0 z^n + a_1 z^{n-1} + \ldots + a_{n-1} z + a_n = 0$, using a variant of Laguerre's method.

2 Essential Input Parameters

1 `Coefficients (highest order first)` (*alias* `coefficients, coefs`) the coefficients a_i, stored in the order a_0 to a_n with $a_0 \neq 0$.

3 Optional Input Parameters

1 `Scale the polynomial` (*alias* `scale`) indicates whether the polynomial is to be scaled to avoid overflow/underflow. Possible values `y`, `n` may be represented as keywords:

```
scaled     s
unscaled   u
```

The default value is `scaled`.

4 Output Parameters

1 `Zeroes` the roots of the polynomial.

5 Example

To find the roots of the polynomial $a_0z^5+a_1z^4+a_2z^3+a_3z^2+a_4z+a_5 = 0$, where $a_0 = (5.0+6.0i)$, $a_1 = (30.0 + 20.0i)$, $a_2 = -(0.2 + 6.0i)$, $a_3 = (50.0 + 100000.0i)$, $a_4 = -(2.0 - 40.0i)$ and $a_5 = (10.0 + 1.0i)$.

```
% c02aff example

on rounded$

print!-precision 5$

c02aff(vec coefficients {  5   +      6*i,
                          30   +     20*i,
                          -0.2 -      6*i,
                          50   + 100000*i,
                          -2   +     40*i,
                          10   +        i})$

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

{ZEROES}
```

```
zeroes;

[   -24.328 - 4.8555*I   ]
[                        ]
[   5.2487 + 22.736*I    ]
[                        ]
[   14.653 - 16.569*I    ]
[                        ]
[-0.0069264 - 0.0074434*I]
[                        ]
[0.0065264 + 0.0074232*I ]

print!-precision(-1)$

off rounded$
```

Appendix D

Brief descriptions of jazzing commands

Where not otherwise indicated, the commands marked "built-in" were provided by Dewar as part of IRENA-0; those marked "jazz-function" or "output-function" (see sections 7.3.1 and 7.3.2) were added by the present author. Commands which have become redundant, due to the withdrawal of the NAG routines to which they applied, are excluded.

The commands in each section are ordered mainly according to decreasing frequency of usage, with related commands grouped together in the input jazzing section. The precise usage frequencies may be found in tables 15.1 and 15.3.

For convenience in locating individual commands, an index of these is given overleaf, in table D.1.

command	page	command	page
Append	225	Output-rectangle	223
Build-rectangle	222	Out-tuple	224
Calculate	221	Phased-prompt	213
Cmat2ivec	217	Precedence	220
Cmat2rvec	217	Prompt-alias	211
Cmplxquots	223	Qkeyword	212
Column-mat	216	Ragged-in	214
Complex-in	214	Raggedlengths-1	218
Complex-out	220	Ragged-out	221
Concatenate	213	Raggedvalues	217
Cond-out	221	Rect2scalar	215
Cuhessandlow	225	Rectangle	213
Diagonal	217	Reshape-output	222
Elements	223	Row-mat	216
Fill-knots	215	Rowmat2vec	215
Fort-dims	212	Sbandlengths	216
Gridfirst	213	Sbandvalues	216
Gridsecond	214	Scalar	212
Hi-d-dims	219	Set-type	211
Hi-d-im-vals	218	Silent-alias	211
Hi-d-re-vals	218	Sparsecolumn	216
I2o	220	Sparserow	215
Interpret	223	Sparsevalues	216
Interps	225	Sumraggedlengths	218
Key-alias	211	Sup+dinv2up	225
Keyword	212	Template	213
Local	211	Trim-matrix	217
Lower	224	Trim-vector	217
Mat2vec	215	Tuples1	214
Matels2list	224	Tuples2	214
Matoverlay	224	Tuples3	214
Maxraggedlengths	218	Unpack	218
Message	220	Up1diagandlow	221
Newscalar	215	Upandlow1diag	225
Out-dims	220	Upandslow	223
Output	219	Vec2rowmat	221
Outputconj	224	Vector	212
Output-order	219		

Table D.1: Index of jazz commands

D.1 Input jazzing commands

Unless otherwise qualified, "vector" (or "general vector") below means an object which may be an IRENA vector or a row or column vector represented as a REDUCE matrix.

Prompt-alias

Built-in command.

Specifies the principal alternative IRENA name for a NAG or IRENA parameter. The name specified is used in prompting the user when **promptval** is **on**.

Key-alias

Built-in command.

Specifies an alternative IRENA name for a NAG or IRENA parameter.

Silent-alias

Built-in command, added by the present author.

Specifies an additional alias for a parameter, which will not be separately documented. Used to allow singular and plural forms to be used interchangeably, where appropriate.

Local

Built-in command.

Specifies a "very local constant" – that is, a name which may be used as an IRENA value to represent a particular value of a NAG parameter. Commonly used to allow * to represent the value or values chosen in a particular NAG routine to mean "unbounded".

Set-type

Built-in command, added by the present author.

Overrides the default type, generated by IRENA for an input object and used in prompting.

Scalar

Built-in command.

Introduces a scalar quantity which mimics an additional NAG parameter. Also used for communication between and within the jazz and defaults files.

Vector

Built-in command.

Similar to `scalar` but introducing a non-scalar quantity.

Fort-dims

Built-in command, added by the present author.

Specifies the dimensions to be used for an array in the generated Fortran program: these may depend on the values of other parameters. Used where the dimensions could not be generated automatically or where the automatically generated values would be inappropriate – for instance, where a NAG input array (which would normally be given dimensions based on the IRENA object from which it is constructed) is specified as being of a certain minimum size, greater than that necessarily implied by the data which it contains.

Keyword

Built-in command.

Specifies an IRENA keyword to represent a particular value of a NAG parameter. The possible keywords will be used in generating an IRENA prompt for "one of the following:" if it is necessary to prompt for the parameter in question.

Qkeyword

Built-in command, added by the present author.

Specifies an IRENA keyword, which represents a particular value of a NAG parameter but which will not be used in generating a prompt. Used, for example, where it is more appropriate to ask a "Yes or no?" question as the prompt.

Rectangle

Built-in command.

Defines an IRENA "rectangle" to represent a pair of NAG scalars or one-dimensional arrays.

Concatenate

Jazz-function.

Several objects – which may be vectors, IRENA scalar parameters, IRENA local scalars, constants (including * meaning "unset") or repeated constants – are concatenated to form a NAG one-dimensional array.

Template

Built-in command.

Indicates that a different Fortran routine to that which shares the name of the IRENA-function should be used to generated the Fortran code. So called since it causes the specified name to be used in selecting the C and Fortran templates.

Phased-prompt

Built-in command, added by the present author.

Specifies a two-level prompting mechanism, in which the response to the first prompt either sets a "special" value (often **unset**) for a NAG parameter or triggers a second prompt to elicit a "normal" value.

Gridfirst

Jazz-function.

A "grid" is used to specify a rectangular grid of points as a pair, each element of which may be a single value or a list of values, with gridpoints occurring at each combination of values from the two elements. `Gridfirst` converts the first element of the pair into a one-dimensional NAG array, optionally padded at each end with a specified number of copies of a specified entry.

Gridsecond

Jazz-function.

Similar to `gridfirst`, processing the second element of the pair.

Ragged-in

Jazz-function.

Unpacks a ragged array to produce a specified member of a set of NAG scalars and one-dimensional arrays. Used in passing details of a matrix factorisation, generated by an associated routine, to a linear system solver.

Tuples1

Jazz-function.

Takes a list of lists, each representing an n-tuple, and generates a one-dimensional NAG array consisting of the tuples' first elements.

Tuples2

Jazz-function.

Similar to `tuples1` but extracting the second elements.

Tuples3

Jazz-function.

Similar to `tuples1` and `tuples2` but extracting the third elements.

Complex-in

Built-in command.

Takes an IRENA complex-valued vector or matrix and generates a pair of real-valued NAG arrays or a single real-valued NAG array with an extra dimension of 2.

Mat2vec

Jazz-function.

Takes a matrix and generates a one-dimensional NAG array with elements in Fortran (that is, column major) order.

Rowmat2vec

Jazz-function.

Takes a matrix and generates a one-dimensional NAG array with elements in row major order.

Fill-knots

Jazz-function.

Takes a vector and generates a one-dimensional NAG array, adding three repetitions of the first and last elements to provide the form required for a set of "knots" by various NAG spline fitting routines.

Newscalar

Built-in command.

Specifies an IRENA scalar from which a NAG scalar parameter may be calculated, allowing, for example, the user to provide `N` as a means of specifying the NAG parameter `NPLUS1`.

Rect2scalar

Jazz-function.

Extracts a NAG scalar from a specified position in an IRENA rectangle.

Sparserow

Jazz-function.

Takes an Irena sparse or symmetric sparse matrix and extracts the row address vector (of the upper triangle, in the latter case) to provide a one-dimensional NAG array.

Sparsecolumn

Jazz-function.

Similar to **sparserow**, this extracts the column address vector.

Sparsevalues

Jazz-function.

Similar to **sparserow** and **sparsecolumn**, this extracts the values of the matrix elements into a one-dimensional NAG array.

Column-mat

Jazz-function.

Originally written by Dewar to illustrate the use of jazz-functions, it took several IRENA vectors and used them as the columns of a NAG matrix. Later rewritten by Richardson to accept general vectors as input.

Row-mat

Jazz-function.

Similar to **column-mat**, it constructs the rows of a NAG matrix from vectors.

Sbandvalues

Jazz-function.

Takes a symmetric band matrix and generates a one-dimensional NAG array containing the entries within the band, in row major order.

Sbandlengths

Jazz-function.

Takes a symmetric band matrix and generates a one-dimensional NAG array containing the lengths of the rows within the band. Optionally also generates an IRENA output parameter with a ***noname*** prefix, for later re-input to other routines.

Cmat2rvec

Jazz-function.

Takes a complex matrix and generates a one-dimensional real-valued NAG array consisting of the real parts of its entries, with elements in column major order.

Cmat2ivec

Jazz-function.

Similar to `cmat2rvec`, producing an array of imaginary parts.

Diagonal

Jazz-function.

Creates a one-dimensional NAG array from a diagonal (or sub- or super-diagonal) of a REDUCE or IRENA matrix.

Trim-vector

Jazz-function.

Takes an IRENA vector and generates a one-dimensional NAG array, consisting of elements in a specified range. Used for sorting routines, to isolate the part of a vector to be processed; this is later reinserted in the original vector. Duplicates Fortran's facility for processing part of an array *in situ*.

Trim-matrix

Jazz-function.

Similar to `trim-vector`, processing a matrix and generating a two-dimensional array.

Raggedvalues

Jazz-function.

Takes a ragged array and generates a one-dimensional NAG array containing the same entries.

Raggedlengths-1

Jazz-function.

Takes a ragged array and generates a one-dimensional NAG array whose entries are one less than the row lengths.

