Contents

Preface to Volume 2 viii

Foreword by Michael Metcalf x

License Information xvii

21 Introduction to Fortran 90 Language Features 935

21.0 Introduction 935
21.1 Quick Start: Using the Fortran 90 Numerical Recipes Routines 936
21.2 Fortran 90 Language Concepts 937
21.3 More on Arrays and Array Sections 941
21.4 Fortran 90 Intrinsic Procedures 945
21.5 Advanced Fortran 90 Topics 953
21.6 And Coming Soon: Fortran 95 959

22 Introduction to Parallel Programming 962

22.0 Why Think Parallel? 962
22.1 Fortran 90 Data Parallelism: Arrays and Intrinsics 964
22.2 Linear Recurrence and Related Calculations 971
22.3 Parallel Synthetic Division and Related Algorithms 977
22.4 Fast Fourier Transforms 981
22.5 Missing Language Features 983

23 Numerical Recipes Utility Functions for Fortran 90 987

23.0 Introduction and Summary Listing 987
23.1 Routines That Move Data 990
23.2 Routines Returning a Location 992
23.3 Argument Checking and Error Handling 994
23.4 Routines for Polynomials and Recurrences 996
23.5 Routines for Outer Operations on Vectors 1000
23.6 Routines for Scatter with Combine 1002
23.7 Routines for Skew Operations on Matrices 1004
23.8 Other Routines 1007

Fortran 90 Code Chapters 1009

B1 Preliminaries 1010

B2 Solution of Linear Algebraic Equations 1014

B3 Interpolation and Extrapolation 1043
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B4</td>
<td>Integration of Functions</td>
<td>1052</td>
</tr>
<tr>
<td>B5</td>
<td>Evaluation of Functions</td>
<td>1070</td>
</tr>
<tr>
<td>B6</td>
<td>Special Functions</td>
<td>1083</td>
</tr>
<tr>
<td>B7</td>
<td>Random Numbers</td>
<td>1141</td>
</tr>
<tr>
<td>B8</td>
<td>Sorting</td>
<td>1167</td>
</tr>
<tr>
<td>B9</td>
<td>Root Finding and Nonlinear Sets of Equations</td>
<td>1182</td>
</tr>
<tr>
<td>B10</td>
<td>Minimization or Maximization of Functions</td>
<td>1201</td>
</tr>
<tr>
<td>B11</td>
<td>Eigensystems</td>
<td>1225</td>
</tr>
<tr>
<td>B12</td>
<td>Fast Fourier Transform</td>
<td>1235</td>
</tr>
<tr>
<td>B13</td>
<td>Fourier and Spectral Applications</td>
<td>1253</td>
</tr>
<tr>
<td>B14</td>
<td>Statistical Description of Data</td>
<td>1269</td>
</tr>
<tr>
<td>B15</td>
<td>Modeling of Data</td>
<td>1285</td>
</tr>
<tr>
<td>B16</td>
<td>Integration of Ordinary Differential Equations</td>
<td>1297</td>
</tr>
<tr>
<td>B17</td>
<td>Two Point Boundary Value Problems</td>
<td>1314</td>
</tr>
<tr>
<td>B18</td>
<td>Integral Equations and Inverse Theory</td>
<td>1325</td>
</tr>
<tr>
<td>B19</td>
<td>Partial Differential Equations</td>
<td>1332</td>
</tr>
<tr>
<td>B20</td>
<td>Less-Numerical Algorithms</td>
<td>1343</td>
</tr>
</tbody>
</table>

**References**  
1359

**Appendices**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Listing of Utility Modules (nrtype and nrutil)</td>
<td>1361</td>
</tr>
<tr>
<td>C2</td>
<td>Alphabetical Listing of Explicit Interfaces</td>
<td>1384</td>
</tr>
<tr>
<td>C3</td>
<td>Index of Programs and Dependencies</td>
<td>1434</td>
</tr>
</tbody>
</table>

**General Index to Volumes 1 and 2**  
1447
# Contents of Volume 1: Numerical Recipes in Fortran 77

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Preliminaries</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Solution of Linear Algebraic Equations</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Interpolation and Extrapolation</td>
<td>99</td>
</tr>
<tr>
<td>4</td>
<td>Integration of Functions</td>
<td>123</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation of Functions</td>
<td>159</td>
</tr>
<tr>
<td>6</td>
<td>Special Functions</td>
<td>205</td>
</tr>
<tr>
<td>7</td>
<td>Random Numbers</td>
<td>266</td>
</tr>
<tr>
<td>8</td>
<td>Sorting</td>
<td>320</td>
</tr>
<tr>
<td>9</td>
<td>Root Finding and Nonlinear Sets of Equations</td>
<td>340</td>
</tr>
<tr>
<td>10</td>
<td>Minimization or Maximization of Functions</td>
<td>387</td>
</tr>
<tr>
<td>11</td>
<td>Eigensystems</td>
<td>449</td>
</tr>
<tr>
<td>12</td>
<td>Fast Fourier Transform</td>
<td>490</td>
</tr>
<tr>
<td>13</td>
<td>Fourier and Spectral Applications</td>
<td>530</td>
</tr>
<tr>
<td>14</td>
<td>Statistical Description of Data</td>
<td>603</td>
</tr>
<tr>
<td>15</td>
<td>Modeling of Data</td>
<td>650</td>
</tr>
<tr>
<td>16</td>
<td>Integration of Ordinary Differential Equations</td>
<td>701</td>
</tr>
<tr>
<td>17</td>
<td>Two Point Boundary Value Problems</td>
<td>745</td>
</tr>
<tr>
<td>18</td>
<td>Integral Equations and Inverse Theory</td>
<td>779</td>
</tr>
<tr>
<td>19</td>
<td>Partial Differential Equations</td>
<td>818</td>
</tr>
<tr>
<td>20</td>
<td>Less-Numerical Algorithms</td>
<td>881</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>916</td>
</tr>
<tr>
<td></td>
<td>Index of Programs and Dependencies</td>
<td>921</td>
</tr>
</tbody>
</table>
Preface to Volume 2

Fortran 90 is not just the long-awaited updating of the Fortran language to modern computing practices. It is also the vanguard of a much larger revolution in computing, that of multiprocessor computers and widespread parallel programming. Parallel computing has been a feature of the largest supercomputers for quite some time. Now, however, it is rapidly moving towards the desktop.

As we watched the gestation and birth of Fortran 90 by its governing “X3J3 Committee” (a process interestingly described by a leading committee member, Michael Metcalf, in the Foreword that follows), it became clear to us that the right moment for moving Numerical Recipes from Fortran 77 to Fortran 90 was sooner, rather than later.

Fortran 90 compilers are now widely available. Microsoft’s Fortran PowerStation for Windows 95 brings that firm’s undeniable marketing force to PC desktop; we have tested this compiler thoroughly on our code and found it excellent in compatibility and performance. In the UNIX world, we have similarly tested, and had generally fine experiences with, DEC’s Fortran 90 for Alpha AXP and IBM’s xlf for RS/6000 and similar machines. NAG’s Fortran 90 compiler also brings excellent Fortran 90 compatibility to a variety of UNIX platforms. There are no doubt other excellent compilers, both available and on the way. Fortran 90 is completely backwards compatible with Fortran 77, by the way, so you don’t have to throw away your legacy code, or keep an old compiler around.

There have been previous special versions of Fortran for parallel supercomputers, but always specific to a particular hardware. Fortran 90, by contrast, is designed to provide a general, architecture-independent framework for parallel computation. Equally importantly, it is an international standard, agreed upon by a large group of computer hardware and software manufacturers and international standards bodies.

With the Fortran 90 language as a tool, we want this volume to be your complete guide for learning how to “think parallel.” The language itself is very general in this regard, and applicable to many present and future computers, or even to other parallel computing languages as they come along. Our treatment emphasizes general principles, but we are also not shy about pointing out parallelization “tricks” that have frequent applicability. These are not only discussed in this volume’s principal text chapters (Chapters 21–23), but are also sprinkled throughout the chapters of Fortran 90 code, called out by a special “parallel hint” logo (left, above). Also scattered throughout the code chapters are specific “Fortran 90 tips,” with their own distinct graphic call-out (left). After you read the text chapters, you might want simply to browse among these hints and tips.

A special note to C programmers: Right now, there is no effort at producing a parallel version of C that is comparable to Fortran 90 in maturity, acceptance, and stability. We think, therefore, that C programmers will be well served by using this volume for an educational excursion into Fortran 90, its parallel programming constructions, and the numerical algorithms that capitalize on them. C and C++ programming have not been far from our minds as we have written this volume, and we think that you will find that time spent in absorbing its principal lessons (in Chapters 21–23) will be amply repaid in the future, as C and C++ eventually develop standard parallel extensions.
A final word of truth in packaging: **Don’t buy this volume unless you also buy (or already have) Volume 1** (now retitled *Numerical Recipes in Fortran 77*). Volume 2 does not repeat any of the discussion of what individual programs actually do, or of the mathematical methods they utilize, or how to use them. While our Fortran 90 code is thoroughly commented, and includes a header comment for each routine that describes its input and output quantities, these comments are not supposed to be a complete description of the programs; the complete descriptions are in Volume 1, which we reference frequently. But here’s a money-saving hint to our previous readers: If you already own a Second Edition version whose title is *Numerical Recipes in FORTRAN* (which doesn’t indicate either “Volume 1” or “Volume 2” on its title page) then take a marking pen and write in the words “Volume 1.” There! (Differences between the previous reprints and the newest reprinting, the one labeled “Volume 1,” are minor.)

**Acknowledgments**

We continue to be in the debt of many colleagues who give us the benefit of their numerical and computational experience. Many, though not all, of these are listed by name in the preface to the second edition, in Volume 1. To that list we must now certainly add George Marsaglia, whose ideas have greatly influenced our new discussion of random numbers in this volume (Chapter B7).

With this volume, we must acknowledge our additional gratitude and debt to a number of people who generously provided advice, expertise, and time (a great deal of time, in some cases) in the areas of parallel programming and Fortran 90. The original inspiration for this volume came from Mike Metcalf, whose clear lectures on Fortran 90 (in this case, overlooking the beautiful Adriatic at Trieste) convinced us that Fortran 90 could serve as the vehicle for a book with the larger scope of parallel programming generally, and whose continuing advice throughout the project has been indispensable. Gyan Bhanot also played a vital early role in the development of this book; his first translations of our Fortran 77 programs taught us a lot about parallel programming. We are also grateful to Charles Van Loan, Amos Yahi, Keith Kimball, Malcolm Cohen, Barry Caplin, Loren Meissner, Mitsu Sakamoto, and George Schnurer for helpful correspondence and/or discussion of Fortran 90’s subtler aspects.

We once again express in the strongest terms our gratitude to programming consultant Seth Finkelstein, whose contribution to both the coding and the thorough testing of all the routines in this book (against multiple compilers and in sometimes- buggy, and always challenging, early versions) cannot be overstated.

WHP and SAT acknowledge the continued support of the U.S. National Science Foundation for their research on computational methods.

*February 1996*

William H. Press
Saul A. Teukolsky
William T. Vetterling
Brian P. Flannery
Foreword  
by Michael Metcalf

Sipping coffee on a sunbaked terrace can be surprisingly productive. One of the Numerical Recipes authors and I were each lecturing at the International Center for Theoretical Physics in Trieste, Italy, he on numerical analysis and I on Fortran 90. The numerical analysis community had made important contributions to the development of the new Fortran standard, and so, unsurprisingly, it became quickly apparent that the algorithms for which Numerical Recipes had become renowned could, to great advantage, be recast in a new mold. These algorithms had, hitherto, been expressed in serial form, first in Fortran 77 and then in C, Pascal, and Basic. Now, nested iterations could be replaced by array operations and assignments, and the other features of a rich array language could be exploited. Thus was the idea of a “Numerical Recipes in Fortran 90” first conceived and, after three years’ gestation, it is a delight to assist at the birth.

But what is Fortran 90? How did it begin, what shaped it, and how, after nearly foundering, did its driving forces finally steer it to a successful conclusion?

The Birth of a Standard

Back in 1966, the version of Fortran now known as Fortran 66 was the first language ever to be standardized, by the predecessor of the present American National Standards Institute (ANSI). It was an all-American affair. Fortran had first been developed by John Backus of IBM in New York, and it was the dominant scientific programming language in North America. Many Europeans preferred Algol (in which Backus had also had a hand). Eventually, however, the mathematicians who favored Algol for its precisely expressible syntax began to defer to the scientists and engineers who appreciated Fortran’s pragmatic, even natural, style. In 1978, the upgraded Fortran 77 was standardized by the ANSI technical committee, X3J3, and subsequently endorsed by other national bodies and by ISO in Geneva, Switzerland. Its dominance in all fields of scientific and numerical computing grew as new, highly optimizing compilers came onto the market. Although newer languages, particularly Pascal, Basic, PL/1, and later Ada attracted their own adherents, scientific users throughout the 1980s remained true to Fortran. Only towards the end of that decade did C draw increasing support from scientific programmers who had discovered the power of structures and pointers.

During all this time, X3J3 kept functioning, developing the successor version to Fortran 77. It was to be a decade of strife and contention. The early plans, in the late 1970s, were mainly to add to Fortran 77 features that had had to be left out of that standard. Among these were dynamic storage and an array language, enabling it to map directly onto the architecture of supercomputers, then coming onto the market. The intention was to have this new version ready within five years, in 1982. But two new factors became significant at that time. The first was the decision that the next standard should not just codify existing practice, as had largely been the case in 1966 and 1978, but also extend the functionality of the language through...
innovative additions (even though, for the array language, there was significant borrowing from John Iverson’s APL and from DAP Fortran). The second factor was that X3J3 no longer operated under only American auspices. In the course of the 1980s, the standardization of programming languages came increasingly under the authority of the international body, ISO. Initially this was in an advisory role, but now ISO is the body that, through its technical committee WG5 (in full, ISO/IEC JTC1/SC22/WG5), is responsible for determining the course of the language. WG5 also steers the work of the development body, then as now, the highly skilled and competent X3J3. As we shall see, this shift in authority was crucial at the most difficult moment of Fortran 90’s development.

The internationalization of the standards effort was reflected in the welcome given by X3J3 to six or seven European members; they, and about one-third of X3J3’s U.S. members, provided the overlapping core of membership of X3J3 and WG5 that was vital in the final years in bringing the work to a successful conclusion. X3J3 membership, which peaked at about 45, is restricted to one voting member per organization, and significant decisions require a majority of two-thirds of those voting. Nationality plays no role, except in determining the U.S. position on an international issue. Members, who are drawn mainly from the vendors, large research laboratories, and academia, must be present or represented at two-thirds of all meetings in order to retain voting rights.

In 1980, X3J3 reported on its plans to the forerunner of WG5 in Amsterdam, Holland. Fortran 8x, as it was dubbed, was to have a basic array language, new looping constructs, a bit data type, data structures, a free source form, a mechanism to “group” procedures, and another to manage the global name space. Old features, including COMMON, EQUIVALENCE, and the arithmetic-IF, were to be consigned to a so-called obsolete module, destined to disappear in a subsequent revision. This was part of the “core plus modules” architecture, for adding new features and retiring old ones, an aid to backwards compatibility. Even though Fortran 77 compilers were barely available, the work seemed well advanced and the mood was optimistic. Publication was intended to take place in 1985. It was not to be.

One problem was the sheer number of new features that were proposed as additions to the language, most of them worthwhile in themselves but with the totality being too large. This became a recurrent theme throughout the development of the standard. One example was the suggestion of Lawrie Schonfelder (Liverpool University), at a WG5 meeting in Vienna, Austria, in 1982, that certain features already proposed as additions could be combined to provide a full-blown derived data type facility, thus providing Fortran with abstract data types. This idea was taken up by X3J3 and has since come to be recognized, along with the array language, as one of the two main advances brought about by what became Fortran 90. However, the ramifications go very deep: all the technical details of how to handle arrays of objects of derived types that in turn have array components that have the pointer attribute, and so forth, have to be precisely defined and rigorously specified.

Conflict

The meetings of X3J3 were often full of drama. Most compiler vendors were represented as a matter of course but, for many, their main objective appeared to be to maintain the status quo and to ensure that Fortran 90 never saw the light of
day. One vendor’s extended (and much-copied) version of Fortran 77 had virtually become an industry standard, and it saw as its mission the maintenance of this lead. A new standard would cost it its perceived precious advantage. Other large vendors had similar points of view, although those marketing supercomputers were clearly keen on the array language. Most users, on the other hand, were hardly prepared to invest large amounts of their employers’ and their own resources in simply settling for a trivial set of improvements to the existing standard. However, as long as X3J3 worked under a simple-majority voting rule, at least some apparent progress could be made, although the underlying differences often surfaced. These were even sometimes between users — those who wanted Fortran to become a truly modern language and those wanting to maintain indefinite backwards compatibility for their billions of lines of existing code.

At a watershed meeting, in Scranton, Pennsylvania, in 1986, held in an atmosphere that sometimes verged on despair, a fragile compromise was reached as a basis for further work. One breakthrough was to weaken the procedures for removing outdated features from the language, particularly by removing no features whatsoever from the next standard and by striking storage association (i.e., COMMON and EQUIVALENCE) from the list of features to be designated as obsolescent (as they are now known). A series of votes definitively removed from the language all plans to add: arrays of arrays, exception handling, nesting of internal procedures, the FORALL statement (now in Fortran 95), and a means to access skew array sections. There were other features on this list that, although removed, were reinstated at later meetings: user-defined operators, operator overloading, array and structure constructors, and vector-valued subscripts. After many more travails, the committee voted, a year later, by 26 votes to 9, to forward the document for what was to become the first of three periods of public comment.

While the document was going through the formal standards bureaucracy and being placed before the public, X3J3 polished it further. X3J3 also prepared procedures for processing the comments it anticipated receiving from the public, and to each of which, under the rules, it would have to reply individually. It was just as well. Roughly 400 replies flooded in, many of them very detailed and, disappointingly for those of us wanting a new standard quickly, unquestionably negative towards our work. For many it was too radical, but many others pleaded for yet more modern features, such as pointers.

Now the committee was deadlocked. Given that a document had already been published, any further change required not a simple but a two-thirds majority. The conservatives and the radicals could each block a move to modify the draft standard, or to accept a revised one for public review — and just that happened, in Champagne-Urbana, Illinois, in 1988. Any change, be it on the one hand to modify the list of obsolescent features, to add the pointers or bit data type wanted by the public, to add multi-byte characters to support Kanji and other non-European languages or, on the other hand, to emasculate the language by removing modules or operator overloading, and hence abstract data types, to name but some suggestions, none of these could be done individually or collectively in a way that would achieve consensus. I wrote:

“In my opinion, no standard can now emerge without either a huge concession by the users to the vendors (MODULE / USE) and/or a major change in the composition of the committee. I do not see how members who have worked for up to a decade or more, devoting time and intellectual energy far beyond the call of duty, can be
expected to make yet more personal sacrifices if no end to the work is in sight, or if that end is nothing but a travesty of what had been designed and intended as a modern scientific programming language. ... I think the August meeting will be a watershed — if no progress is achieved there will be dramatic resignations, and ISO could even remove the work from ANSI, which is failing conspicuously in its task.”

(However, the same notes began with a quotation from \textit{The Taming of the Shrew}: “And do as adversaries do in law, / Strive mightily, but eat and drink / as friend.” That we always did, copiously.)

\section*{Resolution}

The “August meeting” was, unexpectedly, imbued with a spirit of compromise that had been so sadly lacking at the previous one. Nevertheless, after a week of discussing four separate plans to rescue the standard, no agreement was reached. Now the question seriously arose: Was X3J3 incapable of producing a new Fortran standard for the international community, doomed to eternal deadlock, a victim of ANSI procedures?

Breakthrough was achieved at a traumatic meeting of WG5 in Paris, France, a month later. The committee spent several extraordinary days drawing up a detailed list of what \textit{it} wanted to be in Fortran 8x. Finally, it set X3J3 an ultimatum that was unprecedented in the standards world: The ANSI committee was to produce a new draft document, corresponding to WG5’s wishes, within five months! Failing that, WG5 would assume responsibility and produce the new standard itself.

This decision was backed by the senior U.S. committee, X3, which effectively directed X3J3 to carry out WG5’s wishes. And it did! The following November, it implemented most of the technical changes, adding pointers, bit manipulation intrinsic procedures, and vector-valued subscripts, and removing user-defined elemental functions (now in Fortran 95). The actual list of changes was much longer. X3J3 and WG5, now collaborating closely, often in gruelling six-day meetings, spent the next 18 months and two more periods of (positive) public comment putting the finishing touches to what was now called Fortran 90, and it was finally adopted, after some cliff-hanging votes, for forwarding as a U.S. and international standard on April 11, 1991, in Minneapolis, Minnesota.

Among the remaining issues that were decided along the way were whether pointers should be a data type or be defined in terms of an attribute of a variable, implying strong typing (the latter was chosen), whether the new standard should coexist alongside the old one rather than definitively replace it (it coexisted for a while in the U.S., but was a replacement elsewhere, under ISO rules), and whether, in the new free source form, blanks should be significant (fortunately, they are).

\section*{Fortran 90}

The main new features of Fortran 90 are, first and foremost, the array language and abstract data types. The first is built on whole array operations and assignments, array sections, intrinsic procedures for arrays, and dynamic storage. It was designed with optimization in mind. The second is built on modules and module procedures, derived data types, operator overloading and generic interfaces, together with pointers. Also important are the new facilities for numerical computation including
a set of numeric inquiry functions, the parametrization of the intrinsic types, new control constructs — SELECT CASE and new forms of DO, internal and recursive procedures and optional and keyword arguments, improved I/O facilities, and many new intrinsic procedures. Last but not least are the new free source form, an improved style of attribute-oriented specifications, the IMPLICIT NONE statement, and a mechanism for identifying redundant features for subsequent removal from the language. The requirement on compilers to be able to identify, for example, syntax extensions, and to report why a program has been rejected, are also significant. The resulting language is not only a far more powerful tool than its successor, but a safer and more reliable one too. Storage association, with its attendant dangers, is not abolished, but rendered unnecessary. Indeed, experience shows that compilers detect errors far more frequently than before, resulting in a faster development cycle. The array syntax and recursion also allow quite compact code to be written, a further aid to safe programming.

No programming language can succeed if it consists simply of a definition (witness Algol 68). Also required are robust compilers from a wide variety of vendors, documentation at various levels, and a body of experience. The first Fortran 90 compiler appeared surprisingly quickly, in 1991, especially in view of the widely touted opinion that it would be very difficult to write one. Even more remarkable was that it was written by one person, Malcolm Cohen of NAG, in Oxford, U.K. There was a gap before other compilers appeared, but now they exist as native implementations for almost all leading computers, from the largest to PCs. For the most part, they produce very efficient object code; where, for certain new features, this is not the case, work is in progress to improve them.

The first book, Fortran 90 Explained, was published by John Reid and me shortly before the standard itself was published. Others followed in quick succession, including excellent texts aimed at the college market. At the time of writing there are at least 19 books in English and 22 in various other languages: Chinese, Dutch, French, Japanese, Russian, and Swedish. Thus, the documentation condition is fulfilled.

The body of experience, on the other hand, has yet to be built up to a critical size. Teaching of the language at college level has only just begun. However, I am certain that this present volume will contribute decisively to a significant breakthrough, as it provides models not only of the numerical algorithms for which previous editions are already famed, but also of an excellent Fortran 90 style, something that can develop only with time. Redundant features are abjured. It shows that, if we abandon these features and use new ones in their place, the appearance of code can initially seem unfamiliar, but, in fact, the advantages become rapidly apparent. This new edition of Numerical Recipes stands as a landmark in this regard.

**Fortran Evolution**

The formal procedures under which languages are standardized require them either to evolve or to die. A standard that has not been revised for some years must either be revised and approved anew, or be withdrawn. This matches the technical pressure on the language developers to accommodate the increasing complexity both of the problems to be tackled in scientific computation and of the underlying hardware on which programs run. Increasing problem complexity requires more powerful
features and syntax; new hardware needs language features that map onto it well.

Thus it was that X3J3 and WG5, having finished Fortran 90, began a new round of improvement. They decided very quickly on new procedures that would avoid the disputes that bedevilled the previous work: WG5 would decide on a plan for future standards, and X3J3 would act as the so-called development body that would actually produce them. This would be done to a strict timetable, such that any feature that could not be completed on time would have to wait for the next round. It was further decided that the next major revision should appear a decade after Fortran 90 but, given the somewhat discomforting number of requests for interpretation that had arrived, about 200, that a minor revision should be prepared for mid-term, in 1995. This should contain only “corrections, clarifications and interpretations” and a very limited number (some thought none) of minor improvements.