Maxraggedlengths

Jazz-function.

Takes a ragged array and generates a NAG scalar equal to the maximum of the row lengths, plus an optional adjustment, if specified.

Sumraggedlengths

Jazz-function.

Takes a ragged array and generates a NAG scalar equal to the sum of the row lengths, plus an optional adjustment, if specified.

Unpack

Built-in command.

Specifies an IRENA vector to represent a set of NAG scalar parameters.

Hi-d-re-vals

Jazz-function.

Takes a "hi-d" structure (a structure of nested lists, representing a multi-dimensional object) and generates a one-dimensional NAG array of the real parts of its entries, with elements in column major order.

Hi-d-im-vals

Jazz-function.

Similar to `hi-d-re-vals`, taking the imaginary parts of the entries.

Hi-d-dims

Jazz-function.

Generates a one-dimensional NAG array giving the dimensions of a "hi-d" structure.

D.2 Output jazzing commands

The output-function mechanism provided by Dewar requires all output-functions to process a list specifying at least one NAG output parameter; the parameters in the list are automatically removed from the IRENA output list, so that they must, occasionally, be actively reinstated as IRENA output parameters.

Other NAG input and output parameters may be accessed by an output-function in defining IRENA output parameters, any number of which can be created by one output-function. Where NAG input parameters are allowed, this implicitly includes "quasi-NAG" parameters defined by the input jazzing commands `scalar` and `vector`.

Unless otherwise indicated, the various objects built by the commands below are structures, available at the REDUCE level, whose names are displayed in the IRENA-function's output list.

Output

Built-in command.

Dewar's principal output jazzing command, designed mainly for processing output arrays. It builds IRENA output objects from elements, partial columns and rectangular subarrays of arrays. Additionally, it provides (as a separate instance of the command) a renaming facility including a `case` construct, originally controlled by input parameter values only but extended by the present author to allow control by output values.

Output-order

Built-in command.

Specifies the order in which the names of possible output objects should be displayed in the output list; objects not explicitly named here appear at the end of the output list. It was

modified by the present author to inhibit the display of names listed in the command which do not correspond to actual output objects – for example, as the result of conditional renaming with an `output` command – and of names which begin with the string `*noname*`.

Precedence

Built-in command.

Imposes a jazz processing order on a set of the NAG output parameters.

I2o

Built-in command.

Builds an IRENA output object from a NAG input parameter.

Out-dims

Built-in command.

Converts a one-dimensional NAG output array, assumed to represent a matrix in column major order, into a matrix of the specified dimensions.

Message

Output-function.

Creates an IRENA output object containing a fixed text string if a specified criterion is true. Optionally, produces an IRENA output object corresponding to the specified NAG scalar parameter.

Complex-out

Built-in command.

Converts a pair of NAG real arrays or scalars or an $n \times 2$ array into a complex IRENA output object. Allows subarrays to be processed.

Ragged-out

Output-function.

Collects various NAG arrays, subarrays and scalars to form a ragged array, to which a conditional transformation may be applied, as an IRENA output object. (The only transformation used at present is to conditionally reverse the order of the lists in a two-list ragged array.)

Vec2rowmat

Output-function.

Uses the values, stored in row major order, in a NAG one-dimensional array to produce a REDUCE matrix of specified dimensions as an IRENA output object.

Calculate

Output-function.

Obeys an arbitrary Lisp program. Usually used to perform simple arithmetic, possibly conditionally, on NAG scalar parameter values.

Cond-out

Built-in command.

Designed to handle the case where different NAG output parameters may hold a particular structure, depending on the value or relative values of other parameters, it also has facilities mimicking various other output commands, enabling it to join fragments of various NAG structures, transformed in a variety of ways, to produce the eventual IRENA output structure.

Up1diagandlow

Output-function.

Transforms the strict upper triangle of a two-dimensional NAG array into an upper triangular REDUCE matrix in which each entry on the diagonal is 1 and transforms the lower triangle of the NAG array into a lower triangular REDUCE matrix.

Build-rectangle

Output-function.

Conditionally builds an IRENA rectangle as an output object, constructing it either from two one-dimensional NAG arrays or from two lists of scalars, each element of which may be a NAG input or output parameter or a REDUCE global (set up separately as a ***noname*** object, using **calculate**).

Reshape-output

Output-function.

Takes the elements of sections of any number of one- or two-dimensional NAG arrays and from these builds a matrix of specified dimensions as an IRENA output structure.

This command was written as a "mock-up" for a possible second generation general IRENA output jazzing facility and, as such, simulates a "key and value" syntax by using dotted key and value pairs in the Lisp list which is its principal control; in one case, a freestanding keyword is allowed, represented by the atom **paired**.

Possible entries in the Lisp list are given in table D.2. The value sections referring to the input arrays may be repeated, in which case each instance refers to one of a list of NAG arrays (or a pair of **paired** arrays). Use of a single instance indicates that the same value should be applied to all of the NAG arrays. Default settings are provided and are invoked by using the value **nil**.

Key	Value signifies
iname	the name of the IRENA output matrix
rowtrim	the first and last rows required from the input array
coltrim	the first and last columns required from the input array
dims	the dimensions of the output matrix
majorin	the elements of the input array are to be taken in row or column major order
majorout	the output matrix is to be built in row or column major order
shapein	the shape of the required section of the input array; only **full** is currently allowed
shapeout	the shape of the required output matrix to be built from the array elements
fill	an entry used as fill for the diagonal or throughout the output matrix
paired	if **'paired** present, input arrays are to be processed as (real, imaginary) pairs

Table D.2: Keys in the **reshape-output** control list

Interpret

Output-function.

Builds an IRENA output matrix, corresponding to a section of a NAG two-dimensional array, with numeric values replaced by text strings according to a specified key. Optionally retains the equivalent of the original NAG matrix as an IRENA output object.

Upandslow

Output-function.

Unpacks a NAG two-dimensional array into upper triangular and strict lower triangular REDUCE matrices.

Output-rectangle

Built-in command.

Converts a pair of NAG output scalars or one-dimensional arrays into an IRENA output rectangle.

Cmplxquots

Output-function.

Takes three equal length NAG real one-dimensional arrays, representing the real and imaginary parts of the numerators and the real denominators of a vector of extended complex numbers and constructs that vector as an IRENA output object. The point at infinity is represented by * and possibly indeterminate values appear as %. (Any component less than 10^{-10} is considered to possibly represent zero.) If there are infinities or indeterminate values, a warning message is also returned.

Elements

Output-function.

Extracts from a NAG one-dimensional array a number of IRENA output scalars or from a two-dimensional array a number of single column matrices. Optionally, all or an initial segment of the NAG array may also be made into an IRENA output object.

Lower

Built-in command.

Extracts the lower triangle of a matrix represented as a square, two-dimensional NAG array.

Matels2list

Output-function.

Builds a list, containing specified elements from a NAG two-dimensional array, as an IRENA output object.

Matoverlay

Output-function.

Replaces a specified section of one NAG array with another, to build an IRENA output matrix. (Used to mimic Fortran *in situ* sorting.)

Outputconj

Output-function.

Takes as input a one-dimensional, real NAG "indicator" array and a square, two-dimensional real array, the columns of which represent either real column vectors (for zero elements in the indicator) or the real and imaginary parts of conjugate vectors, and builds the corresponding complex matrix, as an IRENA output structure.

Out-tuple

Output-function.

Builds an IRENA output object consisting of a list of tuples from a set of equal-length NAG one-dimensional arrays. Provides compatibility between the output of a spline coefficient generating IRENA-function and the input of spline evaluation IRENA-functions.

Upandlow1diag

Output-function.

Transforms the upper triangle of a two-dimensional NAG array into an upper triangular REDUCE matrix and transforms the strict lower triangle into a lower triangular REDUCE matrix in which each entry on the diagonal is `1`.

Append

Built-in command.

Transforms a number of NAG scalar parameters into a single column matrix.

Sup+dinv2up

Output-function.

Builds a REDUCE upper triangular matrix from the strict upper triangle, stored as part of a NAG two-dimensional array, and the diagonal, the inverses of whose elements are stored in a NAG one-dimensional array.

Cuhessandlow

Output-function.

Takes two NAG two-dimensional arrays, the upper trapezia of which represent the real and imaginary parts of a complex upper Hessenberg matrix and the remaining lower triangles the real and imaginary parts of a complex lower triangular matrix, and builds these matrices as REDUCE objects.

Interps

Output-function.

Written by Dewar to illustrate the use of output-functions, it transforms a one-dimensional NAG array, which represents an upper triangular array in row major order, into an upper triangular matrix.

Appendix E

Data used in the complexity analysis

Routine	Parameter counts									Jazz file line count	Defaults file line count
	in-put	out-put *	i/o	work-space	dummy	function		subroutine			
						main	2nd level	main	2nd level		
a00aaf	0	0	0	0	0	0	0	0	0	0	0
c02aff	3	1	1	1	0	0	0	0	0	25	11
c02agf	3	1	1	1	0	0	0	0	0	23	11
c05adf	4	1	1	0	0	1	1	0	0	17	7
c05nbf	2	1	3	1	0	0	0	1	4	25	11
c05pbf	3	2	3	1	0	0	0	1	6	27	13
c06eaf	1	0	2	0	0	0	0	0	0	14	9
c06ebf	1	0	2	0	0	0	0	0	0	15	9
c06ecf	1	0	3	0	0	0	0	0	0	15	9
c06ekf	2	0	3	0	0	0	0	0	0	20	9
c06fpf	3	0	3	1	0	0	0	0	0	24	13
c06fqf	3	0	3	1	0	0	0	0	0	24	13
c06frf	3	0	4	1	0	0	0	0	0	32	13
c06fuf	3	0	5	1	0	0	0	0	0	35	15
c06gbf	1	0	2	0	0	0	0	0	0	7	7

* An asterisk under "output" indicates a function (which returns a value through its name).