At the same time, scientific programmers were becoming increasingly concerned at the variety of methods that were necessary to gain efficient performance from the ever-more widely used parallel architectures. Each vendor provided a different set of parallel extensions for Fortran, and some academic researchers had developed yet others. On the initiative of Ken Kennedy of Rice University, a High-Performance Fortran Forum was established. A coalition of vendors and users, its aim was to produce an ad hoc set of extensions to Fortran that would become an informal but widely accepted standard for portable code. It set itself the daunting task of achieving that in just one year, and succeeded. Melding existing dialects like Fortran D, CM Fortran, and Vienna Fortran, and adopting the new Fortran 90 as a base, because of its array syntax, High-Performance Fortran (HPF) was published in 1993 and has since become widely implemented. However, although HPF was designed for data parallel codes and mainly implemented in the form of directives that appear to non-HPF processors as comment lines, an adequate functionality could not be achieved without extending the Fortran syntax. This was done in the form of the PURE attribute for functions — an assertion that they contain no side effects — and the FORALL construct — a form of array assignment expressed with the help of indices.

The dangers of having diverging or competing forms of Fortran 90 were immediately apparent, and the standards committees wisely decided to incorporate these two syntactic changes also into Fortran 95. But they didn’t stop there. Two further extensions, useful not only for their expressive power but also to access parallel hardware, were added: elemental functions, ones written in terms of scalars but that accept array arguments of any permitted shape or size, and an extension to allow nesting of WHERE constructs, Fortran’s form of masked assignment. To readers of Numerical Recipes, perhaps the most relevant of the minor improvements that Fortran 95 brings are the ability to distinguish between a negative and a positive real zero, automatic deallocation of allocatable arrays, and a means to initialize the values of components of objects of derived data types and to initialize pointers to null.

The medium-term objective of a relatively minor upgrade has been achieved on schedule. But what does the future hold? Developments in the underlying principles of procedural programming languages have not ceased. Early Fortran introduced the concepts of expression abstraction (X=Y+Z) and later control expression (e.g., the DO loop). Fortran 77 continued this with the if-then-else, and Fortran 90 with the DO and SELECT CASE constructs. Fortran 90 has a still higher level of expression abstraction (array assignments and expressions) as well as data structures and even full-blown abstract data types. However, during the 1980s the concept of objects
came to the fore, with methods bound to the objects on which they operate. Here, one particular language, C++, has come to dominate the field. Fortran 90 lacks a means to point to functions, but otherwise has most of the necessary features in place, and the standards committees are now faced with the dilemma of deciding whether to make the planned Fortran 2000 a fully object-oriented language. This could possibly jeopardise its powerful, and efficient, numerical capabilities by too great an increase in language complexity, so should they simply batten down the hatches and not defer to what might be only a passing storm? At the time of writing, this is an open issue. One issue that is not open is Fortran’s lack of in-built exception handling. It is virtually certain that such a facility, much requested by the numerical community, and guided by John Reid, will be part of the next major revision. The list of other requirements is long but speculative, but some at the top of the list are conditional compilation, command line argument handling, I/O for objects of derived type, and asynchronous I/O (which is also planned for the next release of HPF). In the meantime, some particularly pressing needs have been identified, for the handling of floating-point exceptions, interoperability with C, and allowing allocatable arrays as structure components, dummy arguments, and function results. These have led WG5 to begin processing these three items using a special form of fast track, so that they might become optional but standard extensions well before Fortran 2000 itself is published in the year 2001.

Conclusion

Writing a book is always something of a gamble. Unlike a novel that stands or falls on its own, a book devoted to a programming language is dependent on the success of others, and so the risk is greater still. However, this new Numerical Recipes in Fortran 90 volume is no ordinary book, since it comes as the continuation of a highly successful series, and so great is its significance that it can, in fact, influence the outcome in its own favor. I am entirely confident that its publication will be seen as an important event in the story of Fortran 90, and congratulate its authors on having performed a great service to the field of numerical computing.

Geneva, Switzerland
January 1996
License Information

Read this section if you want to use the programs in this book on a computer. You’ll need to read the following Disclaimer of Warranty, get the programs onto your computer, and acquire a Numerical Recipes software license. (Without this license, which can be the free “immediate license” under terms described below, the book is intended as a text and reference book, for reading purposes only.)

Disclaimer of Warranty

We make no warranties, express or implied, that the programs contained in this volume are free of error, or are consistent with any particular standard of merchantability, or that they will meet your requirements for any particular application. They should not be relied on for solving a problem whose incorrect solution could result in injury to a person or loss of property. If you do use the programs in such a manner, it is at your own risk. The authors and publisher disclaim all liability for direct or consequential damages resulting from your use of the programs.

How to Get the Code onto Your Computer

Pick one of the following methods:

- You can type the programs from this book directly into your computer. In this case, the only kind of license available to you is the free “immediate license” (see below). You are not authorized to transfer or distribute a machine-readable copy to any other person, nor to have any other person type the programs into a computer on your behalf. We do not want to hear bug reports from you if you choose this option, because experience has shown that virtually all reported bugs in such cases are typing errors!

- You can download the Numerical Recipes programs electronically from the Numerical Recipes On-Line Software Store, located at our Web site (http://www.nr.com). They are packaged as a single compressed file, and you’ll need to purchase a license to download and unpack them. Any number of single-screen licenses can be purchased instantly (with discount for multiple screens) from the On-Line Store, with fees that depend on your operating system (Windows or Macintosh versus Linux or UNIX) and whether you are affiliated with an educational institution. Purchasing a single-screen license is also the way to start if you want to acquire a more general (site or corporate) license; your single-screen cost will be subtracted from the cost of any later license upgrade.

- You can purchase media containing the programs from Cambridge University Press. A diskette version is available for Windows machines. CDROM versions in ISO-9660 format for Windows, Macintosh, and UNIX/Linux systems are also available; these include Fortran, C, and C++ versions on a single CDROM (as well as versions in Pascal and BASIC from the first edition). Diskettes purchased from Cambridge University Press include...
License Information

a single-screen license for Windows only. The CDROM is available with a single-screen license for Windows or Macintosh (order ISBN 0 521 750350), or (at a slightly higher price) with a single-screen license for UNIX/Linux workstations (order ISBN 0 521 750369). Orders for media from Cambridge University Press can be placed at 800 872-7423 (North America only) or by email to orders@cup.org (North America) or directcustserv@cambridge.org (rest of world). Or, visit the Web site http://www.cambridge.org.

Types of License Offered

Here are the types of licenses that we offer. Note that some types are automatically acquired with the purchase of media from Cambridge University Press, or of an unlocking password from the Numerical Recipes On-Line Software Store, while other types of licenses require that you communicate specifically with Numerical Recipes Software (email: orders@nr.com or fax: 781 863-1739). Our Web site http://www.nr.com has additional information.

- [“Immediate License”] If you are the individual owner of a copy of this book and you type one or more of its routines into your computer, we authorize you to use them on that computer for your own personal and noncommercial purposes. You are not authorized to transfer or distribute machine-readable copies to any other person, or to use the routines on more than one machine, or to distribute executable programs containing our routines. This is the only free license.

- [“Single-Screen License”] This is the most common type of low-cost license, with terms governed by our Single Screen (Shrinkwrap) License document (complete terms available through our Web site). Basically, this license lets you use Numerical Recipes routines on any one screen (PC, workstation, X-terminal, etc.). You may also, under this license, transfer pre-compiled, executable programs incorporating our routines to other, unlicensed, screens or computers, providing that (i) your application is noncommercial (i.e., does not involve the selling of your program for a fee), (ii) the programs were first developed, compiled, and successfully run on a licensed screen, and (iii) our routines are bound into the programs in such a manner that they cannot be accessed as individual routines and cannot practicably be unbound and used in other programs. That is, under this license, your program user must not be able to use our programs as part of a program library or “mix-and-match” workbench. Conditions for other types of commercial or noncommercial distribution may be found on our Web site (http://www.nr.com).

- [“Multi-Screen, Server, Site, and Corporate Licenses”] The terms of the Single Screen License can be extended to designated groups of machines, defined by number of screens, number of machines, locations, or ownership. Significant discounts from the corresponding single-screen...
prices are available when the estimated number of screens exceeds 40. Contact Numerical Recipes Software (email: orders@nr.com or fax: 781 863-1739) for details.

- [“Course Right-to-Copy License”] Instructors at accredited educational institutions who have adopted this book for a course, and who have already purchased a Single Screen License (either acquired with the purchase of media, or from the Numerical Recipes On-Line Software Store), may license the programs for use in that course as follows: Mail your name, title, and address; the course name, number, dates, and estimated enrollment; and advance payment of $5 per (estimated) student to Numerical Recipes Software, at this address: P.O. Box 243, Cambridge, MA 02238 (USA). You will receive by return mail a license authorizing you to make copies of the programs for use by your students, and/or to transfer the programs to a machine accessible to your students (but only for the duration of the course).

About Copyrights on Computer Programs

Like artistic or literary compositions, computer programs are protected by copyright. Generally it is an infringement for you to copy into your computer a program from a copyrighted source. (It is also not a friendly thing to do, since it deprives the program's author of compensation for his or her creative effort.) Under copyright law, all “derivative works” (modified versions, or translations into another computer language) also come under the same copyright as the original work.

Copyright does not protect ideas, but only the expression of those ideas in a particular form. In the case of a computer program, the ideas consist of the program’s methodology and algorithm, including the necessary sequence of steps adopted by the programmer. The expression of those ideas is the program source code (particularly any arbitrary or stylistic choices embodied in it), its derived object code, and any other derivative works.

If you analyze the ideas contained in a program, and then express those ideas in your own completely different implementation, then that new program implementation belongs to you. That is what we have done for those programs in this book that are not entirely of our own devising. When programs in this book are said to be “based” on programs published in copyright sources, we mean that the ideas are the same. The expression of these ideas as source code is our own. We believe that no material in this book infringes on an existing copyright.

Trademarks

Several registered trademarks appear within the text of this book: Sun is a trademark of Sun Microsystems, Inc. SPARC and SPARCstation are trademarks of SPARC International, Inc. Microsoft, Windows 95, Windows NT, PowerStation, and MS are trademarks of Microsoft Corporation. DEC, VMS, Alpha AXP, and ULTRIX are trademarks of Digital Equipment Corporation. IBM is a trademark of International Business Machines Corporation. Apple and Macintosh are trademarks of Apple Computer, Inc. UNIX is a trademark licensed exclusively through X/Open.
Co. Ltd. IMSL is a trademark of Visual Numerics, Inc. NAG refers to proprietary computer software of Numerical Algorithms Group (USA) Inc. PostScript and Adobe Illustrator are trademarks of Adobe Systems Incorporated. Last, and no doubt least, Numerical Recipes (when identifying products) is a trademark of Numerical Recipes Software.

Attributions

The fact that ideas are legally “free as air” in no way supersedes the ethical requirement that ideas be credited to their known originators. When programs in this book are based on known sources, whether copyrighted or in the public domain, published or “handed-down,” we have attempted to give proper attribution. Unfortunately, the lineage of many programs in common circulation is often unclear. We would be grateful to readers for new or corrected information regarding attributions, which we will attempt to incorporate in subsequent printings.
Chapter 21. Introduction to Fortran 90 Language Features

21.0 Introduction

Fortran 90 is in many respects a backwards-compatible modernization of the long-used (and much abused) Fortran 77 language, but it is also, in other respects, a new language for parallel programming on present and future multiprocessor machines. These twin design goals of the language sometimes add confusion to the process of becoming fluent in Fortran 90 programming.

In a certain trivial sense, Fortran 90 is strictly backwards-compatible with Fortran 77. That is, any Fortran 90 compiler is supposed to be able to compile any legacy Fortran 77 code without error. The reason for terming this compatibility trivial, however, is that you have to tell the compiler (usually via a source file name ending in “.f” or “.for”) that it is dealing with a Fortran 77 file. If you instead try to pass off Fortran 77 code as native Fortran 90 (e.g., by naming the source file something ending in “.f90”) it will not always work correctly!

It is best, therefore, to approach Fortran 90 as a new computer language, albeit one with a lot in common with Fortran 77. Indeed, in such terms, Fortran 90 is a fairly big language, with a large number of new constructions and intrinsic functions. Here, in one short chapter, we do not pretend to provide a complete description of the language. Luckily, there are good books that do exactly that. Our favorite one is by Metcalf and Reid [1], cited throughout this chapter as “M&R.” Other good starting points include [2] and [3].

Our goal, in the remainder of this chapter, is to give a good, working description of those Fortran 90 language features that are not immediately self-explanatory to Fortran 77 programmers, with particular emphasis on those that occur most frequently in the Fortran 90 versions of the Numerical Recipes routines. This chapter, by itself, will not teach you to write Fortran 90 code. But it ought to help you acquire a reading knowledge of the language, and perhaps provide enough of a head start that you can rapidly pick up the rest of what you need to know from M&R or another Fortran 90 reference book.

CITED REFERENCES AND FURTHER READING:
21.1 Quick Start: Using the Fortran 90 Numerical Recipes Routines

This section is for people who want to jump right in. We’ll compute a Bessel function \( J_0(x) \), where \( x \) is equal to the fourth root of the Julian Day number of the 200th full moon since January 1900. (Now there’s a useful quantity!)

First, locate the important files `nrtype.f90`, `nrutil.f90`, and `nr.f90`, as listed in Appendices C1, C1, and C2, respectively. These contain modules that either are (i) used by our routines, or else (ii) describe the calling conventions of our routines to (your) user programs. Compile each of these files, producing (with most compilers) a .mod file and a .o (or similarly named) file for each one.

Second, create this main program file:

```fortran
PROGRAM hello_bessel
USE nrtype
USE nr, ONLY: flmoon, bessj0
IMPLICIT NONE
INTEGER(I4B) :: n=200, nph=2, jd
REAL(SP) :: x, frac, ans
call flmoon(n, nph, jd, frac)
x=jd**0.25_sp
ans=bessj0(x)
write (*,*) 'Hello, Bessel: ', ans
END PROGRAM
```

Here is a quick explanation of some elements of the above program:

The first USE statement includes a module of ours named `nrtype`, whose purpose is to give symbolic names to some kinds of data types, among them single-precision reals ("sp") and four-byte integers ("i4b"). The second USE statement includes a module of ours that defines the calling sequences, and variable types, expected by (in this case) the Numerical Recipes routines `flmoon` and `bessj0`.

The IMPLICIT NONE statement signals that we want the compiler to require us explicitly to declare all variable types. We strongly urge that you always take this option.

The next two lines declare integer and real variables of the desired kinds. The variable \( n \) is initialized to the value 200, \( nph \) to 2 (a value expected by `flmoon`).

We call `flmoon`, and take the fourth root of the answer it returns as \( jd \). Note that the constant 0.25 is typed to be single-precision by the appended _sp.

We call the `bessj0` routine, and print the answer.

Third, compile the main program file, and also the files `flmoon.f90`, `bessj0.f90`. Then, link the resulting object files with also `nrutil.o` (or similar system-dependent name, as produced in step 1). Some compilers will also require you to link with `nr.o` and `nrtype.o`.

Fourth, run the resulting executable file. Typical output is:

```
Hello, Bessel: 7.3096365E-02
```
21.2 Fortran 90 Language Concepts

The Fortran 90 language standard defines and uses a number of standard terms for concepts that occur in the language. Here we summarize briefly some of the most important concepts. Standard Fortran 90 terms are shown in italics. While by no means complete, the information in this section should help you get a quick start with your favorite Fortran 90 reference book or language manual.

A note on capitalization: Outside a character context, Fortran 90 is not case-sensitive, so you can use upper and lower case any way you want, to improve readability. A variable like sp (see below) is the same variable as the variable SP. We like to capitalize keywords whose use is primarily at compile-time (statements that delimit program and subprogram boundaries, declaration statements of variables, fixed parameter values), and use lower case for the bulk of run-time code. You can adopt any convention that you find helpful to your own programming style; but we strongly urge you to adopt and follow some convention.

Data Types and Kinds

Data types (also called simply types) can be either intrinsic data types (the familiar INTEGER, REAL, LOGICAL, and so forth) or else derived data types that are built up in the manner of what are called “structures” or “records” in other computer languages. (We’ll use derived data types very sparingly in this book.) Intrinsic data types are further specified by their kind parameter (or simply kind), which is simply an integer. Thus, on many machines, REAL(4) (with kind = 4) is a single-precision real, while REAL(8) (with kind = 8) is a double-precision real. Literal constants (or simply literals) are specified as to kind by appending an underscore, as 1.5_4 for single precision, or 1.5_8 for double precision. [M&R, §2.5–§2.6]

Unfortunately, the specific integer values that define the different kind types are not specified by the language, but can vary from machine to machine. For that reason, one almost never uses literal kind parameters like 4 or 8, but rather defines in some central file, and imports into all one’s programs, symbolic names for the kinds. For this book, that central file is the module named nrtype, and the chosen symbolic names include SP, DP (for reals); I2B, I4B (for two- and four-byte integers); and LGT for the default logical type. You will therefore see us consistently writing REAL(SP), or 1.5_sp, and so forth.

Here is an example of declaring some variables, including a one-dimensional array of length 500, and a two-dimensional array with 100 rows and 200 columns:

```fortran
USE nrtype
REAL(SP) :: x,y,z
INTEGER(I4B) :: i,j,k
REAL(SP), DIMENSION(500) :: arr
REAL(SP), DIMENSION(100,200) :: barr
REAL(SP) :: carr(500)
```

The last line shows an alternative form for array syntax. And yes, there are default kind parameters for each intrinsic type, but these vary from machine to machine and can get you into trouble when you try to move code. We therefore specify all kind parameters explicitly in almost all situations.
Array Shapes and Sizes

The shape of an array refers to both its dimensionality (called its rank), and also the lengths along each dimension (called the extents). The shape of an array is specified by a rank-one array whose elements are the extents along each dimension, and can be queried with the shape intrinsic (see p. 949). Thus, in the above example, shape(barr) returns an array of length 2 containing the values (100, 200).

The size of an array is its total number of elements, so the intrinsic size(barr) would return 20000 in the above example. More often one wants to know the extents along each dimension, separately: size(barr,1) returns the value 100, while size(barr,2) returns the value 200. [M&R, §2.10]

Section §21.3, below, discusses additional aspects of arrays in Fortran 90.

Memory Management

Fortran 90 is greatly superior to Fortran 77 in its memory-management capabilities, seen by the user as the ability to create, expand, or contract workspace for programs. Within subprograms (that is, subroutines and functions), one can have automatic arrays (or other automatic data objects) that come into existence each time the subprogram is entered, and disappear (returning their memory to the pool) when the subprogram is exited. The size of automatic objects can be specified by arbitrary expressions involving values passed as actual arguments in the calling program, and thus received by the subprogram through its corresponding dummy arguments. [M&R, §6.4]

Here is an example that creates some automatic workspace named carr:

```fortran
SUBROUTINE dosomething(j,k)
  USE nrtype
  REAL(SP), DIMENSION(2*j,k**2) :: carr
END SUBROUTINE dosomething
```

Finer control on when workspace is created or destroyed can be achieved by declaring allocatable arrays, which exist as names only, without associated memory, until they are allocated within the program or subprogram. When no longer needed, they can be deallocated. The allocation status of an allocatable array can be tested by the program via the allocated intrinsic function (p. 952). [M&R, §6.5]

Here is an example in outline:

```fortran
REAL(SP), DIMENSION(:,,:), ALLOCATABLE :: darr
...
allocate(darr(10,20))
...
dedallocate(darr)
...
allocate(darr(100,200))
...
dedallocate(darr)
```

Notice that darr is originally declared with only “slots” (colons) for its dimensions, and is then allocated/deallocated twice, with different sizes.

Yet finer control is achieved by the use of pointers. Like an allocatable array, a pointer can be allocated, at will, its own associated memory. However, it has the additional flexibility of alternatively being pointer associated with a target that
already exists under another name. Thus, pointers can be used as redefinable aliases for other variables, arrays, or (see §21.3) array sections. [M&R, §6.12]

Here is an example that first associates the pointer `parr` with the array `earr`, then later cancels that association and allocates it its own storage of size 50:

```fortran
REAL(SP), DIMENSION(:,), POINTER :: parr
REAL(SP), DIMENSION(100), TARGET :: earr
...
parr => earr
...
nullify(parr)
allocate(parr(50))
...
deallocate(parr)
```

### Procedure Interfaces

When a procedure is referenced (e.g., called) from within a program or subprogram (examples of scoping units), the scoping unit must be told, or must deduce, the procedure’s interface, that is, its calling sequence, including the types and kinds of all dummy arguments, returned values, etc. The recommended procedure is to specify this interface via an explicit interface, usually an interface block (essentially a declaration statement for subprograms) in the calling subprogram or in some module that the calling program includes via a USE statement. In this book all interfaces are explicit, and the module named `nr` contains interface blocks for all of the Numerical Recipes routines. [M&R, §5.11]

Here is a typical example of an interface block:

```fortran
INTERFACE
  SUBROUTINE caldat(julian, mm, id, iyyy)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: julian
    INTEGER(I4B), INTENT(OUT) :: mm, id, iyyy
  END SUBROUTINE caldat
END INTERFACE
```

Once this interface is made known to a program that you are writing (by either explicit inclusion or a USE statement), then the compiler is able to flag for you a variety of otherwise difficult-to-find bugs. Although interface blocks can sometimes seem overly wordy, they give a big payoff in ultimately minimizing programmer time and frustration.

For compatibility with Fortran 77, the language also allows for implicit interfaces, where the calling program tries to figure out the interface by the old rules of Fortran 77. These rules are quite limited, and prone to producing devilishly obscure program bugs. We strongly recommend that implicit interfaces never be used.

### Elemental Procedures and Generic Interfaces

Many intrinsic procedures (those defined by the language standard and thus usable without any further definition or specification) are also generic. This means that a single procedure name, such as `log(x)`, can be used with a variety of types and kind parameters for the argument `x`, and the result returned will have the same type and kind parameter as the argument. In this example, `log(x)` allows any real or complex argument type.
Better yet, most generic functions are also *elemental*. The argument of an elemental function can be an array of arbitrary shape! Then, the returned result is an array of the same shape, with each element containing the result of applying the function to the corresponding element of the argument. (Hence the name *elemental*, meaning “applied element by element.”) [M&R, §8.1] For example:

```fortran
REAL(SP), DIMENSION(100,100) :: a,b
b=sin(a)
```

Fortran 90 has no facility for creating new, user-defined elemental functions. It does have, however, the related facility of *overloading* by the use of *generic interfaces*. This is invoked by the use of an interface block that attaches a single *generic name* to a number of distinct subprograms whose dummy arguments have different types or kinds. Then, when the generic name is referenced (e.g., called), the compiler chooses the specific subprogram that matches the types and kinds of the actual arguments used. [M&R, §5.18] Here is an example of a generic interface block:

```fortran
INTERFACE myfunc
    FUNCTION myfunc_single(x)
        USE nrtype
        REAL(SP) :: x,myfunc_single
    END FUNCTION myfunc_single

    FUNCTION myfunc_double(x)
        USE nrtype
        REAL(DP) :: x,myfunc_double
    END FUNCTION myfunc_double
END INTERFACE
```

A program with knowledge of this interface could then freely use the function reference `myfunc(x)` for `x`’s of both type SP and type DP.

We use overloading quite extensively in this book. A typical use is to provide, under the same name, both scalar and vector versions of a function such as a Bessel function, or to provide both single-precision and double-precision versions of procedures (as in the above example). Then, to the extent that we have provided all the versions that you need, you can pretend that our routine is elemental. In such a situation, if you ever call our function with a type or kind that we have not provided, the compiler will instantly flag the problem, because it is unable to resolve the generic interface.