Routine	Parameter counts									Jazz file line count	Defaults file line count
	in-put	out-put *	i/o	work-space	dummy	function		subroutine			
						main	2nd level	main	2nd level		
c06gcf	1	0	2	0	0	0	0	0	0	7	7
c06gqf	2	0	2	0	0	0	0	0	0	11	9
c06gsf	3	2	1	0	0	0	0	0	0	13	9
d01ajf	6	4	1	0	0	1	1	0	0	51	14
d01akf	6	4	1	0	0	1	1	0	0	51	14
d01alf	8	4	1	0	0	1	1	0	0	57	18
d01amf	6	4	1	0	0	1	1	0	0	51	22
d01anf	8	4	1	0	0	1	1	0	0	60	16
d01apf	9	4	1	0	0	1	1	0	0	68	29
d01aqf	7	4	1	0	0	1	1	0	0	57	14
d01asf	7	7	1	1	0	1	1	0	0	51	17
d01bbf	4	2	1	0	0	0	0	1	0	28	24
d01fcf	6	2	2	1	0	1	2	0	0	32	16
d01gaf	3	2	1	0	0	0	0	0	0	13	7
d01gbf	6	2	3	0	0	1	2	0	0	54	29
d02bbf	3	0	4	1	0	0	0	2	5	40	11
d02bhf	4	0	4	1	0	1	2	1	3	47	11
d02cjf	4	0	3	1	0	1	2	2	5	40	13
d02ejf	4	0	4	1	0	1	2	3	8	45	15
d02gaf	9	1	3	2	0	0	0	1	3	85	30
d02gbf	7	1	6	2	0	0	0	2	4	89	30
d02kef	5	0	6	0	0	0	0	4	18	161	28
d02raf	10	1	5	2	0	0	0	6	31	111	51
e01baf	5	2	1	1	0	0	0	0	0	17	9
e01bef	3	1	1	0	0	0	0	0	0	13	5
e01bff	5	1	2	0	0	0	0	0	0	29	13
e01bgf	5	2	2	0	0	0	0	0	0	33	13
e01bhf	6	1	1	0	0	0	0	0	0	27	9
e01daf	5	5	1	1	0	0	0	0	0	27	7

Routine	Parameter counts									Jazz file line count	Defaults file line count
	in-put	out-put *	i/o	work-space	dummy	function		subroutine			
						main	2nd level	main	2nd level		
e01saf	4	2	1	0	0	0	0	0	0	17	7
e01sbf	10	1	1	0	0	0	0	0	0	28	16
e01sef	6	2	3	1	0	0	0	0	0	48	13
e01sff	10	1	1	0	0	0	0	0	0	30	18
e02adf	6	2	1	2	0	0	0	0	0	31	17
e02aef	3	1	1	0	0	0	0	0	0	25	19
e02agf	15	4	1	1	0	0	0	0	0	62	25
e02ahf	8	2	1	0	0	0	0	0	0	26	15
e02ajf	9	1	1	0	0	0	0	0	0	25	17
e02akf	7	1	1	0	0	0	0	0	0	19	11
e02baf	5	2	2	2	0	0	0	0	0	25	14
e02bbf	4	1	1	0	0	0	0	0	0	13	7
e02bcf	5	1	1	0	0	0	0	0	0	15	9
e02bdf	3	1	1	0	0	0	0	0	0	11	7
e02bef	8	2	5	0	0	0	0	0	0	43	31
e02daf	11	5	8	3	0	0	0	0	0	102	40
e02dcf	11	2	7	0	0	0	0	0	0	60	35
e02ddf	11	4	6	0	0	0	0	0	0	58	54
e02def	8	1	1	2	0	0	0	0	0	17	11
e02dff	11	1	1	2	0	0	0	0	0	17	19
e02gaf	4	4	3	1	0	0	0	0	0	26	13
e02zaf	9	1	1	1	0	0	0	0	0	17	15
e04dgf	1	3	2	4	0	0	0	1	8	27	7
e04fdf	4	2	2	1	0	0	0	1	4	42	15
e04gcf	4	2	2	1	0	0	0	1	6	42	15
e04jaf	4	1	4	2	0	0	0	1	3	27	21
e04mbf	13	3	2	2	0	0	0	0	0	112	42
e04naf	20	3	3	2	0	0	0	1	7	138	66
e04ucf	11	4	6	4	0	0	0	2	19	127	43

Routine	Parameter counts									Jazz file line count	Defaults file line count
	in-put	out-put *	i/o	work-space	dummy	function		subroutine			
						main	2nd level	main	2nd level		
e04ycf	6	1	2	1	0	0	0	0	0	39	17
f01brf	8	3	4	1	0	0	0	0	0	146	39
f01bsf	11	2	2	1	0	0	0	0	0	69	27
f01maf	5	3	6	1	0	0	0	0	0	95	39
f01mcf	4	2	1	0	0	0	0	0	0	13	9
f01qcf	3	1	2	0	0	0	0	0	0	11	11
f01qdf	9	0	2	1	0	0	0	0	0	46	61
f01qef	6	0	2	1	0	0	0	0	0	35	54
f01rcf	3	1	2	0	0	0	0	0	0	11	11
f01rdf	9	0	2	1	0	0	0	0	0	46	60
f01ref	6	0	2	1	0	0	0	0	0	35	54
f02aaf	2	1	2	1	0	0	0	0	0	9	9
f02abf	4	2	1	1	0	0	0	0	0	11	11
f02adf	3	1	3	1	0	0	0	0	0	13	11
f02aef	4	2	3	2	0	0	0	0	0	17	13
f02aff	2	3	2	0	0	0	0	0	0	11	9
f02agf	4	5	2	0	0	0	0	0	0	14	13
f02ajf	3	2	3	1	0	0	0	0	0	11	11
f02akf	5	4	3	1	0	0	0	0	0	15	15
f02awf	3	1	3	3	0	0	0	0	0	13	11
f02axf	7	3	1	3	0	0	0	0	0	13	15
f02bbf	6	4	2	6	0	0	0	0	0	30	13
f02bjf	6	5	3	0	0	0	0	0	0	56	19
f02fjf	8	1	4	3	0	1	8	2	15	93	41
f02wef	9	4	3	0	0	0	0	0	0	88	53
f02xef	9	4	3	1	0	0	0	0	0	93	46
f04adf	6	1	2	1	0	0	0	0	0	29	15
f04arf	3	1	2	1	0	0	0	0	0	25	9
f04asf	3	1	2	2	0	0	0	0	0	25	9

Routine	Parameter counts									Jazz file line count	Defaults file line count
	in-put	out-put *	i/o	work-space	dummy	function		subroutine			
						main	2nd level	main	2nd level		
f04atf	5	2	1	2	0	0	0	0	0	29	11
f04axf	7	1	1	1	0	0	0	0	0	48	11
f04faf	2	0	4	0	0	0	0	0	0	53	9
f04jgf	5	4	3	0	0	0	0	0	0	46	15
f04maf	10	1	4	0	0	0	0	0	0	63	31
f04mbf	8	7	2	3	0	0	0	2	16	100	30
f04mcf	9	1	4	0	0	0	0	0	0	47	28
f04qaf	10	9	2	3	0	0	0	1	9	105	23
f07adf	3	2	1	0	0	0	0	0	0	13	11
f07aef	7	1	1	0	0	0	0	0	0	35	17
f07fdf	3	1	1	0	0	0	0	0	0	16	11
f07fef	6	1	1	0	0	0	0	0	0	27	13
m01caf	3	0	2	0	0	0	0	0	0	25	11
m01daf	4	1	1	0	0	0	0	0	0	29	11
m01def	7	1	1	0	0	0	0	0	0	33	21
m01djf	7	1	1	0	0	0	0	0	0	29	17
m01eaf	10	1	2	0	0	0	0	0	0	59	31
m01zaf	2	0	2	0	0	0	0	0	0	15	7
s01eaf	1	* 0	1	0	0	0	0	0	0	5	0
s13aaf	1	* 0	1	0	0	0	0	0	0	5	0
s13acf	1	* 0	1	0	0	0	0	0	0	5	0
s13adf	1	* 0	1	0	0	0	0	0	0	5	0
s14aaf	1	* 0	1	0	0	0	0	0	0	5	0
s14abf	1	* 0	1	0	0	0	0	0	0	5	0
s14baf	3	2	1	0	0	0	0	0	0	11	5
s15adf	1	* 0	1	0	0	0	0	0	0	5	0
s15aef	1	* 0	1	0	0	0	0	0	0	5	0
s17acf	1	* 0	1	0	0	0	0	0	0	5	0
s17adf	1	* 0	1	0	0	0	0	0	0	5	0

Routine	Parameter counts									Jazz file line count	Defaults file line count
	in-put	out-put *	i/o	work-space	dummy	function		subroutine			
						main	2nd level	main	2nd level		
s17aef	1	* 0	1	0	0	0	0	0	0	5	0
s17aff	1	* 0	1	0	0	0	0	0	0	5	0
s17agf	1	* 0	1	0	0	0	0	0	0	5	0
s17ahf	1	* 0	1	0	0	0	0	0	0	5	0
s17ajf	1	* 0	1	0	0	0	0	0	0	5	0
s17akf	1	* 0	1	0	0	0	0	0	0	5	0
s17dcf	4	2	1	1	0	0	0	0	0	19	9
s17def	4	2	1	0	0	0	0	0	0	19	9
s17dgf	3	2	1	0	0	0	0	0	0	18	7
s17dhf	3	1	1	0	0	0	0	0	0	14	7
s17dlf	5	2	1	0	0	0	0	0	0	27	9
s18acf	1	* 0	1	0	0	0	0	0	0	5	0
s18adf	1	* 0	1	0	0	0	0	0	0	5	0
s18aef	1	* 0	1	0	0	0	0	0	0	5	0
s18aff	1	* 0	1	0	0	0	0	0	0	5	0
s18dcf	4	2	1	0	0	0	0	0	0	19	7
s18def	4	2	1	0	0	0	0	0	0	19	7
s19aaf	1	* 0	1	0	0	0	0	0	0	5	0
s19abf	1	* 0	1	0	0	0	0	0	0	5	0
s19acf	1	* 0	1	0	0	0	0	0	0	5	0
s19adf	1	* 0	1	0	0	0	0	0	0	5	0
s20acf	1	* 0	1	0	0	0	0	0	0	5	0
s20adf	1	* 0	1	0	0	0	0	0	0	5	0
s21baf	2	* 0	1	0	0	0	0	0	0	5	0
s21bbf	3	* 0	1	0	0	0	0	0	0	5	0
s21bcf	3	* 0	1	0	0	0	0	0	0	5	0
s21bdf	4	* 0	1	0	0	0	0	0	0	7	0
x01aaf	0	* 0	0	0	1	0	0	0	0	5	6
x01abf	0	* 0	0	0	1	0	0	0	0	5	6

Appendix F

Main GLIM run

```
GLIM 4, update 8 for SGI Iris 4D / Irix on 22 Mar 1995 at 11:26:07
                (copyright) 1992 Royal Statistical Society, London

? $c GLIM prompts end with a question mark (?); comments appear in     $
? $c this form.  Blank lines have been added to this transcript, to    $
? $c facilitate its reading.                                           $

? $c Each row of the data matrix refers to a single NAG routine,       $
? $c included in IRENA: the names and meanings of column vectors are:  $
? $c     in           number of input parameters                       $
? $c     fn_flag      a flag (0 for a subroutine, 1 for a function)    $
? $c     out          number of output parameters                      $
? $c     io           number of input/output parameters                $
? $c     work         number of workspace parameters                   $
? $c     dummy        number of dummy parameters                       $
? $c     function     number of external function parameters           $
? $c     fn_param     total number of parameters of these functions    $
? $c     subroutn     number of external subroutine parameters         $
? $c     sr_param     total number of parameters of these subroutines  $
? $c     jazz         total number of lines in the jazz file           $
? $c     default      total number of lines in the defaults file.      $
```

```
? $c First set a default length for data vectors:                    $
? $units 160 $

? $c Now define the names of these vectors, then read the data matrix  $
? $c from a file (nominally on "channel 1"):                         $
? $data in fn_flag out io work dummy function
?       fn_param subroutn sr_param jazz default  $dinput 1 $
File name? complexity_data

? $c Add together the lengths of jazz and defaults files:            $
? $calc length = jazz + default $

? $c Allow for returned function values.                             $
? $c The colon (:) repeats the previous command.                     $
? $calc out = out + fn_flag : fn_param = fn_param + function $

? $c Define the dependent variate:                                   $
? $y length $

? $c Now use the standard GLIM starting model (a constant only,      $
? $c represented in GLIM as "1") to display the estimated total      $
? $c deviance about the mean. (Deviance is a generalisation of       $
? $c variance, equal to it for the model used here):                 $
? $fit $display e $
   deviance =  288784.
residual df =     159

          estimate        s.e.     parameter
     1       46.50       3.369      1
scale parameter 1816.
```

```
? $c Add in potential explanatory variables, to see their effect on    $
? $c the deviance, and display GLIM's parameter estimates (with their  $
? $c standard errors), using the "e" option of "display".              $
? $c (The scale parameter is not relevant to this model.)              $
? $fit + in + out + io $display e $
   deviance =  66485. (change =   -222299.)
residual df =    156  (change =       -3 )

          estimate        s.e.     parameter
     1      -18.24       3.375      1
     2       7.506      0.5731      IN
     3       3.147       1.212      OUT
     4       12.53       1.223      IO
scale parameter 426.2

? $c Comparing the standard error of an estimate with the estimate     $
? $c itself enables us to judge its significance.  In this case, the   $
? $c significance of "out" is somewhat low - below the 99% level - so  $
? $c let us combine the input, output and input/output counts:         $
? $calc io_total = in + out + 2*io $

? $c Start fitting again, with just a constant and this combined term: $
? $fit 1 + io_total $display e $
   deviance =  69900.
residual df =    158

          estimate        s.e.     parameter
     1      -19.45       3.399      1
     2       6.446      0.2898      IO_TOTAL
scale parameter 442.4
```

```
? $c Add in the effects of workspace and dummy parameters:            $
? $fit + work + dummy $display e $
   deviance =  68690. (change =   -1210.)
residual df =    156  (change =      -2 )

          estimate        s.e.     parameter
     1      -20.63       3.492      1
     2       6.520      0.3309      IO_TOTAL
     3      0.1739       1.909      WORK
     4       25.11       15.18      DUMMY
scale parameter 440.3

? $c The high relative standard errors indicate that the significance  $
? $c of these terms is very low - remove them:                         $
? $fit - work - dummy $
   deviance =  69900. (change =   +1210.)
residual df =    158  (change =      +2 )

? $c Now consider subroutine and function parameters:                  $
? $fit + subroutn + function $display e $
   deviance =  57684. (change =   -12216.)
residual df =    156  (change =       -2 )

          estimate        s.e.     parameter
     1      -17.13       3.141      1
     2       5.907      0.2814      IO_TOTAL
     3       12.08       2.175      SUBROUTN
     4       5.078       5.294      FUNCTION
scale parameter 369.8
```

```
? $c Although the significance of function is very low, it is retained $
? $c for later regrouping.  Next, try the effects of functions' and    $
? $c subroutines' own parameters:                                      $

? $fit + sr_param + fn_param $display e $
   deviance =  56714. (change =   -969.7)
residual df =    154  (change =     -2  )

          estimate        s.e.     parameter
     1      -16.24       3.188      1
     2       5.808      0.2893      IO_TOTAL
     3       4.335       5.513      SUBROUTN
     4       3.852       10.13      FUNCTION
     5       1.617       1.100      SR_PARAM
     6      0.8551       3.058      FN_PARAM
scale parameter 368.3

? $c Neither of these is significant alone - try regrouping            $
? $calc sr_total = subroutn + sr_param $
? $calc fn_total = function + fn_param $
? $fit - subroutn - sr_param - function - fn_param
?      + sr_total + fn_total $display e $
   deviance =  56820. (change =   +105.3)
residual df =    156  (change =     +2  )

          estimate        s.e.     parameter
     1      -16.07       3.145      1
     2       5.803      0.2840      IO_TOTAL
     3       2.016      0.3607      SR_TOTAL
     4       1.550       1.261      FN_TOTAL
scale parameter 364.2
```

```
? $c The fn_total effect is still insignificant - combine it with      $
? $c sr_total as sp_total ("subprogram-total"):                        $
? $calc sp_total = sr_total + fn_total $
? $fit - sr_total - fn_total + sp_total $display e $
   deviance =  56862. (change =   +42.43)
residual df =    157  (change =    +1   )

          estimate        s.e.     parameter
     1      -16.16       3.124      1
     2       5.804      0.2832      IO_TOTAL
     3       1.966      0.3276      SP_TOTAL
scale parameter 362.2

? $c Are there any higher order effects?                               $
? $calc io2 = io_total*io_total : sp2 = sp_total*sp_total $
? $fit + io2 + sp2 $display e $
   deviance =  56755. (change =   -107.0)
residual df =    155  (change =     -2  )

          estimate        s.e.     parameter
     1      -17.47       5.303      1
     2       6.038      0.8821      IO_TOTAL
     3       2.269      0.7946      SP_TOTAL
     4   -0.009200     0.03062      IO2
     5    -0.01192     0.02828      SP2
scale parameter 366.2
```

```
? $c Apparently not - but a negative constant term is implausible here $
? $c - what happens if we remove it?                                   $
? $fit - 1 $display e $
   deviance =  60726. (change =   +3971.)
residual df =    156  (change =      +1 )

          estimate        s.e.     parameter
     1       3.347      0.3431      IO_TOTAL
     2       2.545      0.8148      SP_TOTAL
     3     0.07204     0.01870      IO2
     4    -0.01775     0.02911      SP2
scale parameter 389.3

? $c Now the io2 term's effect is significant.  What about the higher  $
? $c order terms in io_total?                                          $
? $calc io3 = io_total*io2 $fit - sp2 + io3 $display e $
   deviance =  53425. (change =   -7302.)
residual df =    156  (change =       0 )

          estimate        s.e.     parameter
     1      0.3231      0.7289      IO_TOTAL
     2       1.780      0.3234      SP_TOTAL
     3      0.4730     0.08790      IO2
     4    -0.01084    0.002325      IO3
scale parameter 342.5
```

```
? $c These are still highly significant.  Further powers?              $
? $calc io4 = io2*io2 : io5 = io2*io3 $fit + io4 + io5 $display e $
   deviance =  52868. (change =   -556.6)
residual df =    154  (change =     -2  )

          estimate        s.e.     parameter
     1       2.936       2.576      IO_TOTAL
     2       1.795      0.3263      SP_TOTAL
     3     -0.3510      0.7224      IO2
     4     0.07369     0.06960      IO3
     5   -0.003439    0.002742      IO4
     6  0.00004790  0.00003769      IO5
scale parameter 343.3

? $c These are not significant - remove them and the now               $
? $c insignificant io_total                                            $
? $fit - io4 - io5 - io_total $
   deviance =  53492. (change =   +623.8)
residual df =    157  (change =     +3  )

? $c Do the higher power terms account for the previous negative       $
? $c constant?                                                         $
? $fit + 1 $display e $
   deviance =  53408. (change =   -84.30)
residual df =    156  (change =    -1   )

          estimate        s.e.     parameter
     1       1.350       2.720      1
     2       1.777      0.3220      SP_TOTAL
     3      0.4947     0.03995      IO2
     4    -0.01126    0.001434      IO3
scale parameter 342.4
```

```
? $c Yes - the effect of the constant term is now insignificant.       $
? $fit - 1 $
   deviance =  53492. (change =   +84.30)
residual df =    157  (change =    +1   )

? $c This looks like a good model - how much of the deviance is        $
? $c explained?  (Calculate the fraction remaining.)                   $
? $calc 53408/288784 $
        0.1849
? $c 82% accounted for.                                                $

? $c We should, perhaps, recheck work and dummy:                       $
? $fit + work + dummy $display e $
   deviance =  52816. (change =   -676.2)
residual df =    155  (change =     -2  )

          estimate        s.e.     parameter
     1       1.859      0.3305      SP_TOTAL
     2      0.5221     0.02649      IO2
     3    -0.01204    0.001036      IO3
     4      -1.995       1.725      WORK
     5       10.49       13.05      DUMMY
scale parameter 340.7

? $c They are still insignificant.                                     $

? $fit - work - dummy $
   deviance =  53492. (change =   +676.2)
residual df =    157  (change =     +2  )
```

```
? $c Take a last look at the residuals:                                    $
? $calc res = length - %fv $plot res %fv $
          |
          |                                  R
          |
          |
          |
          |                                              R
      50. +             2       R                                    R
          |                     R               R
          |                                     2
          |     R                         R
          |     2 R  R  2                               R
          |    2R R     2       R     RR
          |2    3 R 2 4 2  2 R 2 R  R    R                    R
       0. +R  922 6 8 2 R  3         2                 R      R
          |    R  R 4 R R2 R   2R 2              R           R             R
          |     2 R   R 2    2 R2    R   R                      R
          |                2 3 R  2   R  R  R                               R
          |                  R    2                 R         R
          |                     R                 R
          |                           R                      R
     -50. +
           +---------------+---------------+---------------+-----------
           0.              50.             100.            150.
? $c These are pretty well scattered (if a bit non-uniform).               $

? $c Recall what the final parameter values were:                          $
? $display e$
          estimate        s.e.     parameter
     1       1.758      0.3188      SP_TOTAL
     2      0.5104     0.02446      IO2
     3    -0.01177    0.001007      IO3
scale parameter 340.7
```

```
? $c Express the estimates as coefficients of io_total (%pe contains   $
? $c the parameter estimates, %b and %c are scalar variables):         $
? $extract %pe $
? $calc %b = %sqrt(%pe(2)) : %c = -%exp(%log(-%pe(3))/3) $look %b %c $
   0.7144  -0.2275
? $c So we can express the relationship as                             $
? $c    length = 1.758*sp_total                                        $
? $c             + (0.7144*io_total)**2 - (0.2275*io_total)**3         $
? $c and avoid the false impression that third order effect is small.  $

? $c Now look at length/io_total ratios in the io_total inter-quartile $
? $c ranges.  (The "sort" command sorts the contents of the second     $
? $c vector into the first, by applying the permutation which would    $
? $c arrange the last in ascending order.)                             $
? $sort srtd_len length io_total : srtd_iot io_total $
? $c The output from the next command has been edited to remove lines  $
? $c of no immediate interest, leaving those sections where values     $
? $c change to those at the quartiles.  Missing sections are indicated $
? $c by lines containing only a colon (:).                             $
? $look srtd_iot $c to find where the quartiles lie. $
        SRTD_IOT
   1    0.000
      :
  34    5.000
  35    6.000
  36    6.000
  37    6.000
  38    6.000
  39    6.000
  40    6.000
  41    6.000
  42    6.000
  43    6.000
  44    6.000
```

```
 45    6.000
 46    7.000
     :
 72    8.000
 73    9.000
 74    9.000
 75    9.000
 76    9.000
 77    9.000
 78    9.000
 79    9.000
 80    9.000
 81    9.000
 82   10.000
     :
117   12.000
118   13.000
119   13.000
120   13.000
121   13.000
122   13.000
123   13.000
124   13.000
125   13.000
126   14.000
     :
160   32.000
```

```
? $c Define lengths for the subrange- and inter-quartile-range-total   $
? $c (q) vectors (and the ratio of the latter):                        $
? $variate 7 tot_len tot_iot : 4 qtot_len qtot_iot ratlenio $

? $c Now calculate the totals over the various ranges (%cu is the      $
? $c cumulative sum, temp is a vector variable; the endpoints of the   $
? $c ranges bracketing the quartiles came from the above "look"):      $
? $calc temp = %cu(srtd_len) $
? : tot_len(1) = temp(34) $
? : tot_len(2) = temp(45)  - temp(34) $
? : tot_len(3) = temp(72)  - temp(45) $
? : tot_len(4) = temp(81)  - temp(72) $
? : tot_len(5) = temp(117) - temp(81) $
? : tot_len(6) = temp(125) - temp(117) $
? : tot_len(7) = temp(160) - temp(125) $
? : temp = %cu(srtd_iot) $
? : tot_iot(1) = temp(34) $
? : tot_iot(2) = temp(45)  - temp(34) $
? : tot_iot(3) = temp(72)  - temp(45) $
? : tot_iot(4) = temp(81)  - temp(72) $
? : tot_iot(5) = temp(117) - temp(81) $
? : tot_iot(6) = temp(125) - temp(117) $
? : tot_iot(7) = temp(160) - temp(125) $
? : qtot_len(1) =                  tot_len(1) + 6/11*tot_len(2) $
? : qtot_len(2) = 5/11*tot_len(2) + tot_len(3) +  8/9*tot_len(4) $
? : qtot_len(3) =  1/9*tot_len(4) + tot_len(5) +  3/8*tot_len(6) $
? : qtot_len(4) =  5/8*tot_len(6) + tot_len(7) $
? : qtot_iot(1) =                  tot_iot(1) + 6/11*tot_iot(2) $
? : qtot_iot(2) = 5/11*tot_iot(2) + tot_iot(3) +  8/9*tot_iot(4) $
? : qtot_iot(3) =  1/9*tot_iot(4) + tot_iot(5) +  3/8*tot_iot(6) $
? : qtot_iot(4) =  5/8*tot_iot(6) + tot_iot(7) $
```

```
? $c ... and, finally, the desired quartile ratios:                    $
? $calc ratlenio = qtot_len/qtot_iot $look ratlenio $
     RATLENIO
 1    2.206
 2    3.563
 3    4.433
 4    5.569
? $stop $
```