**Modules**

*Modules*, already referred to several times above, are Fortran 90’s generalization of Fortran 77’s common blocks, INCLUDED files of parameter statements, and (to some extent) statement functions. Modules are *program units*, like main programs or subprograms (subroutines and functions), that can be separately compiled. A module is a convenient place to stash global data, named constants (what in Fortran 77 are called “symbolic constants” or “PARAMETERS”), interface blocks to subprograms and/or actual subprograms themselves (*module subprograms*). The convenience is that a module’s information can be incorporated into another program unit via a simple, one-line USE statement. [M&R, §5.5]

Here is an example of a simple module that defines a few parameters, creates some global storage for an array named `arr` (as might be done with a Fortran 77 common block), and defines the interface to a function `yourfunc`:
21.3 More on Arrays and Array Sections

Arrays are the central conceptual core of Fortran 90 as a parallel programming language, and thus worthy of some further discussion. We have already seen that arrays can “come into existence” in Fortran 90 in several ways, either directly declared, as

\[
\text{REAL(SP), DIMENSION(100,200) :: arr}
\]

or else allocated by an \emph{allocatable} variable or a \emph{pointer} variable,

\[
\begin{align*}
\text{REAL(SP), DIMENSION(:,:,), ALLOCATABLE :: arr} \\
\text{REAL(SP), DIMENSION(:,:,), POINTER :: barr} \\
\text{allocate(arr(100,200),barr(100,200))}
\end{align*}
\]

or else (not previously mentioned) passed into a subprogram through a dummy argument:

\[
\begin{align*}
\text{SUBROUTINE myroutine(carr)} \\
\text{USE nrtype} \\
\text{REAL(SP), DIMENSION(:,:,) :: carr} \\
\text{...} \\
\text{i=size(carr,1)}
\end{align*}
\]
j = size(carr, 2)

In the above example we also show how the subprogram can find out the size of the actual array that is passed, using the size intrinsic. This routine is an example of the use of an assumed-shape array, new to Fortran 90. The actual extents along each dimension are inherited from the calling routine at run time. A subroutine with assumed-shape array arguments must have an explicit interface in the calling routine, otherwise the compiler doesn’t know about the extra information that must be passed. A typical setup for calling myroutine would be:

```fortran
PROGRAM use_myroutine
  USE nrtype
  REAL(SP), DIMENSION(10,10) :: arr
  INTERFACE
    SUBROUTINE myroutine(carr)
      USE nrtype
      REAL(SP), DIMENSION(:,:) :: carr
    END SUBROUTINE myroutine
  END INTERFACE
  ...
  call myroutine(a)
```

Of course, for the recipes we have provided all the interface blocks in the file nr.f90, and you need only a USE nr statement in your calling program.

### Conformable Arrays

Two arrays are said to be conformable if their shapes are the same. Fortran 90 allows practically all operations among conformable arrays and elemental functions that are allowed for scalar variables. Thus, if `arr`, `barr`, and `carr` are mutually conformable, we can write,

```fortran
arr = barr + cos(carr) + 2.0_sp
```

and have the indicated operations performed, element by corresponding element, on the entire arrays. The above line also illustrates that a scalar (here the constant `2.0_sp`, but a scalar variable would also be fine) is deemed conformable with any array — it gets “expanded” to the shape of the rest of the expression that it is in. [M&R, §3.11]

In Fortran 90, as in Fortran 77, the default lower bound for an array subscript is 1; however, it can be made some other value at the time that the array is declared:

```fortran
REAL(SP), DIMENSION(100,200) :: farr
REAL(SP), DIMENSION(0:99,0:199) :: garr
...
farr = 3.0_sp*garr + 1.0_sp
```

Notice that `farr` and `garr` are conformable, since they have the same shape, in this case `(100,200)`. Also note that when they are used in an array expression, the operations are taken between the corresponding elements of their shapes, not necessarily the corresponding elements of their indices. [M&R, §3.10] In other words, one of the components evaluated is,

```fortran
farr(1,1) = 3.0_sp*garr(0,0) + 1.0_sp
```

This illustrates a fundamental aspect of array (or data) parallelism in Fortran 90. Array constructions should not be thought of as merely abbreviations for do-loops
over indices, but rather as genuinely parallel operations on same-shaped objects, abstracted of their indices. This is why the standard makes no statement about the order in which the individual operations in an array expression are executed; they might in fact be carried out simultaneously, on parallel hardware.

By default, array expressions and assignments are performed for all elements of the same-shaped arrays referenced. This can be modified, however, by use of a where construction like this:

```fortran
where (harr > 0.0_sp)
    farr = 3.0_sp*garr + 1.0_sp
end where
```

Here harr must also be conformable to farr and garr. Analogously with the Fortran if-statement, there is also a one-line form of the where-statement. There is also a where ... elsewhere ... end where form of the statement, analogous to if ... else if ... end if. A significant language limitation in Fortran 90 is that nested where-statements are not allowed. [M&R, §6.8]

**Array Sections**

Much of the versatility of Fortran 90’s array facilities stems from the availability of array sections. An array section acts like an array, but its memory location, and thus the values of its elements, is actually a subset of the memory location of an already-declared array. Array sections are thus “windows into arrays,” and they can appear on either the left side, or the right side, or both, of a replacement statement. Some examples will clarify these ideas.

Let us presume the declarations

```fortran
REAL(SP), DIMENSION(100) :: arr
INTEGER(I4B), DIMENSION(6) :: iarr=(/11,22,33,44,55,66/)
```

Note that iarr is not only declared, it is also initialized by an initialization expression (a replacement for Fortran 77’s DATA statement). [M&R, §7.5] Here are some array sections constructed from these arrays:

<table>
<thead>
<tr>
<th>Array Section</th>
<th>What It Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr(:)</td>
<td>same as arr</td>
</tr>
<tr>
<td>arr(1:100)</td>
<td>same as arr</td>
</tr>
<tr>
<td>arr(1:10)</td>
<td>one-dimensional array containing first 10 elements of arr</td>
</tr>
<tr>
<td>arr(51:100)</td>
<td>one-dimensional array containing second half of arr</td>
</tr>
<tr>
<td>arr(51:)</td>
<td>same as arr(51:100)</td>
</tr>
<tr>
<td>arr(10:1:-1)</td>
<td>one-dimensional array containing first 10 elements of arr, but in reverse order</td>
</tr>
<tr>
<td>arr((/10,99,1,6/))</td>
<td>one-dimensional array containing elements 10, 99, 1, and 6 of arr, in that order</td>
</tr>
<tr>
<td>arr(iarr)</td>
<td>one-dimensional array containing elements 11, 22, 33, 44, 55, 66 of arr, in that order</td>
</tr>
</tbody>
</table>
Now let’s try some array sections of the two-dimensional array

\[
\text{REAL(SP), DIMENSION(100,100) :: barr}
\]

<table>
<thead>
<tr>
<th>Array Section</th>
<th>What It Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{barr(:,::)}</td>
<td>same as \text{barr}</td>
</tr>
<tr>
<td>\text{barr(1:100,1:100)}</td>
<td>same as \text{barr}</td>
</tr>
<tr>
<td>\text{barr(7,:)       }</td>
<td>one-dimensional array containing the 7th row of \text{barr}</td>
</tr>
<tr>
<td>\text{barr(7,1:100)   }</td>
<td>same as \text{barr(7,:)}</td>
</tr>
<tr>
<td>\text{barr(:,7)}</td>
<td>one-dimensional array containing the 7th column of \text{barr}</td>
</tr>
<tr>
<td>\text{barr(21:30,71:90)}</td>
<td>two-dimensional array containing the sub-block of \text{barr} with the indicated ranges of indices; the shape of this array section is ( (10, 20) )</td>
</tr>
<tr>
<td>\text{barr(100:1:-1,100:1:-1)}</td>
<td>two-dimensional array formed by flipping \text{barr} upside down and backwards</td>
</tr>
<tr>
<td>\text{barr(2:100:2,2:100:2)}</td>
<td>two-dimensional array of shape ( (50, 50) ) containing the elements of \text{barr} whose row and column indices are both even</td>
</tr>
</tbody>
</table>

Some terminology: A construction like \( 2:100:2 \), above, is called a \textit{subscript triplet}. Its integer pieces (which may be integer constants, or more general integer expressions) are called \textit{lower}, \textit{upper}, and \textit{stride}. Any of the three may be omitted. An omitted stride defaults to the value 1. Notice that, if \( (\text{upper} - \text{lower}) \) has a different sign from \textit{stride}, then a subscript triplet defines an empty or zero-length array, e.g., \( 1:5:-1 \) or \( 10:1:1 \) (or its equivalent form, simply \( 10:1 \)). Zero-length arrays are not treated as errors in Fortran 90, but rather as “no-ops.” That is, no operation is performed in an expression or replacement statement among zero-length arrays. (This is essentially the same convention as in Fortran 77 for do-loop indices, which array expressions often replace.) \[M&R, \S 6.10\]

It is important to understand that array sections, when used in array expressions, match elements with other parts of the expression \textit{according to shape}, not according to indices. (This is exactly the same principle that we applied, above, to arrays with subscript lower bounds different from the default value of 1.) One frequently exploits this feature in using array sections to carry out operations on arrays that access neighboring elements. For example,

\[
carr(1:n-1,1:n-1) = \text{barr(1:n-1,1:n-1)} + \text{barr(2:n,2:n)}
\]

constructs in the \((n-1) \times (n-1)\) matrix \textit{carr} the sum of each of the corresponding elements in \( n \times n \) \textit{barr} added to its diagonally lower-right neighbor.

Pointers are often used as aliases for array sections, especially if the same array sections are used repeatedly. \[M&R, \S 6.12\] For example, with the setup

\[
\text{REAL(SP), DIMENSION(:,::), POINTER :: leftb, rightb}
\]
leftb=>barr(1:n-1,1:n-1)
rightb=>barr(2:n,2:n)

the statement above can be coded as

carr(1:n-1,1:n-1)=leftb+rightb

We should also mention that array sections, while powerful and concise, are sometimes not quite powerful enough. While any row or column of a matrix is easily accessible as an array section, there is no good way, in Fortran 90, to access (e.g.) the diagonal of a matrix, even though its elements are related by a linear progression in the Fortran storage order (by columns). These so-called skew-sections were much discussed by the Fortran 90 standards committee, but they were not implemented. We will see examples later in this volume of work-around programming tricks (none totally satisfactory) for this omission. (Fortran 95 corrects the omission; see §21.6.)

CITED REFERENCES AND FURTHER READING:

21.4 Fortran 90 Intrinsic Procedures

Much of Fortran 90’s power, both for parallel programming and for its concise expression of algorithmic ideas, comes from its rich set of intrinsic procedures. These have the effect of making the language “large,” hence harder to learn. However, effort spent on learning to use the intrinsics — particularly some of their more obscure, and more powerful, optional arguments — is often handsomely repaid.

This section summarizes the intrinsics that we find useful in numerical work. We omit, here, discussion of intrinsics whose exclusive use is for character and string manipulation. We intend only a summary, not a complete specification, which can be found in M&R’s Chapter 8, or other reference books.

If you find the sheer number of new intrinsic procedures daunting, you might want to start with our list of the “top 10” (with the number of different Numerical Recipes routines that use each shown in parentheses): size (254), sum (44), dot_product (31), merge (27), all (25), maxval (23), matmul (19), pack (18), any (17), and spread (15). (Later, in Chapter 23, you can compare these numbers with our frequency of using the short utility functions that we define in a module named nrutil — several of which we think ought to have been included as Fortran 90 intrinsic procedures.)

The type, kind, and shape of the value returned by intrinsic functions will usually be clear from the short description that we give. As an additional hint (though not necessarily a precise description), we adopt the following codes:
Hint | What It Means
---|---
[Int] | an INTEGER kind type
[Real] | a REAL kind type
[Cmplx] | a COMPLEX kind type
[Num] | a numerical type and kind
[Lgcl] | a LOGICAL kind type
[Iarr] | a one-dimensional INTEGER array
[argTS] | same type and shape as the first argument
[argT] | same type as the first argument, but not necessarily the same shape

### Numerical Elemental Functions

Little needs to be said about the numerical functions with identical counterparts in Fortran 77: abs, acos, aimag, asin, atan, atan2, conjg, cos, cosh, dim, exp, log, log10, max, min, mod, sign, sin, sinh, sqrt, tan, and tanh. In Fortran 90 these are all *elemental* functions, so that any plausible type, kind, and shape of argument may be used. Except for aimag, which returns a real type from a complex argument, these all return [argTS] (see table above).

Although Fortran 90 recognizes, for compatibility, Fortran 77’s so-called *specific names* for these functions (e.g., iabs, dabs, and cabs for the generic abs), these are entirely superfluous and should be avoided.

Fortran 90 corrects some ambiguity (or at least inconvenience) in Fortran 77’s mod(a,p) function, by introducing a new function modulo(a,p). The functions are essentially identical for positive arguments, but for negative a and positive p, modulo gives results more compatible with one’s mathematical expectation that the answer should always be in the positive range 0 to p. E.g., modulo(11,5)=1, and modulo(-11,5)=4. [M&R, §8.3.2]

### Conversion and Truncation Elemental Functions

Fortran 90’s conversion (or, in the language of C, casting) and truncation functions are generally modeled on their Fortran 77 antecedents, but with the addition of an optional second integer argument, kind, that determines the kind of the result. Note that, if kind is omitted, you get a default kind — not necessarily related to the kind of your argument. The kind of the argument is of course known to the compiler by its previous declaration. Functions in this category (see below for explanation of arguments in slanted type) are:

- [Real] `aint(a,kind)`
  Truncate to integer value, return as a real kind.
- [Real] `anint(a,kind)`
  Nearest whole number, return as a real kind.
- [Cmplx] `cmplx(x,y,kind)`
Convert to complex kind. If \( y \) is omitted, it is taken to be 0.

\[ \text{[Int]} \quad \text{int}(a, \text{kind}) \]

Convert to integer kind, truncating towards zero.

\[ \text{[Int]} \quad \text{nint}(a, \text{kind}) \]

Convert to integer kind, choosing the nearest whole number.

\[ \text{[Real]} \quad \text{real}(a, \text{kind}) \]

Convert to real kind.

\[ \text{[Lgcl]} \quad \text{logical}(a, \text{kind}) \]

Convert one logical kind to another.

We must digress here to explain the use of *optional arguments* and *keywords* as Fortran 90 language features. [M&R, §5.13] When a routine (either intrinsic or user-defined) has arguments that are declared to be optional, then the dummy names given to them also become keywords that distinguish — independent of their position in a calling list — which argument is intended to be passed. (There are some additional rules about this that we will not try to summarize here.) In this section’s tabular listings, we indicate optional arguments in intrinsic routines by printing them in smaller slanted type. For example, the intrinsic function

\[
\text{eoshift}(\text{array}, \text{shift}, \text{boundary}, \text{dim})
\]

has two required arguments, \text{array} and \text{shift}, and two optional arguments, \text{boundary} and \text{dim}. Suppose we want to call this routine with the actual arguments \text{myarray}, \text{myshift}, and \text{mydim}, but omitting the argument in the boundary slot. We do this by the expression

\[
\text{eoshift}(\text{myarray}, \text{myshift}, \text{dim} = \text{mydim})
\]

Conversely, if we wanted a boundary argument, but no \text{dim}, we might write

\[
\text{eoshift}(\text{myarray}, \text{myshift}, \text{boundary} = \text{myboundary})
\]

It is always a good idea to use this kind of keyword construction when invoking optional arguments, even though the rules allow keywords to be omitted in some unambiguous cases. Now back to the lists of intrinsic routines.

A peculiarity of the \text{real} function derives from its use both as a type conversion and for extracting the real part of complex numbers (related, but not identical, usages): If the argument of \text{real} is complex, and \text{kind} is omitted, then the result *isn’t* a default real kind, but rather *is* (as one generally would want) the \text{real} kind type corresponding to the kind type of the complex argument, that is, single-precision real for single-precision complex, double-precision for double-precision, and so on. [M&R, §8.3.1] We recommend *never* using \text{kind} when you intend to extract the real part of a complex, and *always* using \text{kind} when you intend conversion of a real or integer value to a particular kind of REAL. (Use of the deprecated function \text{dble} is not recommended.)

The last two conversion functions are the exception in that they *don’t* allow a \text{kind} argument, but rather return default integer kinds. (The X3J3 standards committee has fixed this in Fortran 95.)

\[ \text{[Int]} \quad \text{ceiling}(a) \]

Convert to integer, truncating towards more positive.
Reduction and Inquiry Functions on Arrays

These are mostly the so-called *transformational functions* that accept array arguments and return either scalar values or else arrays of lesser rank. [M&R, §8.11]

With no optional arguments, such functions act on all the elements of their single array argument, regardless of its shape, and produce a scalar result. When the optional argument `dim` is specified, they instead act on all one-dimensional sections that span the dimension `dim`, producing an answer one rank lower than the first argument (that is, omitting the `dim` dimension from its shape). When the optional argument `mask` is specified, only the elements with a corresponding true value in `mask` are scanned.

```
[Int] floor(a)
    Convert to integer, truncating towards more negative.
```

```
Reduction and Inquiry Functions on Arrays

[Lgcl] all(mask, dim)
    Returns true if all elements of mask are true, false otherwise.

[Lgcl] any(mask, dim)
    Returns true if any of the elements of mask are true, false otherwise.

[Int] count(mask, dim)
    Counts the true elements in mask.

[Num] maxval(array, dim, mask)
    Maximum value of the array elements.

[Num] minval(array, dim, mask)
    Minimum value of the array elements.

[Num] product(array, dim, mask)
    Product of the array elements.

[Int] size(array, dim)
    Size (total number of elements) or the extent along dimension dim.

[Num] sum(array, dim, mask)
    Sum of the array elements.
```

The use of the `dim` argument can be confusing, so an example may be helpful. Suppose we have

```
myarray = [[1, 2, 3, 4],
           [5, 6, 7, 8],
           [9, 10, 11, 12]]
```

where, as always, the `i` index in `array(i,j)` numbers the rows while the `j` index numbers the columns. Then

```
sum(myarray, dim=1) = (15, 18, 21, 24)
```

that is, the `i` indices are “summed away” leaving only a `j` index on the result; while

```
sum(myarray, dim=2) = (10, 26, 42)
```
that is, the \( j \) indices are “summed away” leaving only an \( i \) index on the result. Of course we also have

\[
\text{sum(myarray)} = 78
\]

Two related functions return the location of particular elements in an array. The returned value is a one-dimensional integer array containing the respective subscript of the element along each dimension. Note that when the argument object is a one-dimensional array, the returned object is an integer array of length 1, not simply an integer. (Fortran 90 distinguishes between these.)

\[
\text{maxloc(array, mask)}
\]

Location of the maximum value in an array.

\[
\text{minloc(array, mask)}
\]

Location of the minimum value in an array.

Similarly returning a one-dimensional integer array are

\[
\text{shape(array)}
\]

Returns the shape of array as a one-dimensional integer array.

\[
\text{lbound(array, dim)}
\]

When \( \text{dim} \) is absent, returns an array of lower bounds for each dimension of subscripts of array. When \( \text{dim} \) is present, returns the value only for dimension \( \text{dim} \), as a scalar.

\[
\text{ubound(array, dim)}
\]

When \( \text{dim} \) is absent, returns an array of upper bounds for each dimension of subscripts of array. When \( \text{dim} \) is present, returns the value only for dimension \( \text{dim} \), as a scalar.

### Array Unary and Binary Functions

The most powerful array operations are simply built into the language as operators. All the usual arithmetic and logical operators (+, -, *, /, **, .not., .and., .or., .eqv., .neqv.) can be applied to arrays of arbitrary shape or (for the binary operators) between two arrays of the same shape, or between arrays and scalars. The types of the arrays must, of course, be appropriate to the operator used. The result in all cases is to perform the operation element by element on the arrays.

We also have the intrinsic functions,

\[
\text{dot_product(veca, vecb)}
\]

Scalar dot product of two one-dimensional vectors \( \text{veca} \) and \( \text{vecb} \).

\[
\text{matmul(mata, matb)}
\]

Result of matrix-multiplying the two two-dimensional matrices \( \text{mata} \) and \( \text{matb} \). The shapes have to be such as to allow matrix multiplication. Vectors (one-dimensional arrays) are additionally allowed as either the first or second argument, but not both; they are treated as row vectors in the first argument, and as column vectors in the second.

You might wonder how to form the \textit{outer} product of two vectors, since \text{matmul} specifically excludes this case. (See §22.1 and §23.5 for answer.)
Array Manipulation Functions

These include many powerful features that a good Fortran 90 programmer should master.

[argTS] \texttt{cshift(array, shift, dim)}

If \texttt{dim} is omitted, it is taken to be 1. Returns the result of circularly left-shifting every one-dimensional section of \texttt{array} (in dimension \texttt{dim}) by \texttt{shift} (which may be negative). That is, for positive \texttt{shift}, values are moved to smaller subscript positions. Consult a Fortran 90 reference (e.g., [M&R, §8.13.5]) for the case where \texttt{shift} is an array.

[argTS] \texttt{merge(tsource, fsource, mask)}

Returns same shape object as \texttt{tsource} and \texttt{fsource} containing the former’s components where \texttt{mask} is true, the latter’s where it is false.

[argTS] \texttt{eoshift(array, shift, boundary, dim)}

If \texttt{dim} is omitted, it is taken to be 1. Returns the result of end-off left-shifting every one-dimensional section of \texttt{array} (in dimension \texttt{dim}) by \texttt{shift} (which may be negative). That is, for positive \texttt{shift}, values are moved to smaller subscript positions. If \texttt{boundary} is present as a scalar, it supplies elements to fill in the blanks; if it is not present, zero values are used. Consult a Fortran 90 reference (e.g., [M&R, §8.13.5]) for the case where \texttt{boundary} and/or \texttt{shift} is an array.

[argT] \texttt{pack(array, mask, vector)}

Returns a one-dimensional array containing the elements of \texttt{array} that pass the \texttt{mask}. Components of optional \texttt{vector} are used to pad out the result to the size of \texttt{vector} with specified values.

[argT] \texttt{reshape(source, shape, pad, order)}

Takes the elements of \texttt{source}, in normal Fortran order, and returns them (as many as will fit) as an array whose shape is specified by the one-dimensional integer array \texttt{shape}. If there is space remaining, then \texttt{pad} must be specified, and is used (as many sequential copies as necessary) to fill out the rest. For description of \texttt{order}, consult a Fortran 90 reference, e.g., [M&R, 8.13.3].

[argT] \texttt{spread(source, dim, ncopies)}

Returns an array whose rank is one greater than \texttt{source}, and whose \texttt{dim} dimension is of length \texttt{ncopies}. Each of the result’s \texttt{ncopies} array sections having a fixed subscript in dimension \texttt{dim} is a copy of \texttt{source}. (That is, it spreads \texttt{source} into the \texttt{dimth} dimension.)

[argT] \texttt{transpose(matrix)}

Returns the transpose of \texttt{matrix}, which must be two-dimensional.

[argT] \texttt{unpack(vector, mask, field)}

Returns an array whose type is that of \texttt{vector}, but whose shape is that of \texttt{mask}. The components of \texttt{vector} are put, in order, into the positions where \texttt{mask} is true. Where \texttt{mask} is false, components of \texttt{field} (which may be a scalar or an array with the same shape as \texttt{mask}) are used instead.
Bitwise Functions

Most of the bitwise functions should be familiar to Fortran 77 programmers as longstanding standard extensions of that language. Note that the bit positions number from zero to one less than the value returned by the `bit_size` function. Also note that bit positions number from right to left. Except for `bit_size`, the following functions are all elemental.

[Int] `bit_size(i)`
Number of bits in the integer type of `i`.

[Lgcl] `btest(i,pos)`
True if bit position `pos` is 1, false otherwise.

[Int] `iand(i,j)`
Bitwise logical and.

[Int] `ibclr(i,pos)`
Returns `i` but with bit position `pos` set to zero.

[Int] `ibits(i,pos,len)`
Extracts `len` consecutive bits starting at position `pos` and puts them in the low bit positions of the returned value. (The high positions are zero.)

[Int] `ibset(i,pos)`
Returns `i` but with bit position `pos` set to 1.

[Int] `ieor(i,j)`
Bitwise exclusive or.

[Int] `ior(i,j)`
Bitwise logical or.

[Int] `ishft(i,shift)`
Bitwise left shift by `shift` (which may be negative) with zeros shifted in from the other end.

[Int] `ishftc(i,shift)`
Bitwise circularly left shift by `shift` (which may be negative).

[Int] `not(i)`
Bitwise logical complement.

Some Functions Relating to Numerical Representations

[Real] `epsilon(x)`
Smallest nonnegligible quantity relative to 1 in the numerical model of `x`.

[Num] `huge(x)`
Largest representable number in the numerical model of `x`.

[Int] `kind(x)`
Returns the kind value for the numerical model of \( x \).

\[
\text{[Real] nearest}(x,s)
\]

Real number nearest to \( x \) in the direction specified by the sign of \( s \).

\[
\text{[Real] tiny}(x)
\]

Smallest positive number in the numerical model of \( x \).

**Other Intrinsic Procedures**

\[
\text{[Lgcl] present}(a)
\]

True, within a subprogram, if an optional argument is actually present, otherwise false.

\[
\text{[Lgcl] associated}(\text{pointer}, \text{target})
\]

True if \( \text{pointer} \) is associated with \( \text{target} \) or (if \( \text{target} \) is absent) with any target, otherwise false.

\[
\text{[Lgcl] allocated}(\text{array})
\]

True if the allocatable \( \text{array} \) is allocated, otherwise false.