Appendix G

Source of nagpolysolve

```
procedure nagpolysolve pn;

begin scalar templist, degree, purereal, !*verbose, result;
share !*verbose;

!*nag!-mnemon!-param1!* := first(templist := polycoefs pn);
degree := second templist;
purereal := third templist;
lisp(!*verbose := nil);
if degree = -1 then
   << write "*** Zero polynomial supplied: solution indeterminate";
      return
   >>
else if degree = 0 then
   << write("*** ", pn, " = 0 has no solution");
      return
   >>
```

```
else if degree = 1 then
   return -!*nag!-mnemon!-param1!*(2,1)/!*nag!-mnemon!-param1!*(1,1)
else if degree = 2 then
   if purereal then
      return first c02ajf(coefficients=!*nag!-mnemon!-param1!*)
   else
      return first c02ahf(coefficients=!*nag!-mnemon!-param1!*)
else
   << on mnemprompts;  % c02aff and c02agf have non-housekeeping defaults
       % (switched off again automatically)
     result := if purereal then
                  first c02agf(coefficients=!*nag!-mnemon!-param1!*)
     else
                  first c02aff(coefficients=!*nag!-mnemon!-param1!*);
     return result
  >>
end$


symbolic operator polycoefs;

symbolic procedure polycoefs pn;

begin scalar purereal, saverounded, savecomplex, savefactor, savedmode,
     saveiidvalfn, pp, !*numval, npn, num, den, degree, result,
     term, oldvbl, vbl, coef, coefs, oldexpnt, expnt;

purereal := 't;

savefactor := !*factor;
off factor;
pp := algebraic(print!-precision(-1));  % Otherwise affects functioning of simp
```

```
% The following code gets rid of PIs and Es

% First save settings associated with ROUNDED and COMPLEX

saverounded := !*rounded;
savecomplex := !*complex;
savedmode := dmode!*;
saveiidvalfn := get('i,'idvalfn);

% Now mimic ON COMPLEX, ROUNDED

!*rounded := t;
!*complex := t;
dmode!* := '!:cr!:;
put('i,'idvalfn,'mkdcrn);
rmsubs();

!*numval := 't;
npn := prepsq!* simp pn;  % npn is now the complex, rounded equivalent of pn

% easier to work with rationals in REDUCE, so mimic OFF ROUNDED

!*rounded := nil;
dmode!* := '!:gi!:;
put('i,'idvalfn,'mkdgi);
rmsubs();

npn := simp npn;
num := car npn;
den := cdr npn;
if not numberp den then typerr(pn, "a polynomial");
```

```
if null num then
   << degree := -1;
      result := 'mat . list list 0;
      go to tidyup
   >>
else if numberp num or complex!-integerp num then
   << degree := 0;
      result := 'mat . list list quotient(num,den);
      go to tidyup
   >>;
term := car num;
num := cdr num;
oldvbl := caar term;
if not atom oldvbl then typerr(oldvbl, "a variable");
oldexpnt := (expnt := cdar term);
if not numberp expnt then typerr(expnt, "a real exponent");
degree := expnt;
coef := cdr term;
if not(numberp coef or complex!-integerp coef) then
   typerr(pn, "a univariate polynomial");
if complex!-integerp coef then purereal := nil;
coefs := list list prepsq!*(coef . den);
while expnt neq 0 do
<< if null num then
      << expnt := 0;
 coef := 0
      >>
   else if numberp num or complex!-integerp num then
      << expnt := 0;
 coef := num;
         num := nil
      >>
   else
      << term := car num;
```

```
         vbl := caar term;
         if not atom vbl then typerr(vbl, "a variable");
         if vbl neq oldvbl then typerr(pn, "a univariate polynomial");
         expnt := cdar term;
         if not numberp expnt then typerr(expnt, "a real exponent");
         coef := cdr term;
         if not(numberp coef or complex!-integerp coef) then
            typerr(pn, "a univariate polynomial");
         num := cdr num;
      >>;
   while expnt neq oldexpnt - 1 do
   << oldexpnt := oldexpnt - 1;
      coefs := list 0 . coefs
   >>;
   oldexpnt := expnt;
   if complex!-integerp coef then purereal := nil;
   coefs := list prepsq!*(coef . den) . coefs
>>;
result := 'mat . reverse coefs;
tidyup :
if savefactor then on factor;
!*rounded := saverounded;
!*complex := savecomplex;
dmode!* := savedmode;
put('i,'idvalfn,saveiidvalfn);
rmsubs();
if pp then algebraic(print!-precision pp);
return list('list,result,degree,purereal);

end$


symbolic procedure complex!-integerp n;
begin return eqcar(n,'!:gi!:) end;
```

Appendix H

Fortran 90 jacket for D01AJF and related modules

In a fully developed system, each jacket would **USE** the modules displayed in sections H.1 and H.3, extended to provide **COMPLEX** analogues of the **REAL** types. If the strategy used in IRENA-1, of constructing and compiling an entire program for each run, were again adopted, this would also apply to the module displayed in section H.2; however, as mentioned in section 15.3.5, the "second level defaults" may be changed by users at any time, so, if such a module were included in the final scheme, it could only be compiled at run time. In contrast, if a future IRENA-like system uses the potentially more time-efficient strategy of building and compiling a partial Fortran program for each IRENA-function when the system is built (leaving only ASPs to be built, compiled and linked in at run time) and supplying parameter values as data to **READ**, this module would not be required.

H.1 General precision setting

```
MODULE irena_kinds
INTEGER, PARAMETER :: ireal = KIND(1D0)
END MODULE irena_kinds
```

H.2 Second level defaults

```
MODULE global_defaults
USE irena_kinds
REAL(KIND=ireal), PARAMETER :: user_abs_err   = 0.0001,          &
                               user_rel_err   = 0.0001,          &
                               user_mix_err   = 0.0001,          &
                               user_input_err = 0.0001
END MODULE global_defaults
```

H.3 Derived types for named output

```
MODULE output_types

USE irena_kinds

 TYPE integer_output_scalar
  CHARACTER(LEN=120) :: name
  INTEGER :: value
 END TYPE integer_output_scalar

 TYPE real_output_scalar
  CHARACTER(LEN=120) :: name
  REAL(KIND=ireal) :: value
 END TYPE real_output_scalar
```

```
 TYPE real_output_array_1
  CHARACTER(LEN=120) :: name
  REAL(KIND=ireal), POINTER, DIMENSION(:) :: value
 END TYPE real_output_array_1

 TYPE real_output_array_2
  CHARACTER(LEN=120) :: name
  REAL(KIND=ireal), POINTER, DIMENSION(:,:) :: value
 END TYPE real_output_array_2

END MODULE output_types
```

H.4 Jacket for D01AJF

```
MODULE jackets

CONTAINS

   SUBROUTINE d01ajf_jac(                                          &

   ! Essential input parameters:

         f, a_with_b,                                              &

   ! Output parameters:

         result_with_name, abserr_with_name, iw_1_with_name,       &
         w_1_with_name, w_2_with_name, w_3_with_name,              &
```

```
! Optional input parameters:

      absolute_accuracy, relative_accuracy, workspace_length       &

                    )

USE irena_kinds
USE output_types
USE global_defaults

! Combine a and b into a single PARAMETER (a (1 x 2) 2-d array since
! we shall in general use (n x 2) 2-d arrays for "rectangles"):

REAL(KIND=ireal), DIMENSION(1,2) :: a_with_b

! Allow for defaults:

INTEGER, OPTIONAL :: workspace_length
REAL(KIND=ireal), OPTIONAL :: absolute_accuracy, relative_accuracy

! Make workspace allocatable:

INTEGER, DIMENSION(:), ALLOCATABLE :: iw
REAL(KIND=ireal), DIMENSION(:), ALLOCATABLE :: w

! Other d01ajf parameters:

INTEGER :: lw, liw, ifail
REAL(KIND=ireal) :: f, a, b, epsabs, epsrel, result, abserr
EXTERNAL f
```

```
! Output parameters:

TYPE(integer_output_scalar) :: iw_1_with_name
TYPE(real_output_scalar) :: result_with_name, abserr_with_name
TYPE(real_output_array_1) :: w_2_with_name, w_3_with_name
TYPE(real_output_array_2) :: w_1_with_name

! Local variable:

INTEGER :: n

! Unpack a and b:

a = a_with_b(1,1)
b = a_with_b(1,2)

! Incorporate defaults for epsabs, epsrel and lw:

IF (PRESENT(absolute_accuracy)) THEN
    epsabs = absolute_accuracy
ELSE
    epsabs = user_abs_err
END IF

IF (PRESENT(relative_accuracy)) THEN
    epsrel = relative_accuracy
ELSE
    epsrel = user_rel_err
END IF

IF (PRESENT(workspace_length)) THEN
    lw = workspace_length
    lw = MAX(lw,4)      ! Deals with constraint.
```

```
ELSE
    lw = 2000
END IF

! Eliminate liw:

liw = lw/4      ! Can ignore constraint here since lw/4 already >= 1.
ALLOCATE(w(1:lw))
ALLOCATE(iw(1:liw))

! Now call the NAG F77 routine:
ifail = -1
CALL d01ajf(f,a,b,epsabs,epsrel,result,abserr,w,lw,iw,liw,ifail)

! Have structures for result and abserr, with names as text strings:

result_with_name % value = result
result_with_name % name = 'Integral'

abserr_with_name % value = abserr
abserr_with_name % name = 'Absolute_error_estimate'

IF (ifail == 0) THEN

!  use this:

   iw_1_with_name % value = iw(1)
   iw_1_with_name % name = 'Number_of_subintervals_used'

!  but not these:

   w_1_with_name % name = ''          !  empty names indicate
   w_2_with_name % name = ''          !  "unused" output parameter
   w_3_with_name % name = ''          !  structures
```

```
   ELSE

   !  want diagnostic outputs - use these:

      n = iw(1)

      ALLOCATE(w_1_with_name % value(1:n,2))
      ALLOCATE(w_2_with_name % value(1:n))
      ALLOCATE(w_3_with_name % value(1:n))

      w_1_with_name % value(:,1) = w(1:n)
      w_1_with_name % value(:,2) = w(n+1:2*n)
      w_1_with_name % name = 'Subintervals'

      w_2_with_name % value = w(2*n+1:3*n)
      w_2_with_name % name = 'Integral_approximations_on_subintervals'

      w_3_with_name % value = w(3*n+1:4*n)
      w_3_with_name % name =                                          &
                      'Error_estimates_for_subinterval_approximations'

   !  but not this:

      iw_1_with_name % name = ''

   END IF

   RETURN

   END SUBROUTINE d01ajf_jac

END MODULE jackets
```

H.5 Test program

```
USE irena_kinds
USE output_types
USE global_defaults
USE jackets

REAL(KIND=ireal) :: integrand
EXTERNAL integrand
REAL(KIND=ireal), DIMENSION(1,2) :: range
TYPE(integer_output_scalar), DIMENSION(1) :: ios
TYPE(real_output_scalar), DIMENSION(2)    :: ros
TYPE(real_output_array_1), DIMENSION(2)   :: oa1
TYPE(real_output_array_2)                 :: oa2

range(1,1) = 0.0
range(1,2) = 1.0

testloop: DO itest = 1,2
   IF (itest == 1) THEN
      WRITE(*, '(/A)', ADVANCE='NO') 'Normal run: '
      CALL d01ajf_jac(integrand, range, ros(1), ros(2), ios(1),      &
                      oa2, oa1(1), oa1(2))
   ELSE
      WRITE(*, '(/A/)') 'Abnormal run: D01AJF output on error channel..'
      CALL d01ajf_jac(integrand, range, ros(1), ros(2), ios(1),      &
                      oa2, oa1(1), oa1(2),workspace_length=20)
      WRITE(*, '(/14X)', ADVANCE='NO')
   ENDIF
```

```
   ! Print those output parameters with non-empty names:

   WRITE(*,'(A/)') 'Test program output on standard channel..'

   DO I = 1,2
      IF (ros(i) % name /= '') THEN
         WRITE(*,'(A46," = ",(E25.17))') ros(i)
         WRITE(*,*)
      ENDIF
   ENDDO

   IF (ios(1) % name /= '') THEN
      WRITE(*,'(A46," = ",I9/)') ios(1)
   ENDIF

   IF (oa2 % name /= '') THEN
      jmax = UBOUND(oa2 % value,1)
      WRITE(*, '(A46," = "/(24X,2E25.17))')                          &
               oa2 % name, ((oa2 % value(j,k),k=1,2),j=1,jmax)
               ! value written thus as stored in column-major orderZZ
      WRITE(*,*)
   ENDIF

   DO I = 1,2
      IF (oa1(i) % name /= '') THEN
         WRITE(*, '(A46," = ", E25.17, /(49X,E25.17))')              &
                  oa1(i) % name, oa1(i) % value
         WRITE(*,*)
      ENDIF
   ENDDO

ENDDO testloop

END
```

```
REAL(KIND=ireal) FUNCTION integrand(x)

USE irena_kinds

REAL(KIND=ireal) :: x
integrand = 1/sqrt(1-x**2)
RETURN

END
```

H.6 Test program output

```
Normal run: Test program output on standard channel..

Integral                                    =   0.15707963267867724E+01

Absolute_error_estimate                     =   0.16694195379418630E-05

Number_of_subintervals_used                 =         6


Abnormal run: D01AJF output on error channel..

 ** The maximum number of subdivisions (LIMIT) has been reached:
    LIMIT =                5   LW =               20   LIW =                5
 ** ABNORMAL EXIT from NAG Library routine D01AJF: IFAIL =     1
 ** NAG soft failure - control returned
```

```
          Test program output on standard channel..

Integral                                        =   0.15650556432179701E+01

Absolute_error_estimate                         =   0.16793990427787381E+00

Subintervals                                    =
                        0.00000000000000000E+00  0.50000000000000000E+00
                        0.50000000000000000E+00  0.75000000000000000E+00
                        0.75000000000000000E+00  0.87500000000000000E+00
                        0.87500000000000000E+00  0.93750000000000000E+00
                        0.93750000000000000E+00  0.10000000000000000E+01

Integral_approximations_on_subintervals         =   0.29093955163239531E-14
                                                    0.18028920694018435E-14
                                                    0.12078450271614016E-14
                                                    0.83314318610740795E-15
                                                    0.16793990427786695E+00

Error_estimates_for_subinterval_approximations =    0.52359877559829893E+00
                                                    0.32446330338318213E+00
                                                    0.21737373752925832E+00
                                                    0.14993930859393378E+00
                                                    0.34968051811329698E+00
```

Appendix I

Current and extended specfiles

This appendix illustrates the current specfile for `d01ajf`, and a hypothetical, new style specfile, as described in section 15.4. The new style specfile is presented in its automatically generated "skeleton" form and in a final form incorporating jazzing and completed default setting.

To facilitate comparison of the three forms, they are interleaved, with each section introduced by a line of the form

ijk----------

where *i* may be - or 1, *j* may be - or 2 and *k* may be - or 3: the presence of 1 indicates that the following section applies to the current specfile, 2 to the skeleton new style specfile and 3 to the completed new style specfile.