There are some pitfalls in using \( \text{associated} \) and \( \text{allocated} \), having to do with arrays and pointers that can find themselves in *undefined* status [see §21.5, and also M&R, §3.3 and §6.5.1]. For example, pointers are always “born” in an undefined status, where the \( \text{associated} \) function returns unpredictable values.

For completeness, here is a list of Fortran 90’s intrinsic procedures not already mentioned:

**Other Numerical Representation Functions:** digits, exponent, fraction, rrspacing, scale, set_exponent, spacing, max_exponent, min_exponent, precision, radix, range, selected_int_kind, selected_real_kind

**Lexical comparison:** lge, lgt, lle, llt.

**Character functions:** ichar, char, achar, iachar, index, adjustl, adjustr, len_trim, repeat, scan, trim, verify.

**Other:** mvbits, transfer, date_and_time, system_clock, random_seed, random_number. (We will discuss random numbers in some detail in Chapter B7.)

CITED REFERENCES AND FURTHER READING:

21.5 Advanced Fortran 90 Topics

Pointers, Arrays, and Memory Management

One of the biggest improvements in Fortran 90 over Fortran 77 is in the handling of arrays, which are the cornerstone of many numerical algorithms. In this subsection we will take a closer look at how to use some of these new array features effectively. We will look at how to code certain commonly occurring elements of program design, and we will pay particular attention to avoiding “memory leaks,” where — usually inadvertently — we keep cumulatively allocating new storage for an array, every time some piece of code is invoked.

Let’s first review some of the rules for using allocatable arrays and pointers to arrays. Recall that a pointer is born with an undefined status. Its status changes to “associated” when you make it refer to a target, and to “disassociated” when you nullify the pointer. [M&R, §3.3] You can also use nullify on a newly born pointer to change its status from undefined to disassociated; this allows you to test the status with the associated inquiry function. [M&R, §6.5.4] (While many compilers will not produce a run-time error if you test an undefined pointer with associated, you can’t rely on this laissez-faire in your programming.)

The initial status of an allocatable array is “not currently allocated.” Its status changes to “allocated” when you give it storage with allocate, and back to “not currently allocated” when you use deallocate. [M&R, §6.5.1] You can test the status with the allocated inquiry function. Note that while you can also give a pointer fresh storage with allocate, you can’t test this with allocated — only associated is allowed with pointers. Note also that nullifying an allocated pointer leaves its associated storage in limbo. You must instead deallocate, which gives the pointer a testable “disassociated” status.

While allocating an array that is already allocated gives an error, you are allowed to allocate a pointer that already has a target. This breaks the old association, and could leave the old target inaccessible if there is no other pointer associated with it. [M&R, §6.5.2] Deallocating an array or pointer that has not been allocated is always an error.

Allocated arrays that are local to a subprogram acquire the “undefined” status on exit from the subprogram unless they have the SAVE attribute. (Again, not all compilers enforce this, but be warned!) Such undefined arrays cannot be referenced in any way, so you should explicitly deallocate all allocated arrays that are not saved before returning from a subprogram. [M&R, §6.5.1] The same rule applies to arrays declared in modules that are currently accessed only by the subprogram. While you can reference undefined pointers (e.g., by first nullifying them), it is good programming practice to deallocate explicitly any allocated pointers declared locally before leaving a subprogram or module.

Now let’s turn to using these features in programs. The simplest example is when we want to implement global storage of an array that needs to be accessed by two or more different routines, and we want the size of the array to be determined at run time. As mentioned earlier, we implement global storage with a MODULE rather than a COMMON block. (We ignore here the additional possibility of passing
global variables by having one routine CONTAINed within the other.) There are two good ways of handling the dynamical allocation in a MODULE. Method 1 uses an allocatable array:

```
MODULE a
REAL(SP), DIMENSION(:), ALLOCATABLE :: x
END MODULE a
```

```
SUBROUTINE b(y)
USE a
REAL(SP), DIMENSION(:) :: y
... allocate(x(size(y)))
... [other routines using x called here] ...
END SUBROUTINE b
```

Here the global variable \( x \) gets assigned storage in subroutine \( b \) (in this case, the same as the length of \( y \)). The length of \( y \) is of course defined in the procedure that calls \( b \). The array \( x \) is made available to any other subroutine called by \( b \) by including a `USE a` statement. The status of \( x \) can be checked with an `allocated` inquiry function on entry into either \( b \) or the other subroutine if necessary. As discussed above, you must be sure to deallocate \( x \) before returning from subroutine \( b \). If you want \( x \) to retain its values between calls to \( b \), you add the `SAVE` attribute to its declaration in \( a \), and don’t deallocate it on returning from \( b \). (Alternatively, you could put a `USE a` in your main program, but we consider that bug-prone, since forgetting to do so can create all manner of difficult-to-diagnose havoc.) To avoid allocating \( x \) more than once, you test it on entry into \( b \):

```
if (.not. allocated(x)) allocate(x(size(y)))
```

The second way to implement this type of global storage (Method 2) uses a pointer:

```
MODULE a
REAL(SP), DIMENSION(:), POINTER :: x
END MODULE a
```

```
SUBROUTINE b(y)
USE a
REAL(SP), DIMENSION(:) :: y
REAL(SP), DIMENSION(size(y)), TARGET :: xx
... x=>xx
... [other routines using x called here] ...
END SUBROUTINE b
```

Here the automatic array \( xx \) gets its temporary storage automatically on entry into \( b \), and automatically gets deallocated on exit from \( b \). [M&R, §6.4] The global pointer \( x \) can access this storage in any routine with a `USE a` that is called by \( b \). You can check that things are in order in such a called routine by testing \( x \) with `associated`. If you are going to use \( x \) for some other purpose as well, you should `nullify` it on leaving \( b \) so that it doesn’t have undefined status. Note that this implementation does not allow values to be saved between calls: You can’t `SAVE` automatic arrays — that’s not what they’re for. You would have to `SAVE` \( x \) in the module, and `allocate` it in the subroutine instead of pointing it to a suitable automatic array. But this is essentially Method 1 with the added complication of using a pointer, so Method 1 is simpler when you want to save values. When you don’t
need to save values between calls, we lean towards Method 2 over Method 1 because we like the automatic allocation and deallocation, but either method works fine.

An example of Method 1 (allocatable array) is in routine rkdumb on page 1297. An example of Method 1 with SAVE is in routine pwtset on p. 1265. Method 2 (pointer) shows up in routines newt (p. 1196), broydn (p. 1199), and fitexy (p. 1286). A variation is shown in routines linmin (p. 1211) and dlinmin (p. 1212): When the array that needs to be shared is an argument of one of the routines, Method 2 is better.

An extension of these ideas occurs if we allocate some storage for an array initially, but then might need to increase the size of the array later without losing the already-stored values. The function reallocate in our utility module nrutil will handle this for you, but it expects a pointer argument as in Method 2. Since no automatic arrays are used, you are free to SAVE the pointer if necessary. Here is a simple example of how to use reallocate to create a workspace array that is local to a subroutine:

```fortran
SUBROUTINE a
    USE nrutil, ONLY : reallocate
    REAL(SP), DIMENSION(:), POINTER, SAVE :: wksp
    LOGICAL(LGT), SAVE :: init=.true.
    if (init) then
        init=.false.
        nullify(wksp)
        wksp=>reallocate(wksp,100)
    end if
    ...
    if (nterm > size(wksp)) wksp=>reallocate(wksp,2*size(wksp))
    ...
END SUBROUTINE a
```

Here the workspace is initially allocated a size of 100. If the number of elements used (nterm) ever exceeds the size of the workspace, the workspace is doubled. (In a realistic example, one would of course check that the doubled size is in fact big enough.) Fortran 90 experts can note that the SAVE on init is not strictly necessary: Any local variable that is initialized is automatically saved. [M&R, §7.5]

You can find similar examples of reallocatem in eulsum (p. 1070), hufenc (p. 1348), and arcode (p. 1350). Examples of reallocate used with global variables in modules are in odeint (p. 1300) and ran_state (p. 1144).

Another situation where we have to use pointers and not allocatable arrays is when the storage is required for components of a derived type, which are not allowed to have the allocatable attribute. Examples are in hufmak (p. 1346) and arcmak (p. 1349).

Turning away from issues relating to global variables, we now consider several other important programming situations that are nicely handled with pointers. The first case is when we want a subroutine to return an array whose size is not known in advance. Since dummy arguments are not allocatable, we must use a pointer. Here is the basic construction:

```fortran
SUBROUTINE a(x,nx)
    REAL(SP), DIMENSION(:,), POINTER :: x
    INTEGER(I4B), INTENT(OUT) :: nx
    LOGICAL(LGT), SAVE :: init=.true.
    if (init) then
```

init=.false.
nullify(x)
else
   if (associated(x)) deallocate(x)
end if
   nx...
allocate(x(nx))
x(1:nx)=...
END SUBROUTINE a

Since the length of x can be found from size(x), it is not absolutely necessary to pass nx as an argument. Note the use of the initial logic to avoid memory leaks. If a higher-level subroutine wants to recover the memory associated with x from the last call to SUBROUTINE a, it can do so by first deallocating it, and then nullifying the pointer. Examples of this structure are in zbrak (p. 1184), period (p. 1258), and fasper (p. 1259). A related situation is where we want a function to return an array whose size is not predetermined, such as in voltra (p. 1326). The discussion of voltra also explains the potential pitfalls of functions returning pointers to dynamically allocated arrays.

A final useful pointer construction enables us to set up a data structure that is essentially an array of arrays, independently allocatable on each part. We are not allowed to declare an array of pointers in Fortran 90, but we can do this indirectly by defining a derived type that consists of a pointer to the appropriate kind of array. [M&R, §6.11] We can then define a variable that is an allocatable array of the new type. For example,

```fortran
TYPE ptr_to_arr
   REAL(SP), DIMENSION(:), POINTER :: arr
END TYPE

TYPE(ptr_to_arr), DIMENSION(:), ALLOCATABLE :: x
...
allocate(x(n))
...
do i=1,n
   allocate(x(i)%arr(m))
end do
```

sets up a set x of n arrays of length m. See also the example in mglin (p. 1334).

There is a potential problem with dynamical memory allocation that we should mention. The Fortran 90 standard does not require that the compiler perform “garbage collection,” that is, it is not required to recover deallocated memory into nice contiguous pieces for reuse. If you enter and exit a subroutine many times, and each time a large chunk of memory gets allocated and deallocated, you could run out of memory with a “dumb” compiler. You can often alleviate the problem by deallocating variables in the reverse order that you allocated them. This tends to keep a large contiguous piece of memory free at the top of the heap.

**Scope, Visibility, and Data Hiding**

An important principle of good programming practice is *modularization*, the idea that different parts of a program should be insulated from each other as much as possible. An important subcase of modularization is *data hiding*, the principle that actions carried out on variables in one part of the code should not be able to
affect the values of variables in other parts of the code. When it is necessary for one "island" of code to communicate with another, the communication should be through a well-defined interface that makes it obvious exactly what communication is taking place, and prevents any other interchange from occurring. Otherwise, different sections of code should not have access to variables that they don’t need.

The concept of data hiding extends not only to variables, but also to the names of procedures that manipulate the variables: A program for screen graphics might give the user access to a routine for drawing a circle, but it might “hide” the names (and methods of operation) of the primitive routines used for calculating the coordinates of the points on the circumference. Besides producing code that is easier to understand and to modify, data hiding prevents unintended side effects from producing hard-to-find errors.

In Fortran, the principal language construction that effects data hiding is the use of subroutines. If all subprograms were restricted to have no more than ten executable statements per routine, and to communicate between routines only by an explicit list of arguments, the number of programming errors might be greatly reduced! Unfortunately few tasks can be easily coded in this style. For this and other reasons, we think that too much procedurization is a bad thing; one wants to find the right amount. Fortunately Fortran 90 provides several additional tools to help with data hiding.

Global variables and routine names are important, but potentially dangerous, things. In Fortran 90, global variables are typically encapsulated in modules. Access is granted only to routines with an appropriate USE statement, and can be restricted to specific identifiers by the ONLY option. [M&R, §7.10] In addition, variable and routine names within the module can be designated as PUBLIC or PRIVATE (see, e.g., quad3d on p. 1065). [M&R, §7.6]

The other way global variables get communicated is by having one routine CONTAINED within another. [M&R, §5.6] This usage is potentially lethal, however, because all the outer routine’s variables are visible to the inner routine. You can try to control the problem somewhat by passing some variables back and forth as arguments of the inner routine, but that still doesn’t prevent inadvertent side effects. (The most common, and most stupid, is inadvertent reuse of variables named i or j in the CONTAINED routine.) Also, a long list of arguments reduces the convenience of using an internal routine in the first place. We advise that internal subprograms be used with caution, and only to carry out simple tasks.

There are some good ways to use CONTAINS, however. Several of our recipes have the following structure: A principal routine is invoked with several arguments. It calls a subsidiary routine, which needs to know some of the principal routine’s arguments, some global variables, and some values communicated directly as arguments to the subsidiary routine. In Fortran 77, we have usually coded this by passing the global variables in a COMMON block and all other variables as arguments to the subsidiary routine. If necessary, we copied the arguments of the primary routine before passing them to the subsidiary routine. In Fortran 90, there is a more elegant way of accomplishing this, as follows:

```
SUBROUTINE recipe(arg)
REAL(SP) :: arg
REAL(SP) :: global_var
call recipe_private
END SUBROUTINE
```
SUBROUTINE recipe_private
...
call subsidiary(local_arg)
...
END SUBROUTINE recipe_private
SUBROUTINE subsidiary(local_arg)
...
END SUBROUTINE subsidiary
END SUBROUTINE recipe

Notice that the principal routine (recipe) has practically nothing in it — only declarations of variables intended to be visible to the subsidiary routine (subsidiary). All the real work of recipe is done in recipe_private. This latter routine has visibility on all of recipe’s variables, while any additional variables that recipe_private defines are not visible to subsidiary — which is the whole purpose of this way of organizing things. Obviously arg and global_var can be much more general data types than the example shown here, including function names. For examples of this construction, see amoeba (p. 1208), amebsa (p. 1222), mrqmin (p. 1292), and medfit (p. 1294).

Recursion

A subprogram is recursive if it calls itself. While forbidden in Fortran 77, recursion is allowed in Fortran 90. [M&R, §5.16–§5.17] You must supply the keyword RECURSIVE in front of the FUNCTION or SUBROUTINE keyword. In addition, if a FUNCTION calls itself directly, as opposed to calling another subprogram that in turn calls it, you must supply a variable to hold the result with the RESULT keyword. Typical syntax for this case is:

```fortran
RECURSIVE FUNCTION f(x) RESULT(g)
REAL(SP) :: x,g
if ...
g=...
else
  g=f(...)
end if
END FUNCTION f
```

When a function calls itself directly, as in this example, there always has to be a “base case” that does not call the function; otherwise the recursion never terminates. We have indicated this schematically with the if...else...end if structure.

On serial machines we tend to avoid recursive implementations because of the additional overhead they incur at execution time. Occasionally there are algorithms for which the recursion overhead is relatively small, and the recursive implementation is simpler than an iterative version. Examples in this book are quad_3d (p. 1065), miser (p. 1164), and mglin (p. 1334). Recursion is much more important when parallelization is the goal. We will encounter in Chapter 22 numerous examples of algorithms that can be parallelized with recursion.

SAVE Usage Style

A quirk of Fortran 90 is that any variable with initial values acquires the SAVE attribute automatically. [M&R, §7.5 and §7.9] As a help to understanding
an algorithm, we have elected to put an explicit SAVE on all variables that really do need to retain their values between calls to a routine. We do this even if it is redundant because the variables are initialized. Note that we generally prefer to assign initial values with initialization expressions rather than with DATA statements. We reserve DATA statements for cases where it is convenient to use the repeat count feature to set multiple occurrences of a value, or when binary, octal, or hexadecimal constants are used. [M&R, §2.6.1]

**Named Control Structures**

Fortran 90 allows control structures such as do loops and if blocks to be named. [M&R, §4.3–4.5] Typical syntax is

```fortran
name:do i=1,n
... end do name
```

One use of naming control structures is to improve readability of the code, especially when there are many levels of nested loops and if blocks. A more important use is to allow exit and cycle statements, which normally refer to the innermost do loop in which they are contained, to transfer execution to the end of some outer loop. This is effected by adding the name of the outer loop to the statement: exit name or cycle name.

There is great potential for misuse with named control structures, since they share some features of the much-maligned goto. We recommend that you use them sparingly. For a good example of their use, contrast the Fortran 77 version of simplex with the Fortran 90 version on p. 1216.

CITED REFERENCES AND FURTHER READING:

**21.6 And Coming Soon: Fortran 95**

One of the more positive effects of Fortran 90’s long gestation period has been the general recognition, both by the X3J3 committee and by the community at large, that Fortran needs to evolve over time. Indeed, as we write, the process of bringing forth a minor, but by no means insignificant, updating of Fortran 90 — named Fortran 95 — is well under way.

Fortran 95 will differ from Fortran 90 in about a dozen features, only a handful of which are of any importance to this book. Generally these are extensions that will make programming, especially parallel programming, easier. In this section we give a summary of the anticipated language changes. In §22.1 and §22.5 we will comment further on the implications of Fortran 95 to some parallel programming tasks; in §23.7 we comment on what differences Fortran 95 will make to our nrutil utility functions.

No programs in Chapters B1 through B20 of this book edition use any Fortran 95 extensions.
**FORALL Statements and Blocks**

Fortran 95 introduces a new forall control structure, somewhat akin to the where construct, but allowing for greater flexibility. It is something like a do-loop, but with the proviso that the indices looped over are allowed to be done in any order (ideally, in parallel). The forall construction comes in both single-statement and block variants. Instead of using the do-loop’s comma-separated triplets of lower-value, upper-value, and increment, it borrows its syntax from the colon-separated form of array sections. Some examples will give you the idea.

Here is a simple example that could alternatively be done with Fortran 90’s array sections and transpose intrinsic:

```fortran
forall (i=1:20, j=1:10:2) x(i,j)=y(j,i)
```

The block form allows more than one executable statement:

```fortran
forall (i=1:20, j=1:10:2)
  x(i,j)=y(j,i)
  z(i,j)=y(i,j)**2
end forall
```

Here is an example that cannot be done with Fortran 90 array sections:

```fortran
forall (i=1:20, j=1:20) a(i,j)=3*i+j**2
```

forall statements can also take optional masks that restrict their action to a subset of the loop index combinations:

```fortran
forall (i=1:100, j=1:100, (i>=j .and. x(i,j)/=0.0) ) x(i,j)=1.0/x(i,j)
```

forall constructions can be nested, or nested inside where blocks, or have where constructions inside them. An additional new feature in Fortran 95 is that where blocks can themselves be nested.

**PURE Procedures**

Because the inside iteration of a forall block can be done in any order, or in parallel, there is a logical difficulty in allowing functions or subroutines inside such blocks: If the function or subroutine has side effects (that is, if it changes any data elsewhere in the machine, or in its own saved variables) then the result of a forall calculation could depend on the order in which the iterations happen to be done. This can’t be tolerated, of course; hence a new PURE attribute for subprograms.

While the exact stipulations are somewhat technical, the basic idea is that if you declare a function or subroutine as PURE, with a syntax like,

```fortran
PURE FUNCTION myfunc(x,y,z)
```

or

```fortran
PURE SUBROUTINE mysub(x,y,z)
```

then you are guaranteeing to the compiler (and it will enforce) that the only values changed by mysub or myfunc are returned function values, subroutine arguments with the INTENT(OUT) attribute, and automatic (scratch) variables within the procedure.

You can then use your pure procedures within forall constructions. Pure functions are also allowed in some specification statements.
21.6 And Coming Soon: Fortran 95

**ELEMENTAL Procedures**

Fortran 95 removes Fortran 90’s nagging restriction that only intrinsic functions are elemental. The way this works is that you write a pure procedure that operates on scalar values, but include the attribute ELEMENTAL (which automatically implies PURE). Then, as long as the function has an explicit interface in the referencing program, you can call it with any shape of argument, and it will act elementally. Here’s an example:

```fortran
ELEMENTAL FUNCTION myfunc(x,y,z)
REAL :: x,y,z,myfunc
...
myfunc = ...
END
```

In a program with an explicit interface for `myfunc` you could now have

```fortran
REAL, DIMENSION(10,20) :: x,y,z,w
...
w=myfunc(x,y,z)
```

**Pointer and Allocatable Improvements**

Fortran 95, unlike Fortran 90, requires that any allocatable variables (except those with SAVE attributes) that are allocated within a subprogram be automatically deallocated by the compiler when the subprogram is exited. This will remove Fortran 90’s “undefined allocation status” bugaboo.

Fortran 95 also provides a method for pointer variables to be born with disassociated association status, instead of the default (and often inconvenient) “undefined” status. The syntax is to add an initializing `=> NULL()` to the declaration, as:

```fortran
REAL, DIMENSION(:,,:), POINTER :: mypoint => NULL()
```

This does not, however, eliminate the possibility of undefined association status, because you have to remember to use the null initializer if want your pointer to be disassociated.

**Some Other Fortran 95 Features**

In Fortran 95, `maxloc` and `minloc` have the additional optional argument DIM, which causes them to act on all one-dimensional sections that span through the named dimension. This provides a means for getting the locations of the values returned by the corresponding functions `maxval` and `minval` in the case that their DIM argument is present.

The `sign` intrinsic can now distinguish a negative from a positive real zero value: `sign(2.0,-0.0)` is `-2.0`.

There is a new intrinsic subroutine `cpu_time(time)` that returns as a real value `time` a process’s elapsed CPU time.

There are some minor changes in the namelist facility, in defining minimum field widths for the I, B, O, Z, and F edit descriptors, and in resolving minor conflicts with some other standards.
Chapter 22. Introduction to Parallel Programming

22.0 Why Think Parallel?

In recent years we Numerical Recipes authors have increasingly become convinced that a certain revolution, cryptically denoted by the words “parallel programming,” is about to burst forth from its gestation and adolescence in the community of supercomputer users, and become the mainstream methodology for all computing.

Let’s review the past: Take a screwdriver and open up the computer (workstation or PC) that sits on your desk. (Don’t blame us if this voids your warranty; and be sure to unplug it first!) Count the integrated circuits — just the bigger ones, with more than a million gates (transistors). As we write, in 1995, even lowly memory chips have one or four million gates, and this number will increase rapidly in coming years. You’ll probably count at least dozens, and often hundreds, of such chips in your computer.

Next ask, how many of these chips are CPUs? That is, how many implement von Neumann processors capable of executing arbitrary, stored program code? For most computers, in 1995, the answer is: about one. A significant number of computers do have secondary processors that offload input-output and/or video functions. So, two or three is often a more accurate answer, but only one is usually under the user’s direct control.

Why do our desktop computers have dozens or hundreds of memory chips, but most often only one (user-accessible) CPU? Do CPU chips intrinsically cost more to manufacture? No. Are CPU chips more expensive than memory chips? Yes, primarily because fixed development and design costs must be distributed over a smaller number of units sold. We have been in a kind of economic equilibrium: CPU’s are relatively expensive because there is only one per computer; and there is only one per computer, because they are relatively expensive.

Stabilizing this equilibrium has been the fact that there has been no standard, or widely taught, methodology for parallel programming. Except for the special case of scientific computing on supercomputers (where large problems often have a regular or geometric character), it is not too much of an exaggeration to say that nobody really knows how to program multiprocessor machines. Symmetric multiprocessor
Why Think Parallel?

Operating systems, for example, have been very slow in developing; and efficient, parallel methodologies for query-serving on large databases are even now a subject of continuing research.

However, things are now changing. We consider it an easy prognostication that, by the first years of the new century, the typical desktop computer will have 4 to 8 user-accessible CPUs; ten years after that, the typical number will be between 16 and 512. It is not coincidence that these numbers are characteristic of supercomputers (including some quite different architectures) in 1995. The rough rule of ten years’ lag from supercomputer to desktop has held firm for quite some time now.

Scientists and engineers have the advantage that techniques for parallel computation in their disciplines have already been developed. With multiprocessor workstations right around the corner, we think that now is the right time for scientists and engineers who use computers to start thinking parallel. We don’t mean that you should put an axe through the screen of your fast serial (single-CPU) workstation. We do mean, however, that you should start programming somewhat differently on that workstation, indeed, start thinking a bit differently about the way that you approach numerical problems in general.