Differences in layout have been ignored.

```
123------------------------------------------------------------------------
#### TYPE ####

SUBROUTINE

1------------------------------------------------------------------------
#### SPECIFICATION ####

      //D01AJF//(F,A,B,EPSABS,EPSREL,RESULT,
     1 ABSERR,W,LW,IW,LIW,IFAIL)
C     INTEGER LW,IW(LIW),LIW,IFAIL
C     //real// F,A,B,EPSABS,EPSREL,RESULT,
C    1 ABSERR,W(LW)
C     EXTERNAL F

-23-----------------------------------------------------------------------
#### SPECIFICATION ####

      D01AJF(F,A,B,EPSABS,EPSREL,RESULT,
     1 ABSERR,W,LW,IW,LIW,IFAIL)

1------------------------------------------------------------------------
#### PARAMETERS ####

**** INPUT PARAMETERS:

A
B
EPSABS
EPSREL
LW
LIW

-23-----------------------------------------------------------------------
```

```
#### NAG PARAMETERS ####

**** INPUT PARAMETERS:

A      : real scalar
B      : real scalar
EPSABS : real scalar
EPSREL : real scalar
LW     : integer scalar
LIW    : integer scalar

1------------------------------------------------------------------------
**** OUTPUT PARAMETERS:

RESULT
ABSERR
W@
IW@

-23----------------------------------------------------------------------
**** OUTPUT PARAMETERS:

RESULT  : real scalar
ABSERR  : real scalar
W(LW)   : real vector
IW(LIW) : integer vector

1------------------------------------------------------------------------
**** INPUT/OUTPUT PARAMETERS:

IFAIL
```

```
-23-------------------------------------------------------------------------
**** INPUT/OUTPUT PARAMETERS:

IFAIL : integer scalar

1-------------------------------------------------------------------------
**** WORKSPACE PARAMETERS:

None.

-23-------------------------------------------------------------------------
**** WORKSPACE PARAMETERS:

% none

1-------------------------------------------------------------------------
**** DUMMY PARAMETERS:

None.

-23-------------------------------------------------------------------------
**** DUMMY PARAMETERS:

% none

1-------------------------------------------------------------------------
**** FUNCTIONS:

NAME:     F
SUPPLIER: USER
TYPE:     1

    //real// FUNCTION F(X)
    //real//          X
```

```
-23-------------------------------------------------------------------------
**** FUNCTIONS:

D01AJF_F : real function

1--------------------------------------------------------------------------
**** SUBROUTINES:

None.

-23-------------------------------------------------------------------------
**** SUBROUTINES:

% none

-23-------------------------------------------------------------------------
#### IRENA PARAMETERS ####

-2--------------------------------------------------------------------------
**** INPUT PARAMETERS:

irena_a      : scalar data/control/housekeeping % delete two options
irena_b      : scalar data/control/housekeeping % delete two options
irena_epsabs : scalar data/control/housekeeping % delete two options
irena_epsrel : scalar data/control/housekeeping % delete two options
irena_lw     : scalar data/control/housekeeping % delete two options
irena_liw    : scalar data/control/housekeeping % delete two options
irena_ifail  : scalar data/control/housekeeping % delete two options

% promote requirements of D01AJF_F ... includes

irena_f(x)   : function
```

```
--3-------------------------------------------------------------------------
**** INPUT PARAMETERS:

region       : rectangle(1) data suppliedAs(region, << range >>)
irena_epsabs : scalar       control suppliedAs(absolute accuracy required,
                                               << absacc, aar >>)
irena_epsrel : scalar       control suppliedAs(relative accuracy required,
                                               << relacc, rar >>)
irena_liw    : scalar       control suppliedAs(
                                    maximum number of subintervals allowed,
                                               << maxints, mnsa >>)
irena_f      : function     data suppliedAs(integrand(x), << f >>)

-23-------------------------------------------------------------------------
**** INTERMEDIATE INPUT OBJECTS:

% none

-2--------------------------------------------------------------------------
**** INPUT REDEFINITION:

A            := irena_a
B            := irena_b
EPSABS       := irena_epsabs
EPSREL       := irena_epsrel
LW           := irena_lw
LIW          := irena_liw    = irena-lw/4 % Suggested value
IFAIL        := irena_ifail  := -1

--3-------------------------------------------------------------------------
**** INPUT REDEFINITION:

A            := region(1,1)
B            := region(1,2)
```

```
EPSABS       := irena_epsaps = global(*userabserr*)
EPSREL       := irena_epsrel = global(*userrelerr*)
LW           := 4*LIW
LIW          := irena_liw = 500
IFAIL        := -1

-2-------------------------------------------------------------------------
**** OUTPUT PARAMETERS:

irena_result : scalar
irena_abserr : scalar
irena_w      : vector
irena_iw     : vector

--3-----------------------------------------------------------------------
**** OUTPUT PARAMETERS:

integral                                        : scalar
absolute error estimate                         : scalar
number of subintervals used                     : scalar
subintervals                                    : list(rectangle(1))
integral approximations on subintervals         : list(scalar)
error estimates for subinterval approximations : list(scalar)

-2-------------------------------------------------------------------------
**** INTERMEDIATE OUTPUT OBJECTS:

% none

--3-----------------------------------------------------------------------
**** INTERMEDIATE OUTPUT OBJECTS:

iw1 : scalar
```

```
-2-------------------------------------------------------------------------
**** OUTPUT REDEFINITION

irena_result := RESULT
irena_abserr := ABSERR
irena_w      := W
irena_iw     := IW

--3------------------------------------------------------------------------
**** OUTPUT REDEFINITION

integral                    := RESULT

absolute error estimate     := ABSERR

iw1                         := IW(1)

number of subintervals used := if out(IFAIL) = 0 then iw1
                               else unset
                               endif

subintervals                := if out(IFAIL) ~= 0 then W(1:2*iw1)
                               else unset
                               endif

error estimates for subinterval approximations
      := if out(IFAIL) ~= 0 then W(2*iw1+1:3*iw1)
         else unset
         endif

integral approximations on subintervals
      := if out(IFAIL) ~= 0 then W(3*iw1+1:4*iw1)
         else unset
         endif
```

```
123-------------------------------------------------------------------------
#### IFAIL VALUES ####
#EQ1
12--------------------------------------------------------------------------
    The maximum number of subdivisions allowed with the given
    workspace has been reached without the accuracy requirements
--3-----------------------------------------------------------------------
    The maximum number of subdivisions allowed
    has been reached without the accuracy requirements
123-------------------------------------------------------------------------
    being achieved. Look at the integrand in order to determine the
    integration difficulties. If the position of a local difficulty
    within the interval can be determined (e.g. a singularity of
    the integrand or its derivative, a peak, a discontinuity, etc.)
    you will probably gain from splitting up the interval at this
    point and calling the integrator on the subranges. If
    necessary, another integrator, which is designed for handling
    the type of difficulty involved, must be used. Alternatively,
12--------------------------------------------------------------------------
    consider relaxing the accuracy requirements specified by EPSABS
    and EPSREL, or increasing the amount of workspace.
--3-----------------------------------------------------------------------
    consider relaxing the absolute or relative accuracy requirements
    or increasing the maximum number of subintervals allowed.
    Please note that divergence may have occurred.
123-------------------------------------------------------------------------

#EQ2
    Roundoff error prevents the requested tolerance from being
    achieved. The error may be under-estimated. Consider relaxing
12--------------------------------------------------------------------------
    the accuracy requirements specified by EPSABS and EPSREL, or
    increasing the amount of workspace.
```

```
    Please note that divergence can occur with any non-zero value
    of IFAIL.
--3-------------------------------------------------------------------------
    the absolute or relative accuracy requirements or increasing the
    maximum number of subintervals allowed.
    Please note that divergence may have occurred.
123-------------------------------------------------------------------------

#EQ3
    Extremely bad local integrand behaviour causes a very strong
    subdivision around one (or more) points of the interval. Look
    at the integrand in order to determine the integration
    difficulties. If the position of a local difficulty within the
    interval can be determined (e.g. a singularity of the integrand
    or its derivative, a peak, a discontinuity ...) you will
    probably gain from splitting up the interval at this point and
    calling the integrator on the subranges. If necessary, another
    integrator, which is designed for handling the type of
    difficulty involved, must be used. Alternatively, consider
12------------------------------------------------------------------------
    relaxing the accuracy requirements specified by EPSABS and
    EPSREL, or increasing the amount of workspace.
    Please note that divergence can occur with any non-zero value
    of IFAIL.
--3-------------------------------------------------------------------------
    relaxing the absolute or relative accuracy requirements
    or increasing the amount of maximum number of subintervals allowed.
    Please note that divergence may have occurred.
123-------------------------------------------------------------------------

#EQ4
    The requested tolerance cannot be achieved, because the
    extrapolation does not increase the accuracy satisfactorily;
    the returned result is the best which can be obtained. Look at
```

```
    the integrand in order to determine the integration
    difficulties. If the position of a local difficulty within the
    interval can be determined (e.g. a singularity of the integrand
    or its derivative, a peak, a discontinuity ...) you will
    probably gain from splitting up the interval at this point and
    calling the integrator on the subranges. If necessary, another
    integrator, which is designed for handling the type of
    difficulty involved, must be used. Alternatively, consider
12------------------------------------------------------------------------------
    relaxing the accuracy requirements specified by EPSABS and
    EPSREL, or increasing the amount of workspace.
    Please note that divergence can occur with any non-zero value
    of IFAIL.
--3---------------------------------------------------------------------------
    relaxing the absolute or relative accuracy requirements
    or increasing the amount of maximum number of subintervals allowed.
    Please note that divergence may have occurred.
123-----------------------------------------------------------------------------

#EQ5
    The integral is probably divergent, or slowly convergent.
12------------------------------------------------------------------------------
    Please note that divergence can occur with any non-zero value
    of IFAIL.

#EQ6
    On entry, LW < 4,
    or        LIW < 1. Please note that divergence can occur with
    any non-zero value of IFAIL.

-23-----------------------------------------------------------------------------

#### ASPs ####
```

```
### TYPE ###

real FUNCTION

### SPECIFICATION ###

      D01AJF_F(X)

### NAG PARAMETERS ###

**** INPUT PARAMETERS:

X      : real scalar

**** OUTPUT PARAMETERS:

% none

**** INPUT/OUTPUT PARAMETERS:

% none

**** WORKSPACE PARAMETERS:

% none

**** DUMMY PARAMETERS:

% none

**** FUNCTIONS:

% none
```

**** SUBROUTINES:

% none

% end of D01AJF_F

% end of D01AJF

Glossary

Please note that non-alphabetic characters have been ignored in collating the entries in this glossary.

Alias file — A user-supplied file, specific to each IRENA-function, which allows the introduction of additional aliases for input parameters and the renaming of output parameters.

ASP — Argument Sub-Program. A parameter to a Fortran subprogram, which is itself a subprogram. Also, the specification of such a subprogram in terms of mathematical objects such as matrices and functions.

Auxiliary routine — A routine in the NAG Library, which is not intended to be directly called by users. The term covers both the component routines which provide the underlying functionality of the top level routines and routines which are intended to be used as external parameters to NAG routines, providing alternative functionality (as in the case of **D01BAW**, **D01BAX**, **D01BAY** and **D01BAZ** which handle different quadrature formulae for **D01BBF**) or a default functionality (for example, **E04NFU** provides a matrix × vector multiplication facility for **E04NFF** which is dependent on a standard representation being used for the matrix). If the default functionality is null, the routine is described as "dummy".

Chapter In the NAG Library a "chapter" is a subset of routines, concerned with the same area of numerical calculation and sharing a common prefix based on the extended SHARE classification [1]. This prefix usually consists of a letter followed by two digits – for example, `D01` for quadrature routines – although the "special functions" are considered to form a single `S` chapter.

Column major order An ordering used in storing the entries in a matrix or other two-dimensional structure, in which complete columns of entries are stored in succession. The standard ordering for Fortran two-dimensional arrays.

Control parameter A non-data parameter (in a Fortran routine) which controls the behaviour of the underlying algorithm or other aspects of the routine, such as the frequency of displaying intermediate results. Common examples are convergence criteria and error tolerances.

Data parameter A parameter specifying the actual data which a routine is to process.

Defaults file A routine-specific file, defining default values for NAG parameters as constants or functions of other parameters.

Dummy parameter A parameter in a NAG routine which is not accessed by the routine. Dummy parameters are sometimes used to preserve the NAG interface of a routine whose internal functioning has been revised.

Dummy routine An "auxiliary" routine, in the NAG Library, which may be used as an external parameter to a top level Library routine, when the functionality which that parameter allows is not required.

Envsearch An IRENA switch which, if on, permits values in the REDUCE environment to be recognised as parameters of IRENA-functions.

Fortinclude An IRENA switch which, if **on**, prompts the user for the names of two files of fragments of Fortran code which may contain, respectively, code to be inserted in all the Fortran subprograms (including the main program) and in the main program only, immediately before the executable statements generated by GENTRAN.

Fpeps The smallest floating point number, "safely" representable in a particular Fortran implementation, which, when added to 1 produces a value different to 1. "Safely representable" means that both the number and its negative can be represented and that certain arithmetic operations yield a result; see [26] for further details. This symbol, normally used in IRENA defaults files, produces a call to the NAG routine **X02AJF** in the generated Fortran.

Fphuge The largest floating point number "safely" representable in a particular Fortran implementation. The symbol **fphuge**, used in IRENA function calls (usually represented as *) or in defaults files, produces a call to the NAG routine **X02ALF** in the generated Fortran.

Fset An IRENA notation for defining an indexed family of functions, principally used to satisfy the requirements of ASPs.

GENTRAN A REDUCE package for converting REDUCE code to Fortran, C and other languages. Originally developed for Macsyma. See [12].

Housekeeping An entry in a defaults file specifying which parameters are to be regarded as housekeeping parameters; prompts will not be generated for these, even with **promptall on**, unless their values are unspecified but are required to establish values for NAG parameters.

Housekeeping parameter A parameter which is required, not by the logic of the problem being solved but only for the correct functioning of the Fortran routine. Common examples are parameters specifying workspace arrays and those giving the dimensions of data arrays.

IFAIL An input-output parameter in most NAG routines. Its input value controls the behaviour of the routine on detecting an error (in IRENA, it is always set to -1, to take advantage of any English error messages which the routine may print and to allow a return to the calling Fortran, after an error is detected. An output value of zero indicates successful completion of the routine call, different non-zero values indicate different causes of failure.