In this volume of Numerical Recipes in Fortran, our pedagogical goal is to show you that there are conceptual and practical benefits in parallel thinking, even if you are using a serial machine today. These benefits include conciseness and clarity of code, reusability of code in wider contexts, and (not insignificantly) increased portability of code to today’s parallel supercomputers. Of course, on parallel machines, either supercomputers today or desktop machines tomorrow, the benefits of thinking parallel are much more tangible: They translate into significant improvements in efficiency and computational capability.

Thinking Parallel with Fortran 90

Until very recently, a strong inhibition to thinking parallel was the lack of any standard, architecture-independent, computer language in which to think. That has changed with the finalization of the Fortran 90 language standard, and with the availability of good, optimizing Fortran 90 compilers on a variety of platforms.

There is a significant body of opinion (with which we, however, disagree) that there is no such thing as architecture-independent parallel programming. Proponents of this view, who are generally committed wizards at programming on one or another particular architecture, point to the fact that algorithms that are optimized to one architecture can run hundreds of times more slowly on other architectures. And, they are correct!

Our opposing point of view is one of pragmatism. We think that it is not hard to learn, in a general way, what kinds of architectures are in general use, and what kinds of parallel constructions work well (or poorly) on each kind. With this knowledge (much of which we hope to develop in this book) the user can, we think, write good, general-purpose parallel code that works on a variety of architectures — including, importantly, on purely serial machines. Equally important, the user will be aware of when certain parts of a code can be significantly improved on some, but not other, architectures.

Fortran 90 is a good test-bench for this point of view. It is not the perfect language for parallel programming. But it is a language, and it is the only...
cross-platform standard language now available. The committee that developed the language between 1978 and 1991 (known technically as X3J3) had strong representation from both a traditional “vectorization” viewpoint (e.g., from the Cray XMP and YMP series of computers), and also from the “data parallel” or “SIMD” viewpoints of parallel machines like the CM-2 and CM-5 from Thinking Machines, Inc. Language compromises were made, and a few (in our view) almost essential features were left out (see §22.5). But, by and large, the necessary tools are there: If you learn to think parallel in Fortran 90, you will easily be able to transfer the skill to future parallel standards, whether they are Fortran-based, C-based, or other.

CITED REFERENCES AND FURTHER READING:

22.1 Fortran 90 Data Parallelism: Arrays and Intrinsics

The underlying model for parallel computation in Fortran 90 is data parallelism, implemented by the use of arrays of data, and by the provision of operations and intrinsic functions that act on those arrays in parallel, in a manner optimized by the compiler for each particular hardware architecture. We will not try to draw a fine definitional distinction between “data parallelism” and so-called SIMD (single instruction multiple data) programming. For our purposes the two terms mean about the same thing: The programmer writes a single operation, “+” say, and the compiler causes it to be carried out on multiple pieces of data in as parallel a manner as the underlying hardware allows.

Any kind of parallel computing that is not SIMD is generally called MIMD (multiple instruction multiple data). A parallel programming language with MIMD features might allow, for example, several different subroutines — acting on different parts of the data — to be called into execution simultaneously. Fortran 90 has few, if any, MIMD constructions. A Fortran 90 compiler might, on some machines, execute MIMD code in implementing some Fortran 90 intrinsic functions (pack or unpack, e.g.), but this will be hidden from the Fortran 90 user. Some extensions of Fortran 90, like HPF, do implement MIMD features explicitly; but we will not consider these in this book. Fortran 95’s forall and PURE extensions (see §21.6) will allow some significantly greater access to MIMD features (see §22.5).

Array Parallel Operations

We have already met the most basic, and most important, parallel facility of Fortran 90, namely, the ability to use whole arrays in expressions and assignments, with the indicated operations being effected in parallel across the array. Suppose, for example, we have the two-dimensional matrices a, b, and c,

\[
\text{REAL, DIMENSION(30,30) :: } a, b, c
\]
Then, instead of the serial construction,

\[
\begin{align*}
&\text{do } j=1,30 \\
&\hspace{1em} \text{do } k=1,30 \\
&\hspace{2em} c(j,k)=a(j,k)+b(j,k) \\
&\hspace{1em} \text{end do} \\
&\text{end do}
\end{align*}
\]

which is of course perfectly valid Fortran 90 code, we can simply write

\[c=a+b\]

The compiler deduces from the declaration statement that \(a\), \(b\), and \(c\) are matrices, and what their bounding dimensions are.

Let us dwell for a moment on the conceptual differences between the serial code and parallel code for the above matrix addition. Although one is perhaps used to seeing the nested do-loops as simply an idiom for “do-the-enclosed-on-all-components,” it in fact, according to the rules of Fortran, specifies a very particular time-ordering for the desired operations. The matrix elements are added by rows, in order \((j=1,30)\), and within each row, by columns, in order \((k=1,30)\).

In fact, the serial code above overspecifies the desired task, since it is guaranteed by the laws of mathematics that the order in which the element operations are done is of no possible relevance. Over the 50 year lifetime of serial von Neuman computers, we programmers have been brainwashed to break up all problems into single executable streams in the time dimension only. Indeed, the major design problem for supercomputer compilers for the last 20 years has been to undo such serial constructions and recover the underlying “parallel thoughts,” for execution in vector or parallel processors. Now, rather than taking this expensive detour into and out of serial-land, we are asked simply to say what we mean in the first place, \(c=a+b\).

The essence of parallel programming is not to force “into the time dimension” (i.e., to serialize) operations that naturally extend across a span of data, that is, “in the space dimension.” If it were not for 50-year-old collective habits, and the languages designed to support them, parallel programming would probably strike us as more natural than its serial counterpart.

**Broadcasts and Dimensional Expansion: SSP vs. MMP**

We have previously mentioned the Fortran 90 rule that a scalar variable is conformable with any shape array. Thus, we can implement a calculation such as

\[
y_i = x_i + s, \quad i = 1, \ldots, n
\]

with code like

\[y=x+s\]

where we of course assume previous declarations like

\[
\begin{align*}
\text{REAL(SP)} & :: s \\
\text{REAL(SP), DIMENSION(n)} & :: x,y
\end{align*}
\]

with \(n\) a compile-time constant or dummy argument. (Hereafter, we will omit the declarations in examples that are this simple.)

This seemingly simple construction actually hides an important underlying parallel capability, namely, that of broadcast. The sums in \(y=x+s\) are done in parallel...
on different CPUs, each CPU accessing different components of $x$ and $y$. Yet, they all must access the same scalar value $s$. If the hardware has local memory for each CPU, the value of $s$ must be replicated and transferred to each CPU’s local memory. On the other hand, if the hardware implements a single, global memory space, it is vital to do something that mitigates the traffic jam potentially caused by all the CPUs trying to access the same memory location at the same time. (We will use the term “broadcast” to refer equally to both cases.) Although hidden from the user, Fortran 90’s ability to do broadcasts is an essential feature of it as a parallel language.

Broadcasts can be more complicated than the above simple example. Consider, for example, the calculation

$$w_i = \sum_{j=1}^{n} |x_i + x_j|, \quad i = 1, \ldots, n$$  \hspace{1cm} (22.1.2)

Here, we are doing $n^2$ operations: For each of $n$ values of $i$ there is a sum over $n$ values of $j$.

Serial code for this calculation might be

```fortran
do i=1,n
    w(i)=0.
    do j=1,n
        w(i)=w(i)+abs(x(i)+x(j))
    end do
end do
```

The obvious immediate parallelization in Fortran 90 uses the `sum` intrinsic function to eliminate the inner do-loop. This would be a suitable amount of parallelization for a small-scale parallel machine, with a few processors:

```fortran
do i=1,n
    w(i)=sum(abs(x(i)+x))
end do
```

Notice that the conformability rule implies that a new value of $x(i)$, a scalar, is being broadcast to all the processors involved in the `abs` and `sum`, with each iteration of the loop over $i$.

What about the outer do-loop? Do we need, or want, to eliminate it, too? That depends on the architecture of your computer, and on the tradeoff between time and memory in your problem (a common feature of all computing, no less so parallel computing). Here is an implementation that is free of all do-loops, in principle capable of being executed in a small number (independent of $n$) of parallel operations:

```fortran
REAL(SP), DIMENSION(n,n) :: a...
a = spread(x,dim=2,ncopies=n)+spread(x,dim=1,ncopies=n)
w = sum(abs(a),dim=1)
```

This is an example of what we call dimensional expansion, as implemented by the `spread` intrinsic. Although the above may strike you initially as quite a cryptic construction, it is easy to learn to read it. In the first assignment line, a matrix is constructed with all possible values of $x(i)+x(j)$. In the second assignment line, this matrix is collapsed back to a vector by applying the sum operation to the absolute value of its elements, across one of its dimensions.
More explicitly, the first line creates a matrix $a$ by adding two matrices each constructed via \texttt{spread}. In \texttt{spread}, the $\texttt{dim}$ argument specifies which argument is duplicated, so that the first term varies across its first (row) dimension, and vice versa for the second term:

$$a_{ij} = x_i + x_j$$

\begin{equation}
\begin{pmatrix}
  x_1 & x_2 & x_3 & \cdots \\
  x_2 & x_2 & x_3 & \cdots \\
  x_3 & x_3 & x_3 & \cdots \\
  \vdots & \vdots & \vdots & \ddots \\
\end{pmatrix}
\begin{pmatrix}
  x_1 & x_2 & x_3 & \cdots \\
  x_1 & x_2 & x_3 & \cdots \\
  x_1 & x_2 & x_3 & \cdots \\
  \vdots & \vdots & \vdots & \ddots \\
\end{pmatrix}
\end{equation}

Since equation (22.1.2) above is symmetric in $i$ and $j$, it doesn’t really matter what value of $\texttt{dim}$ we put in the \texttt{sum} construction, but the value $\texttt{dim}=1$ corresponds to summing across the rows, that is, down each column of equation (22.1.3).

Be sure that you understand that the \texttt{spread} construction changed an $O(n)$ memory requirement into an $O(n^2)$ one! If your values of $n$ are large, this is an impossible burden, and the previous implementation with a single do-loop remains the only practical one. On the other hand, if you are working on a massively parallel machine, whose number of processors is comparable to $n^2$ (or at least much larger than $n$), then the \texttt{spread} construction, and the underlying broadcast capability that it invokes, leads to a big win: All $n^2$ operations can be done in parallel. This distinction between small-scale parallel machines — which we will hereafter refer to as \textit{SSP machines} — and massively multiprocessor machines — which we will refer to as \textit{MMP machines} — is an important one. A main goal of parallelism is to saturate the available number of processors, and algorithms for doing so are often different in the SSP and MMP opposite limits. Dimensional expansion is one method for saturating processors in the MMP case.

\textbf{Masks and “Index Loss”}

An instructive extension of the above example is the following case of a product that omits one term (the diagonal one):

$$w_i = \prod_{\substack{j=1 \atop j \neq i}}^n (x_j - x_i), \quad i = 1, \ldots, n$$

Formulas like equation (22.1.4) frequently occur in the context of interpolation, where all the $x_i$'s are known to be distinct, so let us for the moment assume that this is the case.

Serial code for equation (22.1.4) could be

```fortran
  do i=1,n
    w(i)=1.0_sp
    do j=1,n
      if (j /= i) w(i)=w(i)*(x(j)-x(i))
    end do
  end do
```

Parallel code for SSP machines, or for large enough $n$ on MMP machines, could be
do i=1,n
w(i)=product( x-x(i), mask=(x/=x(i)) )
end do

Here, the `mask` argument in the `product` intrinsic function causes the diagonal term to be omitted from the product, as we desire. There are some features of this code, however, that bear commenting on.

First, notice that, according to the rules of conformability, the expression `x/=x(i)` broadcasts the scalar `x(i)` and generates a logical array of length `n`, suitable for use as a `mask` in the `product` intrinsic. It is quite common in Fortran 90 to generate masks “on the fly” in this way, particularly if the mask is to be used only once.

Second, notice that the `j` index has disappeared completely. It is now implicit in the two occurrences of `x` (equivalent to `x(1:n)`) on the right-hand side. With the disappearance of the `j` index, we also lose the ability to do the test on `i` and `j`, but must use, in essence, `x(i)` and `x(j)` instead! That is a very general feature in Fortran 90: when an operation is done in parallel across an array, there is no associated index available within the operation. This “index loss,” as we will see in later discussion, can sometimes be quite an annoyance.

A language construction present in CM [Connection Machine] Fortran, the so-called `forall`, which would have allowed access to an associated index in many cases, was eliminated from Fortran 90 by the X3J3 committee, in a controversial decision. Such a construction will come into the language in Fortran 95.

What about code for an MMP machine, where we are willing to use dimensional expansion to achieve greater parallelism? Here, we can write,

```fortran
a = spread(x,dim=2,ncopies=n)-spread(x,dim=1,ncopies=n)
w = product(a,dim=1,mask=(a/=0.))
```

This time it does matter that the value of `dim` in the `product` intrinsic is 1 rather than 2. If you write out the analog of equation (22.1.3) for the present example, you’ll see that the above fragment is the right way around. The problem of index loss is still with us: we have to construct a mask from the array `a`, not from its indices, both of which are now lost to us!

In most cases, there are workarounds (more, or less, awkward as they may be) for the problem of index loss. In the worst cases, which are quite rare, you have to create objects to hold, and thus bring back into play, the lost indices. For example,

```fortran
INTEGER(I4B), DIMENSION(n) :: jj
...jj = (/ (i,i=1,n) /)
do i=1,n
w(i)=product(x-x(i), mask=(jj/=i))
end do
```

Now the array `jj` is filled with the “lost” `j` index, so that it is available for use in the mask. A similar technique, involving spreads of `jj`, can be used in the above MMP code fragment, which used dimensional expansion. (Fortran 95’s `forall` construction will make index loss much less of a problem. See §21.6.)

Incidentally, the above Fortran 90 construction, `/ (i,i=1,n) /`, is called an array constructor with implied do list. For reasons to be explained in §22.2, we almost never use this construction, in most cases substituting a Numerical Recipes utility function for generating arithmetical progressions, which we call `arth`. 
Interprocessor Communication Costs

It is both a blessing and a curse that Fortran 90 completely hides from the user the underlying machinery of interprocessor communication, that is, the way that data values computed by (or stored locally near) one CPU make their way to a different CPU that might need them next. The blessing is that, by and large, the Fortran 90 programmer need not be concerned with how this machinery works. If you write

\[ a(1:10,1:10) = b(1:10,1:10) + c(10:1:-1,10:1:-1) \]

the required upside-down-and-backwards values of the array \( c \) are just there, no matter that a great deal of routing and switching may have taken place. An ancillary blessing is that this book, unlike so many other (more highly technical) books on parallel programming (see references below) need not be filled with complex and subtle discussions of CPU connectivity, topology, routing algorithms, and so on.

The curse is, just as you might expect, that the Fortran 90 programmer can't control the interprocessor communication, even when it is desirable to do so. A few regular communication patterns are “known” to the compiler through Fortran 90 intrinsic functions, for example \( b = \text{transpose}(a) \). These, presumably, are done in an optimal way. However, many other regular patterns of communication, which might also allow highly optimized implementations, don't have corresponding intrinsic functions. (An obvious example is the “butterfly” pattern of communication that occurs in fast Fourier transforms.) These, if coded in Fortran 90 by using general vector subscripts (e.g., \( \text{barr}=\text{arr}(\text{iarr}) \) or \( \text{barr}(\text{jarr})=\text{arr} \), where \( \text{iarr} \) and \( \text{jarr} \) are integer arrays), lose all possibility of being optimized. The compiler can’t distinguish a communication step with regular structure from one with general structure, so it must assume the worst case, potentially resulting in very slow execution.

About the only thing a Fortran 90 programmer can do is to start with a general awareness of the kind of apparently parallel constructions that \textit{might} be quite slow on his/her parallel machine, and then to refine that awareness by actual experience and experiment. Here is our list of constructions most likely to cause interprocessor communication bottlenecks:

- vector subscripts, like \( \text{barr}=\text{arr}(\text{iarr}) \) or \( \text{barr}(\text{jarr})=\text{arr} \) (that is, general gather/scatter operations)
- the \text{pack} and \text{unpack} intrinsic functions
- mixing positive strides and negative strides in a single expression (as in the above \( b(1:10,1:10)+c(10:1:-1,10:1:-1) \))
- the \text{reshape} intrinsic when used with the \text{order} argument
- possibly, the \text{cshift} and \text{eoshift} extrinsics, especially for nonsmall values of the shift.

On the other hand, the fact is that these constructions are parallel, and are there for you to use. If the alternative to using them is strictly serial code, you should almost always give them a try.

Linear Algebra

You should be alert for opportunities to use combinations of the \text{matmul}, \text{spread}, and \text{dot_product} intrinsics to perform complicated linear algebra calculations. One useful intrinsic that is not provided in Fortran 90 is the \text{outer product}
of two vectors,
\[ c_{ij} = a_i b_j \]  
(22.1.5)

We already know how to implement this (cf. equation 22.1.3):
\[ c = \text{spread}(a, \text{dim}=2, \text{ncopies}=\text{size}(b)) \ast \text{spread}(b, \text{dim}=1, \text{ncopies}=\text{size}(a)) \]

In fact, this operation occurs frequently enough to justify making it a utility function, `outerprod`, which we will do in Chapter 23. There we also define other “outer” operations between vectors, where the multiplication in the outer product is replaced by another binary operation, such as addition or division.

Here is an example of using these various functions: Many linear algebra routines require that a submatrix be updated according to a formula like
\[ a_{jk} = a_{jk} + b_i a_{p \iota} \sum_{p=i}^m a_{p \iota} a_{pk}, \quad j = i, \ldots, m, \quad k = l, \ldots, n \]  
(22.1.6)

where \( i, m, l, \) and \( n \) are fixed values. Using an array slice like \( a(:, i) \) to turn \( a_{p \iota} \) into a vector indexed by \( p \), we can code the sum with a `matmul`, yielding a vector indexed by \( k \):
\[ \text{temp}(1:n) = b(i) \ast \text{matmul}(a(i:m, i), a(i:m, l:n)) \]

Here we have also included the multiplication by \( b_i \), a scalar for fixed \( i \). The vector `temp`, along with the vector \( a_{ji} = a(:, i) \), is then turned into a matrix by the `outerprod` utility and used to increment \( a_{jk} \):
\[ a(i:m, l:n) = a(i:m, l:n) + \text{outerprod}(a(i:m, i), \text{temp}(1:n)) \]

Sometimes the update formula is similar to (22.1.6), but with a slight permutation of the indices. Such cases can be coded as above if you are careful about the order of the quantities in the `matmul` and the `outerprod`.

CITED REFERENCES AND FURTHER READING:
22.2 Linear Recurrence and Related Calculations

We have already seen that Fortran 90's array constructor with implied do list can be used to generate simple series of integers, like \( / (i, i=1, n) / \). Slightly more generally, one might want to generate an arithmetic progression, by the formula

\[
v_j = b + (j - 1)a, \quad j = 1, \ldots, n
\]  

(22.2.1)

This is readily coded as

\[
v(1:n) = / (b+(j-1)*a, j=1,n) /
\]

Although it is concise, and valid, we don't like this coding. The reason is that it violates the fundamental rule of "thinking parallel": it turns a parallel operation across a data vector into a serial do-loop over the components of that vector. Yes, we know that the compiler might be smart enough to generate parallel code for implied do lists; but it also might not be smart enough, here or in more complicated examples.

Equation (22.2.1) is also the simplest example of a linear recurrence relation.

It can be rewritten as

\[
v_1 = b, \quad v_j = v_{j-1} + a, \quad j = 2, \ldots, n
\]  

(22.2.2)

In this form (assuming that, in more complicated cases, one doesn’t know an explicit solution like equation 22.2.1) one can’t write an explicit array constructor. Code like

\[
v(1) = b \\
v(2:n) = / (v(j-1)+a, j=2,n) /
\]

is legal Fortran 90 syntax, but illegal semantics; it does not do the desired recurrence! (The rules of Fortran 90 require that all the components of \( v \) on the right-hand side be evaluated before any of the components on the left-hand side are set.) Yet, as we shall see, techniques for accomplishing the evaluation in parallel are available.

With this as our starting point, we now survey some particular tricks of the (parallel) trade.
Subvector Scaling: Arithmetic and Geometric Progressions

For explicit arithmetic progressions like equation (22.2.1), the simplest parallel technique is subvector scaling [1]. The idea is to work your way through the desired vector in larger and larger parallel chunks:

\[
\begin{align*}
    v_1 &= b \\
    v_2 &= v_1 + a \\
    v_3 &= v_1 + 2a \\
    v_5 &= v_1 + 4a \\
    v_9 &= v_1 + 8a
\end{align*}
\]

And so on, until you reach the length of your vector. (The last step will not necessarily go all the way to the next power of 2, therefore.) The powers of 2, times \(a\), can of course be obtained by successive doublings, rather than the explicit multiplications shown above.

You can see that subvector scaling requires about \(\log_2 n\) parallel steps to process a vector of length \(n\). Equally important for serial machines, or SSP machines, the scalar operation count for subvector scaling is no worse than entirely serial code: each new component \(v_i\) is produced by a single addition.

If addition is replaced by multiplication, the identical algorithm will produce geometric progressions, instead of arithmetic progressions. In Chapter 23, we will use subvector scaling to implement our utility functions \(\text{arith}\) and \(\text{geop}\) for these two progressions. (You can then call one of these functions instead of recoding equation 22.2.3 every time you need it.)

Vector Reduction: Evaluation of Polynomials

Logically related to subvector scaling is the case where a calculation can be parallelized across a vector that shrinks by a factor of 2 in each iteration, until a desired scalar result is reached. A good example of this is the parallel evaluation of a polynomial [2]

\[
P(x) = \sum_{j=0}^{N} c_j x^j
\]  

(22.2.4)

For clarity we take the special case of \(N = 5\). Start with the vector of coefficients (imagining appended zeros, as shown):

\[c_0, c_1, c_2, c_3, c_4, c_5, 0, \ldots\]

Now, add the elements by pairs, multiplying the second of each pair by \(x\):

\[c_0 + c_1 x, c_2 + c_3 x, c_4 + c_5 x, 0, \ldots\]

Now, the same operation, but with the multiplier \(x^2\):

\[(c_0 + c_1 x) + (c_2 + c_3 x)x^2, (c_4 + c_5 x) + (0)x^2, 0, \ldots\]
And a final time, with multiplier $x^4$:

$$[(c_0 + c_1 x) + (c_2 + c_3 x)x^2] + [(c_4 + c_5 x) + (0)x^2]x^4, \quad 0, \ldots$$

We are left with a vector of (active) length 1, whose value is the desired polynomial evaluation. (You can see that the zeros are just a bookkeeping device for taking account of the case where the active subvector has odd length.) The key point is that the combining by pairs is a parallel operation at each stage.

As in subvector scaling, there are about $\log_2 n$ parallel stages. Also as in subvector scaling, our total operations count is only negligibly different from purely scalar code: We do one add and one multiply for each original coefficient $c_j$. The only extra operations are $\log_2 n$ successive squarings of $x$; but this comes with the extra benefit of better roundoff properties than the standard scalar coding. In Chapter 23 we use vector reduction to implement our utility function $\text{poly}$ for polynomial evaluation.

**Recursive Doubling: Linear Recurrence Relations**

Please don’t confuse our use of the word “recurrence” (as in “recurrence relation,” “linear recurrence,” or equation 22.2.2) with the words “recursion” and “recursive,” which both refer to the idea of a subroutine calling itself to obtain an efficient or concise algorithm. There are ample grounds for confusion, because recursive algorithms are in fact a good way of obtaining parallel solutions to linear recurrence relations, as we shall now see!

Consider the general first order linear recurrence relation

$$u_j = a_j + b_{j-1}u_{j-1}, \quad j = 2, 3, \ldots, n$$

(22.2.5)

with initial value $u_1 = a_1$. On a serial machine, we evaluate such a recurrence with a simple do-loop. To parallelize the recurrence, we can employ the powerful general strategy of recursive doubling. Write down equation (22.2.5) for $2j$ and for $2j - 1$:

$$u_{2j} = a_{2j} + b_{2j-1}u_{2j-1}$$

(22.2.6)

$$u_{2j-1} = a_{2j-1} + b_{2j-2}u_{2j-2}$$

(22.2.7)

Substitute equation (22.2.7) in equation (22.2.6) to eliminate $u_{2j-1}$ and get

$$u_{2j} = (a_{2j} + a_{2j-1}b_{2j-1}) + (b_{2j-2}b_{2j-1})u_{2j-2}$$

(22.2.8)

This is a new recurrence of the same form as (22.2.5) but over only the even $u_j$, and hence involving only $n/2$ terms. Clearly we can continue this process recursively, halving the number of terms in the recurrence at each stage, until we are left with a recurrence of length 1 or 2 that we can do explicitly. Each time we finish a subpart of the recursion, we fill in the odd terms in the recurrence, using equation (22.2.7). In practice, it’s even easier than it sounds. Turn to Chapter B5 to see a straightforward implementation of this algorithm as the recipe recur1.

On a machine with more processors than $n$, all the arithmetic at each stage of the recursion can be done simultaneously. Since there are of order $\log n$ stages in the
recursion, the execution time is \(O(\log n)\). The total number of operations carried out is of order \(n + n/2 + n/4 + \cdots = O(n)\), the same as for the obvious serial do-loop.

In the utility routines of Chapter 23, we will use recursive doubling to implement the routines \texttt{poly}_term, \texttt{cumsum}, and \texttt{cumprod}. We could use recursive doubling to implement parallel versions of \texttt{arth} and \texttt{geop} (arithmetic and geometric progressions), and \texttt{zrootsUnity} (complex \(n\)th roots of unity), but these can be done slightly more efficiently by subvector scaling, as discussed above.

**Cyclic Reduction: Linear Recurrence Relations**

There is a variant of recursive doubling, called cyclic reduction, that can be implemented with a straightforward iteration loop, instead of a recursive procedure call. Here we start by writing down the recurrence (22.2.5) for all adjacent terms \(u_j\) and \(u_{j-1}\) (not just the even ones, as before). Eliminating \(u_{j-1}\), just as in equation (22.2.8), gives

\[
  u_j = (a_j + a_{j-1}b_{j-1}) + (b_{j-2}b_{j-1})u_{j-2}
\]

which is a first order recurrence with new coefficients \(a'_j\) and \(b'_j\). Repeating this process gives successive formulas for \(u_j\) in terms of \(u_{j-2}\), \(u_{j-4}\), \(u_{j-8}\),... The procedure terminates when we reach \(u_{j-n}\) (for \(n\) a power of 2), which is zero for all \(j\). Thus the last step gives \(u_j\) equal to the last set of \(a'_j\)'s.

Here is a code fragment that implements cyclic reduction by direct iteration. The quantities \(a'_j\) are stored in the variable \texttt{recur1}.

```c
recur1=a
bb=b
j=1
do if (j >= n) exit
   recur1(j+1:n)=recur1(j+1:n)+bb(j:n-1)*recur1(1:n-j)
   bb(2*j:n-1)=bb(2*j:n-1)*bb(j:n-j-1)
j=2*j
endo
```

In cyclic reduction the length of the vector \(u_j\) that is updated at each stage does not decrease by a factor of 2 at each stage, but rather only decreases from \(\sim n\) to \(\sim n/2\) during all \(\log_2 n\) stages. Thus the total number of operations carried out is \(O(n \log n)\), as opposed to \(O(n)\) for recursive doubling. For a serial machine or SSP machine, therefore, cyclic reduction is rarely superior to recursive doubling when the latter can be used. For an MMP machine, however, the issue is less clear cut, because the pattern of communication in cyclic reduction is quite different (and, for some parallel architectures, possibly more favorable) than that of recursive doubling.

**Second Order Recurrence Relations**

Consider the second order recurrence relation

\[
y_j = a_j + b_{j-2}y_{j-1} + c_{j-2}y_{j-2}, \quad j = 3, 4, \ldots, n
\]

with initial values

\[
y_1 = a_1, \quad y_2 = a_2
\]
Our labeling of subscripts is designed to make it easy to enter the coefficients in a computer program: You need to supply $a_1, \ldots, a_n, b_1, \ldots, b_{n-2}$, and $c_1, \ldots, c_{n-2}$. Rewrite the recurrence relation in the form (22.2.12)

$$
\begin{pmatrix}
y_j \\
y_{j+1}
\end{pmatrix} =
\begin{pmatrix}
0 & 1 \\
-1 & 0
\end{pmatrix}
\begin{pmatrix}
y_{j-1} \\
y_j
\end{pmatrix},
$$

that is,

$$
u_j = a_j + b_{j-1} \cdot u_{j-1}, \quad j = 2, \ldots, n-1 (22.2.13)
$$

where

$$
u_j = \begin{pmatrix} y_j & y_{j+1} \end{pmatrix}, \quad a_j = \begin{pmatrix} 0 & 1 \end{pmatrix}, \quad b_{j-1} = \begin{pmatrix} c_{j-1} & 1 \end{pmatrix}, \quad j = 2, \ldots, n-1 (22.2.14)
$$

and

$$
u_1 = a_1 = \begin{pmatrix} y_1 & y_2 \end{pmatrix} = \begin{pmatrix} a_1 & \end{pmatrix} (22.2.15)
$$

This is a first order recurrence relation for the vectors $u_j$, and can be solved by the algorithm described above (and implemented in the recipe recur1). The only difference is that the multiplications are matrix multiplications with the $2 \times 2$ matrices $b_j$. After the first recursive call, the zeros in $a$ and $b$ are lost, so we have to write the routine for general two-dimensional vectors and matrices.

Note that this algorithm does not avoid the potential instability problems associated with second order recurrences that are discussed in §5.5 of Volume 1. Also note that the algorithm generalizes in the obvious way to higher-order recurrences: An $n$th order recurrence can be written as a first order recurrence involving $n$-dimensional vectors and matrices.

Parallel Solution of Tridiagonal Systems

Closely related to recurrence relations, recursive doubling, and cyclic reduction is the parallel solution of tridiagonal systems. Since Fortran 90 vectors “know their own size,” it is most logical to number the components of both the sub- and super-diagonals of the tridiagonal matrix from 1 to $N-1$. Thus equation (2.4.1), here written in the special case of $N = 7$, becomes (blank elements denoting zero),

$$
\begin{bmatrix}
b_1 & c_1 \\
a_1 & b_2 & c_2 \\
a_2 & b_3 & c_3 \\
a_3 & b_4 & c_4 \\
a_4 & b_5 & c_5 \\
a_5 & b_6 & c_6 \\
a_6 & b_7
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_2 \\
u_3 \\
u_4 \\
u_5 \\
u_6 \\
u_7
\end{bmatrix} =
\begin{bmatrix}
r_1 \\
r_2 \\
r_3 \\
r_4 \\
r_5 \\
r_6 \\
r_7
\end{bmatrix} (22.2.16)
$$

The basic idea for solving equation (22.2.16) on a parallel computer is to partition the problem into even and odd elements, recurse to solve the former, and
then solve the latter in parallel. Specifically, we first rewrite (22.2.16), by permuting its rows and columns, as

\[
\begin{bmatrix}
  b_1 & c_1 & a_2 & c_3 & b_5 & a_4 & c_5 & b_7 & a_6 \\
  a_1 & c_2 & b_3 & a_2 & c_3 & b_5 & a_4 & c_5 & b_7 \\
  a_3 & c_4 & b_5 & a_4 & c_5 & b_7 & a_6 \\
  a_5 & c_6 & b_7 & a_6 & c_5 & b_7 & a_6
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_3 \\
u_5 \\
u_7 \\
u_2 \\
u_4 \\
u_6 
\end{bmatrix}
= \begin{bmatrix}
r_1 \\
r_3 \\
r_5 \\
r_7 \\
r_2 \\
r_4 \\
r_6
\end{bmatrix}
\] (22.2.17)

Now observe that, by row operations that subtract multiples of the first four rows from each of the last three rows, we can eliminate all nonzero elements in the lower-left quadrant. The price we pay is bringing some new elements into the lower-right quadrant, whose nonzero elements we now call \( x \)'s, \( y \)'s, and \( z \)'s. We call the modified right-hand sides \( q \). The transformed problem is now

\[
\begin{bmatrix}
b_1 & c_1 & a_2 & c_3 & b_5 & a_4 & c_5 & b_7 & a_6 \\
  y_1 & z_1 & x_1 & y_2 & z_2 & x_2 & y_3 \\
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_3 \\
\end{bmatrix}
= \begin{bmatrix}
r_1 \\
r_3 \\
\end{bmatrix}
\] (22.2.18)

Notice that the last three rows form a new, smaller, tridiagonal problem, which we can solve simply by recursing! Once its solution is known, the first four rows can be solved by a simple, parallelizable, substitution. This algorithm is implemented in \texttt{tridag} in Chapter B2.

The above method is essentially cyclic reduction, but in the case of the tridiagonal problem, it does not “unwind” into a simple iteration; on the contrary, a recursive subroutine is required. For discussion of this and related methods for parallelizing tridiagonal systems, and references to the literature, see Hockney and Jesshope [3].

Recursive doubling can also be used to solve tridiagonal systems, the method requiring the parallel solution (as above) of both a first order recurrence and a second order recurrence [3,4]. For tridiagonal systems, however, cyclic reduction is usually more efficient than recursive doubling.

CITED REFERENCES AND FURTHER READING:


Hockney, R.W., and Jesshope, C.R. 1988, \textit{Parallel Computers 2: Architecture, Programming, and Algorithms} (Bristol and Philadelphia: Adam Hilger), §5.2.4 (cyclic reduction); §5.4.2 (second order recurrences); §5.4 (tridiagonal systems). [3]

22.3 Parallel Synthetic Division and Related Algorithms

There are several techniques for parallelization that relate to synthetic division but that can also find application in wider contexts, as we shall see.

Cumulants of a Polynomial

Suppose we have a polynomial

\[ P(x) = \sum_{j=0}^{N} c_j x^{N-j} \quad (22.3.1) \]

(Note that, here, the \( c_j \)'s are indexed from highest degree to lowest, the reverse of the usual convention.) Then we can define the cumulants of the polynomial to be partial sums that occur in the polynomial's usual, serial evaluation,

\[ P_0 = c_0 \]
\[ P_1 = c_0 x + c_1 \]
\[ \vdots \]
\[ P_N = c_0 x^N + \cdots + c_N = P(x) \quad (22.3.2) \]

Evidently, the cumulants satisfy a simple, linear first order recurrence relation,

\[ P_0 = c_0, \quad P_j = c_j + x P_{j-1}, \quad j = 2, \ldots, N \quad (22.3.3) \]

This is slightly simpler than the general first order recurrence, because the value of \( x \) does not depend on \( j \). We already know, from §22.2's discussion of recursive doubling, how to parallelize equation (22.3.3) via a recursive subroutine. In Chapter 23, the utility routine `poly_term` will implement just such a procedure. An example of a routine that calls `poly_term` to evaluate a recurrence equivalent to equation (22.3.3) is `eulsum` in Chapter B5.

Notice that while we could use equation (22.3.3), parallelized by recursive doubling, simply to evaluate the polynomial \( P(x) = P_N \), this is likely somewhat slower than the alternative technique of vector reduction, also discussed in §22.2, and implemented in the utility function `poly`. Equation (22.3.3) should be saved for cases where the rest of the \( P_j \)'s (not just \( P_N \)) can be put to good use.

Synthetic Division by a Monomial

We now show that evaluation of the cumulants of a polynomial is equivalent to synthetic division of the polynomial by a monomial, also called deflation (see §9.5 in Volume 1). To review briefly, and by example, here is a standard tableau from high school algebra for the (long) division of a polynomial \( 2x^3 - 7x^2 + x + 3 \) by the monomial factor \( x - 3 \).
Now, here is the same calculation written as a *synthetic division*, really the same procedure as tableau (22.3.4), but with unnecessary notational baggage omitted (and also a changed sign for the monomial’s constant, so that subtractions become additions):

\[
\begin{array}{c|cccc}
  & 2x^2 & -x & -2 \\
\hline
x-3 & 2x^3 & -7x^2 & +x & +3 \\
 & 2x^3 & -6x^2 & \\
 & -x^2 & +x & \\
 & -x^2 & +3x & \\
 & -2x & +3 & \\
 & -2x & +6 & \\
 & & & -3 \text{ (remainder)}
\end{array}
\] (22.3.4)

If we substitute symbols for the above quantities with the correspondence

\[
\begin{array}{c|cccc}
x & c_0 & c_1 & c_2 & c_3 \\
\hline
P_0 & P_1 & P_2 & P_3
\end{array}
\] (22.3.6)

then it is immediately clear that the \(P_j\)'s in equation (22.3.6) are simply the \(P_j\)'s of equation (22.3.3); the calculation is thus parallelizable by recursive doubling. In this context, the utility routine \texttt{poly\_term} is used by the routine \texttt{zroots} in Chapter B9.

### Repeated Synthetic Division

It is well known from high-school algebra that repeated synthetic division of a polynomial yields, as the remainders that occur, first the value of the polynomial, next the value of its first derivative, and then (up to multiplication by the factorial of an integer) the values of higher derivatives.

If you want to parallelize the calculation of the value of a polynomial and one or two of its derivatives, it is not unreasonable to evaluate equation (22.3.3), parallelized by recursive doubling, two or three times. Our routine \texttt{ddpoly} in Chapter B5 is meant for such use, and it does just this, as does the routine \texttt{laguer} in Chapter B9.

There are other cases, however, for which you want to perform repeated synthetic division and “go all the way,” until only a constant remains. For example, this is the preferred way of “shifting a polynomial,” that is, evaluating the coefficients of a polynomial in a variable \(y\) that differs from the original variable \(x\) by an additive constant. (The recipe \texttt{pcshft} has this as its assigned task.) By way of example, consider the polynomial \(3x^3 + x^2 + 4x + 7\), and let us perform repeated synthetic division by a general monomial \(x - a\). The conventional calculation then proceeds according to the following tableau, reading it in conventional lexical order (left-to-right and top-to-bottom):
Here, each row (after the first) shows a synthetic division or, equivalently, evaluation of the cumulants of the polynomial whose coefficients are the preceding row. The results at the right edge of the rows are the values of the polynomial and (up to integer factorials) its three nonzero derivatives, or (equivalently, without factorials) coefficients of the shifted polynomial.

We could parallelize the calculation of each row of tableau (22.3.7) by recursive doubling. That is a lot of recursion, which incurs a nonnegligible overhead. A much better way of doing the calculation is to deform tableau (22.3.7) into the following equivalent tableau,

\[
\begin{array}{cccc}
3 & 1 & 4 & 7 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
3 & \rightarrow & 3a + 1 & \rightarrow & 3a^2 + a + 4 & \rightarrow & 3a^3 + a^2 + 4a + 7 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
3 & \rightarrow & 6a + 1 & \rightarrow & 9a^2 + 2a + 4 \\
\downarrow & \downarrow \\
3 & \rightarrow & 9a + 1 \\
\downarrow \\
3
\end{array}
\]  
(22.3.7)

Now each row explicitly depends on only the previous row (and the given first column), so the rows can be calculated in turn by an explicit parallel expression, with no recursive calls needed. An example of coding (22.3.8) in Fortran 90 can be found in the routine `pcshft` in Chapter B5. (It is also possible to eliminate most of the multiplications in (22.3.8), at the expense of a much smaller number of divisions. We have not done this because of the necessity for then treating all possible divisions by zero as special cases. See [1] for details and references.)

Actually, the deformation of (22.3.7) into (22.3.8) is the same trick as was used in Volume 1, p. 167, for evaluating a polynomial and its derivative simultaneously, also generalized in the Fortran 77 implementation of the routine `ddpoly` (Chapter 5). In the Fortran 90 implementation of `ddpoly` (Chapter B5) we don’t use this trick, but instead use `polym_term`, because, there, we want to parallelize over the length of the polynomial, not over the number of desired derivatives.
Don’t confuse the cases of *iterated* synthetic division, discussed here, with the simpler case of doing many simultaneous synthetic divisions. In the latter case, you can simply implement equation (22.3.3) serially, exactly as written, but with each operation being data-parallel across your problem set. (This case occurs in our routine `polcoe` in Chapter B3.)

**Polynomial Coefficients from Roots**

A parallel calculation algorithmically very similar to (22.3.7) or (22.3.8) occurs when we want to find the coefficients of a polynomial $P(x)$ from its roots $r_1, \ldots, r_N$. For this, the tableau is

\[
\begin{array}{cccc}
 & r_1 & & \\
 r_2: & r_1 + r_2 & r_1r_2 & \\
 r_3: & r_1 + r_2 + r_3 & r_1r_2 + r_3(r_1 + r_2) & r_1r_2r_3 \\
\end{array}
\]

(22.3.9)

As before, the rows are computed consecutively, from top to bottom. Each row is computed via a single parallel expression. Note that values moving on vertical arrows are simply added in, while values moving on diagonal arrows are multiplied by a new root before adding. Examples of coding (22.3.9) in Fortran 90 can be found in the routines `vander` (Chapter B2) and `polcoe` (Chapter B3).

An equivalent deformation of (22.3.9) is

\[
\begin{array}{cccc}
 & r_1 & & \\
r_2: & r_1r_2 & r_1 + r_2 & \\
r_3: & r_1r_2r_3 & r_1r_2 + r_3(r_1 + r_2) & r_1 + r_2 + r_3 \\
\end{array}
\]

(22.3.10)

Here the diagonal arrows are simple additions, while the vertical arrows represent multiplication by a root value. Note that the coefficient answers in (22.3.10) come out in the opposite order from (22.3.9). An example of coding (22.3.10) in Fortran 90 can be found in the routine `fixrts` in Chapter B13.

CITED REFERENCES AND FURTHER READING:
22.4 Fast Fourier Transforms

Fast Fourier transforms are beloved by computer scientists, especially those who are interested in parallel algorithms, because the FFT’s hierarchical structure generates a complicated, but analyzable, set of requirements for interprocessor communication on MMPs. Thus, almost all books on parallel algorithms (e.g., [1–3]) have a chapter on FFTs.

Unfortunately, the resulting algorithms are highly specific to particular parallel architectures, and therefore of little use to us in writing general purpose code in an architecture-independent parallel language like Fortran 90.

Luckily there is a good alternative that covers almost all cases of both serial and parallel machines. If, for a one-dimensional FFT of size \( N \), one is satisfied with parallelism of order \( \sqrt{N} \), then there is a good, general way of achieving a parallel FFT with quite minimal interprocessor communication; and the communication required is simply the matrix transpose operation, which Fortran 90 implements as an intrinsic. That is the approach that we discuss in this section, and implement in Chapter B12.

For a machine with \( M \) processors, this approach will saturate the processors (the desirable condition where none are idle) when the size of a one-dimensional Fourier transform, \( N \), is large enough: \( N > M^2 \). Smaller \( N \)'s will not achieve maximum parallelism. But such \( N \)'s are in fact so small for one-dimensional problems that they are unlikely to be the rate-determining step in scientific calculations. If they are, it is usually because you are doing many such transforms independently, and you should recover “outer parallelism” by doing them all at once.

For two or more dimensions, the adopted approach will saturate \( M \) processors when each dimension of the problem is larger than \( M \).

Column- and Row-Parallel FFTs

The basic building block that we assume (and implement in Chapter B12) is a routine for simultaneously taking the FFT of each row of a two-dimensional matrix. The method is exactly that of Volume 1’s four1 routine, but with array sections like \( \text{data}(\cdot,j) \) replacing scalars like \( \text{data}(j) \). Chapter B12’s implementation of this is a routine called fourrow. If all the data for one column (that is, all the values \( \text{data}(i,:) \), for some \( i \)) are local to a single processor, then the parallelism involves no interprocessor communication at all: The independent FFTs simply proceed, data parallel and in lockstep. This is architecture-independent parallelism with a vengeance.

We will also need to take the FFT of each column of a two-dimensional matrix. One way to do this is to take the transpose (a Fortran 90 intrinsic that hides a lot of interprocessor communication), then take the FFT of the rows using fourrow, then take the transpose again. An alternative method is to recode the four1 routine with array sections in the other dimension (\( \text{data}(j,:) \)) replacing four1’s scalars (\( \text{data}(j) \)). This scheme, in Chapter B12, is a routine called fourcol. In this case, good parallelism will be achieved only if the values \( \text{data}(:,i) \), for some \( i \), are local to a single processor. Of course, Fortran 90 does not give the user direct control over how data are distributed over the machine; but extensions such as HPF are designed to give just such control.
On a serial machine, you might think that `fourrow` and `fourcol` should have identical timings (acting on a square matrix, say). The two routines do exactly the same operations, after all. Not so! On modern serial computers, `fourrow` and `fourcol` can have timings that differ by a factor of 2 or more, even when their detailed arithmetic is made identical (by giving to one a data array that is the transpose of the data array given to the other). This effect is due to the multilevel cache architecture of most computer memories, and the fact that serial Fortran always stores matrices by columns (first index changing most rapidly). On our workstations, `fourrow` is significantly faster than `fourcol`, and this is likely the generic behavior. However, we do not exclude the possibility that some machines, and some sizes of matrices, are the other way around.

**One-Dimensional FFTs**

Turn now to the problem of how to do a single, one-dimensional, FFT. We are given a complex array \( f \) of length \( N \), an integer power of 2. The basic idea is to address the input array as if it were a two-dimensional array of size \( m \times M \), where \( m \) and \( M \) are each integer powers of 2. Then the components of \( f \) can be addressed as

\[
f(Jm + j), \quad 0 \leq j < m, \quad 0 \leq J < M
\]

(22.4.1)

where the \( j \) index changes more rapidly, the \( J \) index more slowly, and parentheses denote Fortran-style subscripts.

Now, suppose we had some magical (parallel) method to compute the discrete Fourier transform

\[
F(kM + K) = \sum_{j, J} e^{2\pi i (kM + K)(Jm + j)/(Mm)} f(Jm + j),
\]

(22.4.2)

Then, you can see that the indices \( k \) and \( K \) would address the desired result (FFT of the original array), with \( K \) varying more rapidly.

Starting with equation (22.4.2) it is easy to verify the following identity,

\[
F(kM + K) = \sum_j \left[ e^{2\pi i k j / m} \left( e^{2\pi i K / (Mm)} \sum_J e^{2\pi i j K / M} f(Jm + j) \right) \right]
\]

(22.4.3)

But this, reading it from the innermost operation outward, is just the magical method that we need:

- Reshape the original array to \( m \times M \) in Fortran normal order (storage by columns).
- FFT on the second (column) index for all values of the first (row) index, using the routine `fourrow`.
- Multiply each component by a phase factor \( \exp[2\pi i j K / (Mm)] \).
- Transpose.
22.5 Missing Language Features

A few facilities that are fairly important to parallel programming are missing from the Fortran 90 language standard. On scalar machines this lack is not a
problem, since one can readily program the missing features by using do-loops. On parallel machines, both SSP machines and MMP machines, one must hope that hardware manufacturers provide library routines, callable from Fortran 90, that provide access to the necessary facilities, or use extensions of Fortran 90, such as High Performance Fortran (HPF).

**Scatter-with-Combine Functions**

Fortran 90 allows the use of *vector subscripts* for so-called *gather* and *scatter* operations. For example, with the setup

```fortran
REAL(SP), DIMENSION(6) :: arr, barr, carr
INTEGER(I4B), DIMENSION(6) :: iarr, jarr

iarr = (/ 1,3,5,2,4,6 /)
jarr = (/ 3,2,3,2,1,1 /)
```

Fortran 90 allows both the *one-to-one* gather and the *one-to-many* gather,

- `barr=arr(iarr)`
- `carr=arr(jarr)`

It also allows the one-to-one scatter,

- `barr(iarr)=carr`

where the elements of `carr` are “scattered” into `barr` under the direction of the vector subscript `iarr`.

Fortran 90 does *not* allow the *many-to-one* scatter

- `barr(jarr)=carr` ! illegal for this `jarr`

because the repeated values in `jarr` try to assign different components of `carr` to the same location in `barr`. The result would not be deterministic.

Sometimes, however, one would in fact like a many-to-one construction, where the colliding elements get combined by a (commutative and associative) operation, like `+` or `*`, or `max()`. These so-called *scatter-with-combine* functions are readily implemented on serial machines by a do-loop, for example,

```fortran
barr=0.
do j=1,size(carr)
   barr(jarr(j))=barr(jarr(j))+carr(j)
end do
```

Fortran 90 unfortunately provides no means for effecting scatter-with-combine functions in parallel. Luckily, almost all parallel machines do provide such a facility as a library program, as does HPF, where the above facility is called `SUM_SCATTER`. In Chapter 23 we will define utility routines `scatter_add` and `scatter_max` for scatter-with-combine functionalities, but the implementation given in Fortran 90 will be strictly serial, with a do-loop.
Skew Sections

Fortran 90 provides no good, parallel way to access the diagonal elements of a matrix, either to read them or to set them. Do-loops will obviously serve this need on serial machines. In principle, a construction like the following bizarre fragment could also be utilized,

```fortran
REAL(SP), DIMENSION(n,n) :: mat
REAL(SP), DIMENSION(n*n) :: arr
REAL(SP), DIMENSION(n) :: diag

... 
arr = reshape(mat,shape(arr))
diag = arr(1:n*n:n+1)
```

which extracts every \((n + 1)st\) element from a one-dimensional array derived by reshaping the input matrix. However, it is unlikely that any foreseeable parallel compiler will implement the above fragment without a prohibitive amount of unnecessary data movement; and code like the above is also exceedingly slow on all serial machines that we have tried.