Infofile A file, generated automatically from the specfile as part of the IRENA setup process, to provide routine-specific information in IRENA.

IRENA-function A function provided within IRENA to generate and run Fortran code which calls one or more NAG routines. Other (REDUCE) functions provided by the IRENA package, including those which call IRENA-functions, are not described as IRENA-functions.

Jacket A subprogram which calls one or more other subprograms, to provide these with an alternative interface. Some IRENA functionality was provided by writing Fortran jackets for NAG routines.

Jazz The IRENA system whereby the user interfaces of routines are redefined.

Jazz file A file of jazz commands, redefining the user interface of a particular routine.

Jazz-function An RLISP function written to provide additional input jazzing functionality not present in the original IRENA system. In conjunction with each jazz function, two other RLISP functions must be provided, to deliver the dimensions of the NAG parameter being processed and to check whether all of the objects needed to specify that parameter are available.

In contrast with the original jazz commands, which only require the names of the NAG and IRENA parameters between which a mapping is being defined, its use requires the provision of a Lisp object as the final syntactic element. This may simply be the name of an IRENA input parameter but, in some instances, is considerably more complex.

Key *or* **Key-alias** A symbol which may be used in a call to an IRENA-function to introduce a value for a particular parameter, using the syntax *key=value*. If `envsearch` is `on`, it also defines the name of a REDUCE variable which may be used to supply the parameter value, prior to the function call.

Keyline The collection of parameter definitions in an IRENA-function, initially supplied by the user in the function call.

Keyword A symbol whose appearance in an IRENA keyline or in response to an IRENA prompt defines a fixed value for a particular parameter, without the value being explicitly specified there.

Long form (output name) An alternative name, automatically generated for each IRENA output parameter, in which the normal output name is prefixed by the name of the generating routine. This provides extra security when parameters are passed between paired IRENA-functions and allows users to automatically retain synonymous output from related functions.

NAGlink Symbolic-numeric link, between Axiom release 2 and the NAG Fortran Library, developed at NAG under the auspices of the Teaching Company Scheme.

Naglink — Early symbolic-numeric link, between Macsyma and the NAG Fortran Library, developed at the University of Waikato. See [2].

nagman — A module incorporated in Axiom, as part of the mechanism of providing a link to the NAG Foundation Library, which handles communication between Axiom and the numeric server.

noname — A prefix used in IRENA output names to indicate that the name of the object generated should not be displayed in the output list.

Output-function — An RLISP function written to provide additional output jazzing functionality not present in the original IRENA system.

In contrast with the original jazz commands, which only require the names of the NAG and IRENA parameters between which a mapping is being defined, its use requires the provision of a Lisp list as the final syntactic element. This may simply contain the name of an IRENA output parameter but, in some instances, is considerably more complex.

Prompt-alias — A string used by IRENA in prompting, to identify a particular parameter. It may also be used, with spaces optionally replaced by underline characters, as an additional key-alias.

Promptval — An IRENA switch which, if **on**, causes IRENA to prompt for parameter values not otherwise supplied.

PSL — Portable Standard Lisp: the version of Lisp underlying some versions of REDUCE, including that on which IRENA is built. See [36].

Rectangle — A data structure consisting of a list of paired upper and lower bounds, which usually maps into two NAG scalars or one-dimensional arrays.

Routine — A term used by NAG to indicate a Fortran function or subroutine: usually, a function or subroutine occurring in the NAG Library.

Row major order An ordering used in storing the entries in a matrix or other two-dimensional structure, in which complete rows of entries are stored in succession.

Scalar (jazz command) This sets up local scalar variables, emulating extra NAG routine parameters, commonly for communication between the jazz and defaults systems.

Second level defaults The four special symbols `*userabserr*`, `*userrelerr*`, `*usermixerr*` and `*userinputerr*` provide a second level default mechanism, in that they are set globally and used to specify parameter defaults. Their values may be reset at the REDUCE level by the user, thereby redefining default values throughout the system.

Specfile An intermediate file containing routine-specific information, which is derived automatically from NAG documentation, to provide a single target for manual correction or modification, prior to the generation of the infofile and templates.

Template In GENTRAN, a partial program from which a complete program is generated by expanding REDUCE formulae. IRENA uses a Fortran template, defining the program which calls the NAG routine, and a C template, defining the interface between this program and REDUCE, for each included NAG routine.

Unset A special "value" for parameters in IRENA, normally indicating that no Fortran assignments are to be produced for a particular parameter. In the case of ASPs, it may signal that a NAG dummy routine is to be used.

User defaults file A user-supplied file, specific to each IRENA-function, which allows the introduction of additional defaults or the cancellation of the system-supplied defaults.

Vector (jazz command) This sets up local non-scalar variables, emulating extra NAG routine parameters, commonly for communication between the jazz and defaults systems.

Very local constant A symbol, introduced by the IRENA jazz command `local`, which represents a specific input value of a particular NAG parameter. For example, to supply a "very large number" representing *unbounded* in a constrained optimisation routine, the symbol * might be used to represent the quantity `fphuge`.

Bibliography

[1] ACM, Collected Algorithms from ACM, Index by subject to algorithms, 1960-1976.

[2] Broughan, K. A., Naglink – a working symbolic/numeric interface. In *Problem Solving Environments for Scientific Computing*, pp. 343 – 347, Ford, B. & Chatelin, F. eds. North Holland, 1987.

[3] Davenport, J. H., Siret, Y. & Tournier, E., Computer Algebra - Systems and Algorithms for Algebraic Computation. Academic Press, 1988.

[4] Davenport, J. H., Dewar, M. C. & Richardson, M. G., Symbolic and Numeric Computation: the Example of IRENA. In *Symbolic and Numerical Computation for Artificial Intelligence*, pp. 347 – 362, Donald, B. R., Kapur, D. & Mundy, J. L. eds. Academic Press, 1992.

[5] Dewar, M. C., Interfacing Algebraic and Numeric Computation (Ph. D. Thesis). University of Bath, 1991.

[6] Dewar, M. C. & Richardson, M. G., Reconciling Symbolic and Numeric Computation in a Practical Setting. In *Proceedings of DISCO '90*, pp. 171 – 179. Springer-Verlag, 1990.

[7] Dodson, D. S., Grimes, R. G. & Lewis, J. G., Sparse Extensions to the Fortran Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software, 17, pp. 253 – 263, 1991.

[8] Dongarra, J. J., Du Croz, J. J., Duff, I. S. & Hammarling, S., A Set of Level 3 Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software, 16, pp. 1 – 28, 1990.

[9] Dongarra, J. J., Du Croz, J. J., Hammarling, S. & Hanson, R. J., An Extended Set of FORTRAN Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software, 14, pp. 1 – 32, 1988.

[10] Dupée, B. J. & Davenport, J. H., Using Computer Algebra to Choose and Apply Numerical Routines. AXIS 2(1995) 3, pp. 31 - 41.

[11] Francis, B., Green, M. & Payne, C. eds., The GLIM System - Release 4 Manual. Oxford, 1993.

[12] Gates, B. L., GENTRAN: An Automatic Code Generation Facility for REDUCE. SIGSAM Bulletin 19 (1985) 3, pp. 24 - 42.

[13] Gates, B. L., GENTRAN USER'S MANUAL - REDUCE VERSION. Rand, 1987. 19 (1985) 3, pp. 24 - 42.

[14] Hearn, A. C., REDUCE User's Manual Version 3.4. Rand, 1991.

[15] van Hulzen, J. A., SCOPE 1.5 - A Source-Code Optimization PackagE for REDUCE 3.5 - User's Manual. University of Twente, Department of Computer Science, 1995

[16] Hopper, M. J., TSSD, a Typesetting System for Scientific Documents. United Kingdom Atomic Energy Authority, Harwell, Report AERE-R 8574, 1982.

[17] Keady, G. & Richardson, M. G., An application of IRENA to systems of nonlinear equations arising in equilibrium flows in networks. In *Proceedings of ISSAC '93*, pp. 311 - 320, Bronstein M., ed., ACM, 1993.

[18] Lawson, C., Hanson, R., Kincaid, D. & Krough, F., Basic Linear Algebra Subprograms for Fortran Usage. ACM Transactions on Mathematical Software, 5, pp. 308 - 323, 1979.

[19] MacCullum, M. A. H. & Wright, F. J., Algebraic Computing with REDUCE, Oxford. 1991.

[20] Numerical Algorithms Group, The NAG Fortran Library Manual, Mark 11. The Numerical Algorithms Group, Oxford, 1984.

[21] Numerical Algorithms Group, The NAG Fortran Workstation Library Handbook. The Numerical Algorithms Group, Oxford, 1986.

[22] Numerical Algorithms Group, NAG Fortran Library Concise Reference, Mark 15. The Numerical Algorithms Group, Oxford, 1991.

[23] Numerical Algorithms Group, The Essential f77 Tools (Unix). The Numerical Algorithms Group, Oxford, 1992.

[24] Numerical Algorithms Group, NAG Foundation Library Reference Manual. The Numerical Algorithms Group, Oxford, 1992.

[25] Numerical Algorithms Group, NAG Foundation Library Handbook. The Numerical Algorithms Group, Oxford, 1992.

[26] Numerical Algorithms Group, The NAG Fortran Library Manual, Mark 16. The Numerical Algorithms Group, Oxford, 1993.

[27] Numerical Algorithms Group, The NAG Fortran Library Introductory Guide, Mark 16. The Numerical Algorithms Group, Oxford, 1993.

[28] Numerical Algorithms Group, IRENA Installation Note. The Numerical Algorithms Group, Oxford, 1993.

[29] Numerical Algorithms Group, NAG *f*90 Manual Release 1. The Numerical Algorithms Group, Oxford, 1994.

[30] Payne, R. W. et al., Genstat 5 Release 3 Reference Manual. Oxford, 1993.

[31] Rayna, G., REDUCE – Software for Algebraic Computation. Springer-Verlag, 1987.

[32] Richardson. M. G., The IRENA User Interface to the NAG Fortran Library. In *Computer Algebra in Industry – Problem Solving in Practice – Proceedings of the 1991 SCAFI Seminar at CWI, Amsterdam*, pp. 233 – 243, Cohen, A. M., ed., Wiley, 1993.

[33] Richardson. M. G., IRENA User Guide, 3rd Edition. The Numerical Algorithms Group, Oxford, 1994.

[34] Schulze, K. & Cryer, C. W., NAXPERT: a prototype expert system for numerical analysis. Technical Report 7/86 1, Institut für Numerische und Instrumentelle Mathematik, 1986.

[35] Shneiderman, B., Designing the user interface : strategies for effective human-computer interaction (2nd ed.). Addison-Wesley, 1992.

[36] The Utah Symbolic Computation Group, The PSL Users Manual. Department of Computer Science, University of Utah, 1984.