In Chapter 23 we will define utility routines `get_diag`, `put_diag`, `diagadd`, `diagmult`, and `unit_matrix` to manipulate diagonal elements, but the implementation given in Fortran 90 will again be strictly serial, with do-loops.

Fortran 95 (see §21.6) will essentially fix Fortran 90’s skew sections deficiency. For example, using its `forall` construction, the diagonal elements of an array can be accessed by a statement like

```fortran
forall (j=1:20) diag(j) = arr(j,j)
```

SIMD vs. MIMD

Recall that we use “SIMD” (single-instruction, multiple data) and “data parallel” as interchangeable terms, and that “MIMD” (multiple-instruction, multiple data) is a more general programming model. (See §22.1.)

You should not be too quick to jump to the conclusion that Fortran 90’s data parallel or SIMD model is “bad,” and that MIMD features, absent in Fortran 90, are therefore “good.” On the contrary, Fortran 90’s basic data-parallel paradigm has a lot going for it. As we discussed in §22.1, most scientific problems naturally have a “data dimension” across which the time ordering of the calculation is irrelevant. Parallelism across this dimension, which is by nature most often SIMD, frees the mind to think clearly about the computational steps in an algorithm that actually need to be sequential. SIMD code has advantages of clarity and predictability that should not be taken lightly. The general MIMD model of “lots of different things all going on at the same time and communicating data with each other” is a programming and debugging nightmare.

Having said this, we must at the same time admit that a few MIMD features — most notably the ability to go through different logical branches for calculating different data elements in a data-parallel computation — are badly needed in certain programming situations. Fortran 90 is quite weak in this area.

Note that the `where...elsewhere...end where` construction is _not_ a MIMD construction. Fortran 90 requires that the `where` clause be executed completely before the `elsewhere` is started. (This allows the results of any calculations in the former
clause to be available for use in the latter.) So, this construction cannot be used to allow two logical branches to be calculated in parallel.

Special functions, where one would like to calculate function values for an array of input quantities, are a particularly compelling example of the need for some MIMD access. Indeed, you will find that Chapter B6 contains a number of intricate, and in a few cases truly bizarre, workarounds, using allowed combinations of merge, where, and CONTAINS (the latter, for separating different logical branches into formally different subprograms).

Fortran 95’s ELEMENTAL and PURE constructions, and to some extent also forall (whose body will be able to include PURE function calls), will go a long way towards providing exactly the kind of MIMD constructions that are most needed. Once Fortran 95 becomes available and widespread, you can expect to see a new version of this volume, with a much-improved Chapter B6.

Conversely, the number of routines outside of Chapter B6 that can be significantly improved by the use of MIMD features is relatively small; this illustrates the underlying viability of the basic data-parallel SIMD model, even in a future language version with useful MIMD features.
Chapter 23. Numerical Recipes
Utility Functions for Fortran 90

23.0 Introduction and Summary Listing

This chapter describes and summarizes the Numerical Recipes utility routines that are used throughout the rest of this volume. A complete implementation of these routines in Fortran 90 is listed in Appendix C1.

Why do we need utility routines? Aren't there already enough of them built into the language as Fortran 90 intrinsics? The answers lie in this volume's dual purpose: to implement the Numerical Recipes routines in Fortran 90 code that runs efficiently on fast serial machines, and to implement them, wherever possible, with efficient parallel code for multiprocessor machines that will become increasingly common in the future. We have found three kinds of situations where additional utility routines seem desirable:

1. Fortran 90 is a big language, with many high-level constructs — single statements that actually result in a lot of computing. We like this; it gives the language the potential for expressing algorithms very readably, getting them “out of the mud” of microscopic coding. In coding the 350+ Recipes for this volume, we kept a systematic watch for bits of microscopic coding that were repeated in many routines, and that seemed to be at a lower level of coding than that aspired to by good Fortran 90 style. Once these bits were identified, we pulled them out and substituted calls to new utility routines. These are the utilities that arguably ought to be new language intrinsics, equally useful for serial and parallel machines. (A prime example is swap.)

2. Fortran 90 contains many highly parallelizable language constructions. But, as we have seen in §22.5, it is also missing a few important constructions. Most parallel machines will provide these missing elements as machine-coded library subroutines. Some of our utility routines are provided simply as a standard interface to these common, but nonstandard, functionalities. Note that it is the nature of these routines that our specific implementation, in Appendix C1, will be serial, and therefore inefficient on parallel machines. If you have a parallel machine, you will need to recode these; this often involves no more than substituting a one-line library function call for the body of our implementation. Utilities in this category will likely become unnecessary over time, either as machine-dependent libraries converge to standard interfaces, or as the utilities get added to future Fortran versions. (Indeed,
some routines in this category will be unnecessary in Fortran 95, once it is available; see §23.7.)

3. Some tasks should just be done differently in serial, versus parallel, implementation. Linear recurrence relations are a good example (§22.2). These are trivially coded with a do-loop on serial machines, but require a fairly elaborate recursive construction for good parallelization. Rather than provide separate serial and parallel versions of the Numerical Recipes, we have chosen to pull out of the Recipes, and into utility routines, some identifiable tasks of this kind. These are cases where some recoding of our implementation in Appendix C1 might result in improved performance on your particular hardware. Unfortunately, it is not so simple as providing a single “serial implementation” and another single “parallel implementation,” because even the seemingly simple word “serial” hides, at the hardware level, a variety of different degrees of pipelining, wide instructions, and so on. Appendix C1 therefore provides only a single implementation, although with some adjustable parameters that you can customize (by experiment) to maximize performance on your hardware.

The above three cases are not really completely distinct, and it is therefore not possible to assign any single utility routine to exactly one situation. Instead, we organize the rest of this chapter as follows: first, an alphabetical list, with short summary, of all the new utility routines; next, a series of short sections, grouped by functionality, that contain the detailed descriptions of the routines.

Alphabetical Listing

The following list gives an abbreviated mnemonic for the type, rank, and/or shape of the returned values (as in §21.4), the routine’s calling sequence (optional arguments shown in italics), and a brief, often incomplete, description. The complete description of the routine is given in the later section shown in square brackets.

For each entry, the number shown in parentheses is the approximate number of distinct Recipes in this book that make use of that particular utility function, and is thus a rough guide to that utility’s importance. (There may be multiple invocations of the utility in each such Recipe.) Where this number is small or zero, it is usually because the utility routine is a member of a related family of routines whose total usage was deemed significant enough to include, and we did not want users to have to “guess” which family members were instantiated.

call array_copy(src,dest,n_copied,n_not_copied)
Copy one-dimensional array (whose size is not necessarily known). [23.1] (9)

[Arr] arth(first,increment,n)
Return an arithmetic progression as an array. [23.4] (55)

call assert(n1,n2,...,string)
Exit with error message if any logical arguments are false. [23.3] (50)

[Int] assert_eq(n1,n2,...,string)
Exit with error message if all integer arguments are not equal; otherwise return common value. [23.3] (133)

[argTS] cumprod(arr,seed)
Cumulative products of one-dimensional array, with optional seed value. [23.4] (3)

```fortran
subroutine cumsum(arr,seed)
    ! Cumulative sums of one-dimensional array, with optional seed value.
    ! [23.4] (9)
    call diagadd(mat,diag)
        ! Adds vector to diagonal of a matrix. [23.7] (4)
    call diagmult(mat,diag)
        ! Multiplies vector into diagonal of a matrix. [23.7] (2)
end subroutine cumsum
```

```fortran
subroutine geop(first,factor,n)
    ! Return a geometrical progression as an array. [23.4] (7)
end subroutine geop
```

```fortran
subroutine get_diag(mat)
    ! Gets diagonal of a matrix. [23.7] (2)
end subroutine get_diag
```

```fortran
subroutine ifirstloc(arr)
    ! Location of first true value in a logical array, returned as an integer.
    ! [23.7] (3)
end subroutine ifirstloc
```

```fortran
subroutine imaxloc(arr)
    ! Location of array maximum, returned as an integer. [23.7] (11)
end subroutine imaxloc
```

```fortran
subroutine iminloc(arr)
    ! Location of array minimum, returned as an integer. [23.7] (8)
end subroutine iminloc
```

```fortran
subroutine lower_triangle(j,k,extra)
    ! Returns a lower triangular logical mask. [23.7] (1)
end subroutine lower_triangle
```

```fortran
subroutine nrerror(string)
    ! Exit with error message. [23.3] (96)
end subroutine nrerror
```

```fortran
subroutine outerand(a,b)
    ! Returns the outer logical and of two vectors. [23.5] (1)
end subroutine outerand
```

```fortran
subroutine outerdiff(a,b)
    ! Returns the outer difference of two vectors. [23.5] (4)
end subroutine outerdiff
```

```fortran
subroutine outerdiv(a,b)
    ! Returns the outer quotient of two vectors. [23.5] (0)
end subroutine outerdiv
```

```fortran
subroutine outerprod(a,b)
    ! Returns the outer product of two vectors. [23.5] (14)
end subroutine outerprod
```

```fortran
subroutine outersum(a,b)
    ! Returns the outer sum of two vectors. [23.5] (0)
end subroutine outersum
```

```fortran
subroutine poly(x,coeffs,mask)
    ! Evaluate a polynomial \( P(x) \) for one or more values \( x \), with optional mask.
    ! [23.4] (15)
end subroutine poly
```

```fortran
subroutine poly_term(a,x)
    ! Returns partial cumulants of a polynomial, equivalent to synthetic
end subroutine poly_term
```
division. [23.4] (4)

call put_diagonal(diag, mat)
Sets diagonal of a matrix. [23.7] (0)

[Ptr]
reallocate(p,n,m,...)
Reallocate pointer to new size, preserving its contents. [23.1] (5)

call scatter_add(dest, source, dest_index)
Scatter-add source vector to specified components of destination vector. [23.6] (2)

call scatter_max(dest, source, dest_index)
Scatter-max source vector to specified components of destination vector. [23.6] (0)

call swap(a,b,mask)
Swap corresponding elements of a and b. [23.1] (24)

call unit_matrix(mat)
Sets matrix to be a unit matrix. [23.7] (6)

[Mat] upper_triangle(j,k,extra)
Returns an upper triangular logical mask. [23.7] (4)

[Real] vabs(v)
Length of a vector in \( L_2 \) norm. [23.8] (6)

[CArr] zroots_unity(n,nn)
Returns \( nn \) consecutive powers of the complex \( n \)th root of unity. [23.4] (4)

Comment on Relative Frequencies of Use

We find it interesting to compare our frequency of using the \texttt{nrutil} utility routines, with our most used language intrinsics (see §21.4). On this basis, the following routines are as useful to us as the top 10 language intrinsics: \texttt{arth}, \texttt{assert}, \texttt{assert_eq}, \texttt{outerprod}, \texttt{poly}, and \texttt{swap}. We strongly recommend that the X3J3 standards committee, as well as individual compiler library implementors, consider the inclusion of new language intrinsics (or library routines) that subsume the capabilities of at least these routines. In the next tier of importance, we would put some further cumulative operations (\texttt{geop}, \texttt{cumsum}), some other “outer” operations on vectors (e.g., \texttt{outerdiff}), basic operations on the diagonals of matrices (\texttt{get_diag}, \texttt{put_diag}, \texttt{diag_add}), and some means of access to an array of unknown size (\texttt{array_copy}).

23.1 Routines That Move Data

To describe our utility routines, we introduce two items of Fortran 90 pseudocode: We use the symbol \( T \) to denote some type and rank declaration (including...
scalar rank, i.e., zero); and when we append a colon to a type specification, as in INTEGER(I4B)(:), for example, we denote an array of the given type.

The routines swap, array_copy, and reallocate simply move data around in useful ways.

\* \* \*

**swap** (swaps corresponding elements)

User interface (or, "USE nrutil"):

```fortran
SUBROUTINE swap(a,b,mask)
  T, INTENT(INOUT) :: a,b
  LOGICAL(LGT), INTENT(IN), OPTIONAL :: mask
END SUBROUTINE swap
```

Applicable types and ranks:

\[ T \equiv \text{any type, any rank} \]

Types and ranks implemented (overloaded) in nrutil:

\[ T \equiv \text{INTEGER(I4B), REAL(SP), REAL(SP)(:), REAL(DP),} \]
\[ \text{COMPLEX(SPC), COMPLEX(SPC)(:), COMPLEX(SPC)(:,:),} \]
\[ \text{COMPLEX(DPC), COMPLEX(DPC)(:), COMPLEX(DPC)(:,:)} \]

Action:

Swaps the corresponding elements of a and b. If mask is present, performs the swap only where mask is true. (Following code is the unmasked case. For speed at run time, the masked case is implemented by overloading, not by testing for the optional argument.)

Reference implementation:

```fortran
T :: dum
  dum = a
  a = b
  b = dum
```

\* \* \*

**array_copy** (copy one-dimensional array)

User interface (or, "USE nrutil"):

```fortran
SUBROUTINE array_copy(src,dest,n_copied,n_not_copied)
  T, INTENT(IN) :: src
  T, INTENT(OUT) :: dest
  INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
END SUBROUTINE array_copy
```

Applicable types and ranks:

\[ T \equiv \text{any type, rank 1} \]

Types and ranks implemented (overloaded) in nrutil:

\[ T \equiv \text{INTEGER(I4B)(:), REAL(SP)(:), REAL(DP)(:)} \]

Action:

Copies to a destination array dest the one-dimensional array src, or as much of src as will fit in dest. Returns the number of components copied as n_copied, and the number of components not copied as n_not_copied.

The main use of this utility is where src is an expression that returns an array whose size is not known in advance, for example, the value returned by the pack intrinsic.
Reference implementation:

\[
\begin{align*}
n_{\text{copied}} &= \min (\text{size}(\text{src}), \text{size}(\text{dest})) \\
n_{\text{not} \text{copied}} &= \text{size}(\text{src}) - n_{\text{copied}} \\
\text{dest}(1:n_{\text{copied}}) &= \text{src}(1:n_{\text{copied}})
\end{align*}
\]

reallocate (reallocates a pointer, preserving contents)

User interface (or, “USE nrutil”):

\[
\begin{align*}
\text{FUNCTION reallocate} & (p, n[, m, ...]) \\
\text{T, POINTER} & :: p, \text{reallocate} \\
\text{INTEGER(I4B), INTENT(IN)} & :: n[, m, ...] \\
\text{END FUNCTION reallocate}
\end{align*}
\]

Applicable types and ranks:

\[ \text{T} \equiv \text{any type, rank 1 or greater} \]

Types and ranks implemented (overloaded) in nrutil:

\[ \text{T} \equiv \text{INTEGER(I4B)(:), INTEGER(I4B)(:,,:), REAL(SP)(:), REAL(SP)(:,:), CHARACTER(1)(:)} \]

Action:

Allocates storage for a new array with shape specified by the integer(s) \( n, m, \ldots \) (equal in number to the rank of pointer \( p \)). Then, copies the contents of \( p \)’s target (or as much as will fit) into the new storage. Then, deallocates \( p \) and returns a pointer to the new storage.

The typical use is \( p = \text{reallocate}(p, n[, m, \ldots]) \), which has the effect of changing the allocated size of \( p \) while preserving the contents.

The reference implementation, below, shows only the case of rank 1.

Reference implementation:

\[
\begin{align*}
\text{INTEGER(I4B)} & :: n\text{old}, \text{ierr} \\
\text{allocate} & (\text{reallocate}(n), \text{stat}=\text{ierr}) \\
\text{if (ierr /= 0) call &} \\
\text{nrerror} & ('\text{reallocate: problem in attempt to allocate memory}') \\
\text{if (.not. associated}(p) & \text{RETURN} \\
\text{nold} & = \text{size}(p) \\
\text{reallocate}(1:\text{min}(\text{nold}, n)) & = p(1:\text{min}(\text{nold}, n)) \\
\text{deallocate} & (p)
\end{align*}
\]

23.2 Routines Returning a Location

Fortran 90’s intrinsics \text{maxloc} and \text{minloc} return rank-one arrays. When, in the most frequent usage, their argument is a one-dimensional array, the answer comes back in the inconvenient form of an array containing a single component, which cannot be itself used in a subscript calculation. While there are workaround tricks (e.g., use of the \text{sum} intrinsic to convert the array to a scalar), it seems clearer to define routines \text{imaxloc} and \text{iminloc} that return integers directly.

The routine \text{ifirstloc} adds a related facility missing among the intrinsics: Return the first true location in a logical array.

\[
\begin{align*}
\end{align*}
\]
23.2 Routines Returning a Location

**imaxloc**  (location of array maximum as an integer)

*User interface (or, “USE nrutil”):*

```fortran
FUNCTION imaxloc(arr)
  T, INTENT(IN) :: arr
  INTEGER(I4B) :: imaxloc
END FUNCTION imaxloc
```

*Applicable types and ranks:*

- \( T \equiv \) any integer or real type, rank 1

*Types and ranks implemented (overloaded) in nrutil:*

- \( T \equiv \) INTEGER(I4B)(:), REAL(SP)(:)

*Action:*

For one-dimensional arrays, identical to the \texttt{maxloc} intrinsic, except returns its answer as an integer rather than as \texttt{maxloc}'s somewhat awkward rank-one array containing a single component.

*Reference implementation:*

```fortran
INTEGER(I4B), DIMENSION(1) :: imax
imax=maxloc(arr(:))
imaxloc=imax(1)
```

**iminloc**  (location of array minimum as an integer)

*User interface (or, “USE nrutil”):*

```fortran
FUNCTION iminloc(arr)
  T, INTENT(IN) :: arr
  INTEGER(I4B) :: iminloc
END FUNCTION iminloc
```

*Applicable types and ranks:*

- \( T \equiv \) any integer or real type, rank 1

*Types and ranks implemented (overloaded) in nrutil:*

- \( T \equiv \) REAL(SP)(:)

*Action:*

For one-dimensional arrays, identical to the \texttt{minloc} intrinsic, except returns its answer as an integer rather than as \texttt{minloc}'s somewhat awkward rank-one array containing a single component.

*Reference implementation:*

```fortran
INTEGER(I4B), DIMENSION(1) :: imin
imin=minloc(arr(:))
iminloc=imin(1)
```

**ifirstloc**  (returns location of first “true” in a logical vector)

*User interface (or, “USE nrutil”):*

```fortran
FUNCTION ifirstloc(mask)
  T, INTENT(IN) :: mask
  INTEGER(I4B) :: ifirstloc
END FUNCTION ifirstloc
```
23.3 Argument Checking and Error Handling

It is good programming practice for a routine to check the assumptions ("assertions") that it makes about the sizes of input arrays, allowed range of numerical arguments, and so forth. The routines assert and assert_eq are meant for this kind of use. The routine nrerror is our default error reporting routine.

* * *

assert  (exit with error message if any assertion is false)

User interface (or, "USE nrutil"):  
SUBROUTINE assert(n1,n2,...,string)
CHARACTER(LEN=*) , INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1,n2,...
END SUBROUTINE assert

Action:  
Embedding program dies gracefully with an error message if any of the logical arguments are false. Typical use is with logical expressions as the actual arguments. nrutil implements and overloads forms with 1, 2, 3, and 4 logical arguments, plus a form with a vector logical argument,

LOGICAL, DIMENSION(:), INTENT(IN) :: n
that is checked by the all(n) intrinsic.
Reference implementation:

```fortran
if (.not. (n1.and.n2.and...)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', string
    STOP 'program terminated by assert'
end if
```

```
* * *
```

**assert_eq** (exit with error message if integer arguments not all equal)

**User interface (or, "USE nrutil")**:

```fortran
FUNCTION assert_eq(n1,n2,n3,...,string)
    CHARACTER(LEN=*) :: string
    INTEGER :: assert_eq
END FUNCTION assert_eq
```

**Action:**

Embedding program dies gracefully with an error message if any of the integer arguments are not equal to the first. Otherwise, return the value of the first argument. Typical use is for enforcing equality on the sizes of arrays passed to a subprogram. *nrutil* implements and overloads forms with 1, 2, 3, and 4 integer arguments, plus a form with a vector integer argument,

```fortran
INTEGER, DIMENSION(:), INTENT(IN) :: n
```

that is checked by the conditional if (all(nn(2::)==nn(1))).

**Reference implementation**:

```fortran
if (n1==n2.and.n2==n3.and...) then
    assert_eq=n1
else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', string
    STOP 'program terminated by assert_eq'
end if
```

```
* * *
```

**nrerror** (report error message and stop)

**User interface (or, "USE nrutil")**:

```fortran
SUBROUTINE nrerror(string)
    CHARACTER(LEN=*) :: string
END SUBROUTINE nrerror
```

**Action:**

This is the minimal error handler used in this book. In applications of any complexity, it is intended only as a placeholder for a user’s more complicated error handling strategy.

**Reference implementation**:

```fortran
write (*,*) 'nrerror: ',string
STOP 'program terminated by nrerror'
```
23.4 Routines for Polynomials and Recurrences

Apart from programming convenience, these routines are designed to allow for nontrivial parallel implementations, as discussed in §22.2 and §22.3.

⋆⋆⋆

arth (returns arithmetic progression as an array)

User interface (or, ‘USE nrutil’):

FUNCTION arth(first,increment,n)
T, INTENT(IN) :: first,increment
INTEGER(I4B), INTENT(IN) :: n
T, DIMENSION(n) :: arth
END FUNCTION arth

Applicable types and ranks:

T ≡ any numerical type, any rank

Types and ranks implemented (overloaded) in nrutil:

T ≡ INTEGER(I4B), REAL(SP), REAL(DP)

Action:

Returns an array of length n containing an arithmetic progression whose first value is first and whose increment is increment. If first and increment have rank greater than zero, returns an array of one larger rank, with the last subscript having size n and indexing the progressions. Note that the following reference implementation (for the scalar case) is definitional only, and neither parallelized nor optimized for roundoff error. See §22.2 and Appendix C1 for implementation by subvector scaling.

Reference implementation:

INTEGER(I4B) :: k
if (n > 0) arth(1)=first
   do k=2,n
       arth(k)=arth(k-1)+increment
   end do

⋆⋆⋆

geop (returns geometric progression as an array)

User interface (or, “USE nrutil”):

FUNCTION geop(first,factor,n)
T, INTENT(IN) :: first,factor
INTEGER(I4B), INTENT(IN) :: n
T, DIMENSION(n) :: geop
END FUNCTION geop

Applicable types and ranks:

T ≡ any numerical type, any rank

Types and ranks implemented (overloaded) in nrutil:

T ≡ INTEGER(I4B), REAL(SP), REAL(DP), REAL(DP)(::), COMPLEX(SPC)
Action:
Returns an array of length \( n \) containing a geometric progression whose first value is \( \text{first} \) and whose multiplier is \( \text{factor} \). If \( \text{first} \) and \( \text{factor} \) have rank greater than zero, returns an array of one larger rank, with the last subscript having size \( n \) and indexing the progression. Note that the following reference implementation (for the scalar case) is definitional only, and neither parallelized nor optimized for roundoff error. See §22.2 and Appendix C1 for implementation by subvector scaling.

Reference implementation:

```
INTEGER(I4B) :: k
if (n > 0) geop(1)=first
   do k=2,n
      geop(k)=geop(k-1)*factor
   end do
```

\( \star \star \star \)

\textbf{cumsum} (cumulative sum on an array, with optional additive seed)

User interface (or, “USE nrutil”):

```
FUNCTION cumsum(arr,seed)
   T, DIMENSION(:), INTENT(IN) :: arr
   T, OPTIONAL, INTENT(IN) :: seed
   T, DIMENSION(size(arr)), INTENT(OUT) :: cumsum
END FUNCTION cumsum
```

Applicable types and ranks:

\( T \equiv \text{any numerical type} \)

Types and ranks implemented (overloaded) in \textit{nrutil}:

\( T \equiv \text{INTEGER(I4B), REAL(SP)} \)

Action:
Given the rank 1 array \( \text{arr} \) of type \( T \), returns an array of identical type and size containing the cumulative sums of \( \text{arr} \). If the optional argument \( \text{seed} \) is present, it is added to the first component (and therefore, by cumulation, all components) of the result. See §22.2 for parallelization ideas.

Reference implementation:

```
INTEGER(I4B) :: n,j
T :: sd
n=size(arr)
if (n == 0) return
sd=0.0
if (present(seed)) sd=seed
   cumsum(1)=arr(1)+sd
do j=2,n
   cumsum(j)=cumsum(j-1)+arr(j)
end do
```

\( \star \star \star \)

\textbf{cumprod} (cumulative prod on an array, with optional multiplicative seed)

User interface (or, “USE nrutil”):

```
FUNCTION cumprod(arr,seed)
   T, DIMENSION(:), INTENT(IN) :: arr
   T, OPTIONAL, INTENT(IN) :: seed
   T, DIMENSION(size(arr)), INTENT(OUT) :: cumprod
END FUNCTION cumprod
```
Applicable types and ranks:

T ≡ any numerical type

Types and ranks implemented (overloaded) in nrutil:

T ≡ REAL(SP)

Action:
Given the rank 1 array arr of type T, returns an array of identical type and size containing the cumulative products of arr. If the optional argument seed is present, it is multiplied into the first component (and therefore, by cumulation, into all components) of the result. See §22.2 for parallelization ideas.

Reference implementation:

```fortran
INTEGER(I4B) :: n,j
T :: sd
n=size(arr)
if (n == 0) return
sd=1.0
if (present(seed)) sd=seed
cumprod(1)=arr(1)*sd
do j=2,n
   cumprod(j)=cumprod(j-1)*arr(j)
end do
```

poly (polynomial evaluation)

User interface (or, “USE nrutil”):

```fortran
FUNCTION poly(x,coeffs,mask)
T, T, DIMENSION(:,...), INTENT(IN) :: x
T, DIMENSION(:), INTENT(IN) :: coeffs
LOGICAL(LGT), DIMENSION(:,...), OPTIONAL, INTENT(IN) :: mask
T :: poly
END FUNCTION poly
```

Applicable types and ranks:

T ≡ any numerical type (x may be scalar or have any rank; x and coeffs may have different numerical types)

Types and ranks implemented (overloaded) in nrutil:

T ≡ various combinations of REAL(SP), REAL(SP)(:), REAL(DP), REAL(DP)(,:), COMPLEX(SPC) (see Appendix C1 for details)

Action:

Returns a scalar value or array with the same type and shape as x, containing the result of evaluating the polynomial with one-dimensional coefficient vector coeffs on each component of x. The optional argument mask, if present, has the same shape as x, and suppresses evaluation of the polynomial where its components are .false.. The following reference code shows the case where mask is not present. (The other case can be included by overloading.)
23.4 Routines for Polynomials and Recurrences

Reference implementation:

```fortran
INTEGER(I4B) :: i, n
n = size(coeffs)
if (n <= 0) then
  poly = 0.0
else
  poly = coeffs(n)
  do i = n-1, 1, -1
    poly = x * poly + coeffs(i)
  end do
end if
```

```
⋆⋆⋆
```

**poly_term** (partial cumulants of a polynomial)

*User interface (or, “USE nrutil”):*

```fortran
FUNCTION poly_term(a,x)
  T, DIMENSION(:), INTENT(IN) :: a
  T, INTENT(IN) :: x
  T, DIMENSION(size(a)) :: poly_term
END FUNCTION poly_term
```

**Applicable types and ranks:**

- \( T \equiv \text{any numerical type} \)

**Types and ranks implemented (overloaded) in nrutil:**

- \( T \equiv \text{REAL(SP), COMPLEX(SPC)} \)

*Action:*

Returns an array of type and size the same as the one-dimensional array \( a \), containing the partial cumulants of the polynomial with coefficients \( a \) (arranged from highest-order to lowest-order coefficients, n.b.) evaluated at \( x \). This is equivalent to synthetic division, and can be parallelized. See §22.3. Note that the order of arguments is reversed in \( \text{poly} \) and \( \text{poly}_\text{term} \) — each routine returns a value with the size and shape of the first argument, the usual Fortran 90 convention.

Reference implementation:

```fortran
INTEGER(I4B) :: n, j
n = size(a)
if (n <= 0) return
poly_term(1) = a(1)
  do j = 2, n
    poly_term(j) = a(j) * x * poly_term(j-1)
  end do
```

```
⋆⋆⋆
```

**zroots_unity** (returns powers of complex \( n \)th root of unity)

*User interface (or, “USE nrutil”):*

```fortran
FUNCTION zroots_unity(n,nn)
  INTEGER(I4B), INTENT(IN) :: n,nn
  COMPLEX(SPC), DIMENSION(nn) :: zroots_unity
END FUNCTION zroots_unity
```
Action:

Returns a complex array containing \( nn \) consecutive powers of the \( n \)th complex root of unity. Note that the following reference implementation is definitional only, and neither parallelized nor optimized for roundoff error. See Appendix C1 for implementation by subvector scaling.

Reference implementation:

```
INTEGER(I4B) :: k
REAL(SP) :: theta
if (nn==0) return
zroots_unity(1)=1.0
if (nn==1) return
theta=TWOPI/n
zroots_unity(2)=cmplx(cos(theta),sin(theta))
do k=3,nn
  zroots_unity(k)=zroots_unity(k-1)*zroots_unity(2)
end do
```

23.5 Routines for Outer Operations on Vectors

Outer operations on vectors take two vectors as input, and return a matrix as output. One dimension of the matrix is the size of the first vector, the other is the size of the second vector. Our convention is always the standard one,

\[
result(i,j) = first\_operand(i) \times op \times second\_operand(j)
\]

where \( (op) \) is any of addition, subtraction, multiplication, division, and logical and. The reason for coding these as utility routines is that Fortran 90’s native construction, with two spreads (cf. §22.1), is difficult to read and thus prone to programmer errors.

```
*   *   *
outerprod (outer product)

User interface (or, “USE nrutil”):

```
FUNCTION outerprod(a,b)
  T, DIMENSION(:), INTENT(IN) :: a,b
  T, DIMENSION(size(a),size(b)) :: outerprod
END FUNCTION outerprod
```

Applicable types and ranks:

\( T \equiv \) any numerical type

Types and ranks implemented (overloaded) in nrutil:

\( T \equiv \) REAL(SP), REAL(DP)

Action:

Returns a matrix that is the outer product of two vectors.

Reference implementation:

```
outerprod = spread(a,dim=2,ncopies=size(b)) * k
  spread(b,dim=1,ncopies=size(a))
```

```
*   *   *
```
outerdiv  (outer quotient)

User interface (or, “USE nrutil”):
FUNCTION outerdiv(a,b)
  T, DIMENSION(:,), INTENT(IN) :: a,b
  T, DIMENSION(size(a),size(b)) :: outerdiv
END FUNCTION outerdiv

Applicable types and ranks:
  T ≡ any numerical type

Types and ranks implemented (overloaded) in nrutil:
  T ≡ REAL(SP)

Action:
Returns a matrix that is the outer quotient of two vectors.

Reference implementation:
outerdiv = spread(a,dim=2,ncopies=size(b)) / &
          spread(b,dim=1,ncopies=size(a))

outersum  (outer sum)

User interface (or, “USE nrutil”):
FUNCTION outersum(a,b)
  T, DIMENSION(:,), INTENT(IN) :: a,b
  T, DIMENSION(size(a),size(b)) :: outersum
END FUNCTION outersum

Applicable types and ranks:
  T ≡ any numerical type

Types and ranks implemented (overloaded) in nrutil:
  T ≡ REAL(SP)

Action:
Returns a matrix that is the outer sum of two vectors.

Reference implementation:
outersum = spread(a,dim=2,ncopies=size(b)) + &
           spread(b,dim=1,ncopies=size(a))

outerdiff  (outer difference)

User interface (or, “USE nrutil”):
FUNCTION outerdiff(a,b)
  T, DIMENSION(:,), INTENT(IN) :: a,b
  T, DIMENSION(size(a),size(b)) :: outerdiff
END FUNCTION outerdiff

Applicable types and ranks:
  T ≡ any numerical type

Types and ranks implemented (overloaded) in nrutil:
  T ≡ INTEGER(I4B), REAL(SP), REAL(DP)

Action:
Returns a matrix that is the outer difference of two vectors.
Reference implementation:
outerdiff = spread(a,dim=2,ncopies=size(b)) - &
          spread(b,dim=1,ncopies=size(a))

⋆⋆⋆

outerand  (outer logical and)

User interface (or, “USE nrutil”):
FUNCTION outerand(a,b)
  LOGICAL(LGT), DIMENSION(:,), INTENT(IN) :: a,b
  LOGICAL(LGT), DIMENSION(size(a),size(b)) :: outerand
END FUNCTION outerand

Applicable types and ranks:
T ≡ any logical type

Types and ranks implemented (overloaded) in nrutil:
T ≡ LOGICAL(LGT)

Action:
Returns a matrix that is the outer logical and of two vectors.

Reference implementation:
outerand = spread(a,dim=2,ncopies=size(b)) .and. &
          spread(b,dim=1,ncopies=size(a))

23.6 Routines for Scatter with Combine

These are common parallel functions that Fortran 90 simply doesn’t provide
a means for implementing. If you have a parallel machine, you should substitute
library routines specific to your hardware.

⋆⋆⋆

scatter_add  (scatter-add source to specified components of destination)

User interface (or, “USE nrutil”):
SUBROUTINE scatter_add(dest,source,dest_index)
  T, DIMENSION(:,), INTENT(OUT) :: dest
  T, DIMENSION(:,), INTENT(IN) :: source
  INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: dest_index
END SUBROUTINE scatter_add

Applicable types and ranks:
T ≡ any numerical type

Types and ranks implemented (overloaded) in nrutil:
T ≡ REAL(SP), REAL(DP)
Action:
Adds each component of the array source into a component of dest
specified by the index array dest_index. (The user will usually have
zeroed dest before the call to this routine.) Note that dest_index
has the size of source, but must contain values in the range from 1 to
size(dest), inclusive. Out-of-range values are ignored. There is no
parallel implementation of this routine accessible from Fortran 90; most
parallel machines supply an implementation as a library routine.

Reference implementation:

```
INTEGER(I4B) :: m,n,j,i
n=assert_eq(size(source),size(dest_index),'scatter_add')
m=size(dest)
do j=1,n
  i=dest_index(j)
  if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
```

`scatter_add` (scatter-add source into specified components of destination)

User interface (or "USE nrutil"):

```
SUBROUTINE scatter_add(dest,source,dest_index)
  T, DIMENSION(:), INTENT(OUT) :: dest
  T, DIMENSION(:), INTENT(IN) :: source
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
END SUBROUTINE scatter_add
```

Applicable types and ranks:

- `T` ≡ any integer or real type

Types and ranks implemented (overloaded) in nrutil:
- `T` ≡ REAL(SP), REAL(DP)

Action:
Takes the add operation between each component of the array source
and a component of dest specified by the index array dest_index, replacing
that component of dest with the value obtained ("adding into" operation).
(The user will often want to fill the array dest with the value −huge before
the call to this routine.) Note that dest_index has the size of source,
but must contain values in the range from 1 to size(dest), inclusive.
Out-of-range values are ignored. There is no parallel implementation of
this routine accessible from Fortran 90; most parallel machines supply an
implementation as a library routine.

Reference implementation:

```
INTEGER(I4B) :: m,n,j,i
n=assert_eq(size(source),size(dest_index),'scatter_max')
m=size(dest)
do j=1,n
  i=dest_index(j)
  if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
```

`scatter_max` (scatter-max source to specified components of destination)

User interface (or "USE nrutil"):

```
SUBROUTINE scatter_max(dest,source,dest_index)
  T, DIMENSION(:), INTENT(OUT) :: dest
  T, DIMENSION(:), INTENT(IN) :: source
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
END SUBROUTINE scatter_max
```

Applicable types and ranks:

- `T` ≡ any integer or real type

Types and ranks implemented (overloaded) in nrutil:
- `T` ≡ REAL(SP), REAL(DP)

Action:
Takes the max operation between each component of the array source
and a component of dest specified by the index array dest_index, replacing
that component of dest with the value obtained ("maxing into" operation).
(The user will often want to fill the array dest with the value −huge before
the call to this routine.) Note that dest_index has the size of source,
but must contain values in the range from 1 to size(dest), inclusive.
Out-of-range values are ignored. There is no parallel implementation of
this routine accessible from Fortran 90; most parallel machines supply an
implementation as a library routine.
23.7 Routines for Skew Operations on Matrices

These are also missing parallel capabilities in Fortran 90. In Appendix C1 they are coded serially, with one or more do-loops.

⋆⋆⋆

diagadd (adds vector to diagonal of a matrix)

User interface (or, “USE nrutil”):
SUBROUTINE diagadd(mat,diag)
  T, DIMENSION(:,,:), INTENT(INOUT) :: mat
  T, DIMENSION(:,), INTENT(IN) :: diag
END SUBROUTINE diagadd

Applicable types and ranks:
  T ≡ any numerical type

Types and ranks implemented (overloaded) in nrutil:
  T ≡ REAL(SP)

Action:
The argument diag, either a scalar or else a vector whose size must be the smaller of the two dimensions of matrix mat, is added to the diagonal of the matrix mat. The following shows an implementation where diag is a vector; the scalar case can be overloaded (see Appendix C1).

Reference implementation:

INTEGER(I4B) :: j,n
n = assert_eq(size(diag),min(size(mat,1),size(mat,2)),'diagadd')
do j=1,n
  mat(j,j)=mat(j,j)+diag(j)
end do

⋆⋆⋆

diagmult (multiplies vector into diagonal of a matrix)

User interface (or, “USE nrutil”):
SUBROUTINE diagmult(mat,diag)
  T, DIMENSION(:,,:), INTENT(INOUT) :: mat
  T, DIMENSION(:,), INTENT(IN) :: diag
END SUBROUTINE diagmult

Applicable types and ranks:
  T ≡ any numerical type

Types and ranks implemented (overloaded) in nrutil:
  T ≡ REAL(SP)

Action:
The argument diag, either a scalar or else a vector whose size must be the smaller of the two dimensions of matrix mat, is multiplied onto the diagonal of the matrix mat. The following shows an implementation where diag is a vector; the scalar case can be overloaded (see Appendix C1).
23.7 Routines for Skew Operations on Matrices

Reference implementation:

```fortran
INTEGER(I4B) :: j,n
n = assert_eq(size(diag),min(size(mat,1),size(mat,2)),'diagmult')
do j=1,n
   mat(j,j)=mat(j,j)*diag(j)
end do
```

get_diag (gets diagonal of matrix)

User interface (or, “USE nrutil”):

```fortran
FUNCTION get_diag(mat)
   T, DIMENSION(:,:), INTENT(IN) :: mat
   T, DIMENSION(min(size(mat,1),size(mat,2))) :: get_diag
END FUNCTION get_diag
```

Applicable types and ranks:

- \( T \equiv \text{any type} \)

Types and ranks implemented (overloaded) in nrutil:

- \( T \equiv \text{REAL(SP), REAL(DP)} \)

Action:

Returns a vector containing the diagonal values of the matrix mat.

Reference implementation:

```fortran
INTEGER(I4B) :: j
do j=1,min(size(mat,1),size(mat,2))
   get_diag(j)=mat(j,j)
end do
```

put_diag (sets the diagonal elements of a matrix)

User interface (or, “USE nrutil”):

```fortran
SUBROUTINE put_diag(diag,mat)
   T, DIMENSION(:), INTENT(IN) :: diag
   T, DIMENSION(:,:), INTENT(INOUT) :: mat
END SUBROUTINE put_diag
```

Applicable types and ranks:

- \( T \equiv \text{any type} \)

Types and ranks implemented (overloaded) in nrutil:

- \( T \equiv \text{REAL(SP)} \)

Action:

Sets the diagonal of matrix mat equal to the argument diag, either a scalar or else a vector whose size must be smaller than or equal to the smaller of the two dimensions of matrix mat. The following shows an implementation where diag is a vector; the scalar case can be overloaded (see Appendix C1).

Reference implementation:

```fortran
INTEGER(I4B) :: j,n
n=assert_eq(size(diag),min(size(mat,1),size(mat,2)),'put_diag')
do j=1,n
   mat(j,j)=diag(j)
end do
```
unit_matrix  (returns a unit matrix)

User interface (or, “USE nrutil”):

```
SUBROUTINE unit_matrix(mat)
   T, DIMENSION(:,,:), INTENT(OUT) :: mat
END SUBROUTINE unit_matrix
```

Applicable types and ranks:

\( T \equiv \text{any numerical type} \)

Types and ranks implemented (overloaded) in nrutil:

\( T \equiv \text{REAL(SP)} \)

Action:

Sets the diagonal components of \( \text{mat} \) to unity, all other components to zero.

When \( \text{mat} \) is square, this will be the unit matrix; otherwise, a unit matrix
with appended rows or columns of zeros.

Reference implementation:

```
INTEGER(I4B) :: i,n
n=min(size(mat,1),size(mat,2))
mat(:,:,)=0.0
do i=1,n
   mat(i,i)=1.0
end do
```

upper_triangle  (returns an upper triangular mask)

User interface (or, “USE nrutil”):

```
FUNCTION upper_triangle(j,k,extra)
   INTEGER(I4B), INTENT(IN) :: j,k
   INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
   LOGICAL(LGT), DIMENSION(j,k) :: upper_triangle
END FUNCTION upper_triangle
```

Action:

When the optional argument \( \text{extra} \) is zero or absent, returns a logical mask of
shape \((j,k)\) whose values are true above and to the right of the diagonal, false
elsewhere (including on the diagonal). When \( \text{extra} \) is present and positive,
a corresponding number of additional (sub-)diagonals are returned as true.
\((\text{extra} = 1 \text{ makes the main diagonal return true.}) \) When \( \text{extra} \) is present
and negative, it suppresses a corresponding number of superdiagonals.

Reference implementation:

```
INTEGER(I4B) :: n,jj,kk
n=0
if (present(extra)) n=extra
do jj=1,j
   do kk=1,k
      upper_triangle(jj,kk)= (jj-kk < n)
   end do
end do
```

lower_triangle  (returns a lower triangular mask)

User interface (or, “USE nrutil”):

```fortran
FUNCTION lower_triangle(j,k,extra)
INTEGER(I4B), INTENT(IN) :: j,k
INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
LOGICAL(LGT), DIMENSION(j,k) :: lower_triangle
END FUNCTION lower_triangle
```

Action:
When the optional argument extra is zero or absent, returns a logical mask of shape \((j,k)\) whose values are true below and to the left of the diagonal, false elsewhere (including on the diagonal). When extra is present and positive, a corresponding number of additional (super-)diagonals are returned as true. \((\text{extra} = 1 \text{ makes the main diagonal return true.})\) When extra is present and negative, it suppresses a corresponding number of subdiagonals.

Reference implementation:
```fortran
INTEGER(I4B) :: n,jj,kk
n=0
if (present(extra)) n=extra
do jj=1,j
  do kk=1,k
    lower_triangle(jj,kk)= (kk-jj < n)
  end do
end do
```

Fortran 95’s forall construction will make the parallel implementation of all our skew operations utilities extremely simple. For example, the do-loop in diagadd will collapse to

```fortran
forall (j=1:n) mat(j,j)=mat(j,j)+diag(j)
```

In fact, this implementation is so simple as to raise the question of whether a separate utility like diagadd will be needed at all. There are valid arguments on both sides of this question: The “con” argument, against a routine like diagadd, is that it is just another reserved name that you have to remember (if you want to use it). The “pro” argument is that a separate routine avoids the “index pollution” (the opposite disease from “index loss” discussed in §22.1) of introducing a superfluous variable \(j\), and that a separate utility allows for additional error checking on the sizes and compatibility of its arguments. We expect that different programmers will have differing tastes.

The argument for keeping a routine like upper_triangle or lower_triangle, once Fortran 95’s masked forall constructions become available, is less persuasive. We recommend that you consider these two routines as placeholders for “remember to recode this in Fortran 95, someday.”

23.8 Other Routine(s)

You might argue that we don’t really need a routine for the idiom

```fortran
sqrt(dot_product(v,v))
```
You might be right. The ability to overload the complex case, with its additional complex conjugate, is an argument in its favor, however.

* * *

\textbf{vabs} \hspace{1em} (L_2 \text{ norm of a vector})

\textit{User interface (or, “USE nrutil”):}

\begin{verbatim}
FUNCTION vabs(v)
    T, DIMENSION(:), INTENT(IN) :: v
    T :: vabs
END FUNCTION vabs
\end{verbatim}

\textbf{Applicable types and ranks:}

\begin{itemize}
    \item \( T \equiv \text{any real or complex type} \)
\end{itemize}

\textbf{Types and ranks implemented (overloaded) in nrutil:}

\begin{itemize}
    \item \( T \equiv \text{REAL(SP)} \)
\end{itemize}

\textbf{Action:}

Returns the length of a vector \( v \) in \( L_2 \) norm, that is, the square root of the sum of the squares of the components. (For complex types, the \texttt{dot_product} should be between the vector and its complex conjugate.)

\textbf{Reference implementation:}

\begin{verbatim}
vabs=sqrt(dot_product(v,v))
\end{verbatim}
Fortran 90 Code Chapters B1–B20

Fortran 90 versions of all the Numerical Recipes routines appear in the following Chapters B1 through B20, numbered in correspondence with Chapters 1 through 20 in Volume 1. Within each chapter, the routines appear in the same order as in Volume 1, but not broken out separately by section number within Volume 1’s chapters.

There are commentaries accompanying many of the routines, generally following the printed listing of the routine to which they apply. These are of two kinds: issues related to parallelizing the algorithm in question, and issues related to the Fortran 90 implementation. To distinguish between these two, rather different, kinds of discussions, we use the two icons,

the left icon (above) indicating a “parallel note,” and the right icon denoting a “Fortran 90 tip.” Specific code segments of the routine that are discussed in these commentaries are singled out by reproducing some of the code as an “index line” next to the icon, or at the beginning of subsequent paragraphs if there are several items that are commented on.

\[
d = \text{merge}(\text{FPMIN}, d, \text{abs}(d) < \text{FPMIN})
\]

This would be the start of a discussion of code that begins at the line in the listing containing the indicated code fragment.

\* \* \*

A row of stars, like the above, is used between unrelated routines, or at the beginning and end of related groups of routines.

Some chapters contain discussions that are more general than commentary on individual routines, but that were deemed too specific for inclusion in Chapters 21 through 23. Here are some highlights of this additional material:

- Approximations to roots of orthogonal polynomials for parallel computation of Gaussian quadrature formulas (Chapter B4)
- Difficulty of, and tricks for, parallel calculation of special function values in a SIMD model of computation (Chapter B6)
- Parallel random number generation (Chapter B7)
- Fortran 90 tricks for dealing with ties in sorted arrays, counting things in boxes, etc. (Chapter B14)
- Use of recursion in implementing multigrid elliptic PDE solvers (Chapter B19)
Chapter B1. Preliminaries

SUBROUTINE flmoon(n, nph, jd, frac)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n, nph
INTEGER(I4B), INTENT(OUT) :: jd
REAL(SP), INTENT(OUT) :: frac

Our programs begin with an introductory comment summarizing their purpose and explaining their calling sequence. This routine calculates the phases of the moon. Given an integer \(n\) and a code \(nph\) for the phase desired (\(nph = 0\) for new moon, 1 for first quarter, 2 for full, 3 for last quarter), the routine returns the Julian Day Number \(jd\), and the fractional part of a day \(frac\) to be added to it, of the \(n\)th such phase since January, 1900. Greenwich Mean Time is assumed.

REAL(SP), PARAMETER :: RAD=PI/180.0_sp
INTEGER(I4B) :: i
REAL(SP) :: am, as, c, t, t2, xtra

\\[ c=n+nph/4.0_sp \]

This is how we comment an individual line.

\\[ t=c/1236.85_sp \]
\\[ t2=t**2 \]
\\[ as=359.2242_sp+29.105356_sp*c \]

You aren't really intended to understand this algorithm, but it does work!

\\[ am=306.0253_sp+385.816918_sp*c+0.010730_sp*t2 \]

Greenwich Mean Time is assumed.

REAL(SP), PARAMETER :: RAD=PI/180.0_sp
INTEGER(I4B) :: i
REAL(SP) :: am, as, c, t, t2, xtra

\\[ xtra=xtra+(0.1734_sp-3.93e-4_sp*t)*sin(RAD*as)-0.4068_sp*sin(RAD*am) \]

This is how we will indicate error conditions.

END SUBROUTINE flmoon

Fortran 90 includes a \texttt{case} construction that executes at most one of several blocks of code, depending on the value of an integer, logical, or character expression. Ideally, the \texttt{case} construction will execute more efficiently than a long sequence of cascaded \texttt{if...else if...else if...} constructions. C programmers should note that the Fortran 90 construction, perhaps mercifully, does not have C’s “drop-through” feature.

merge(xtra, xtra-1.0_sp, xtra >= 0.0) The \texttt{merge} construction in Fortran 90, while intended primarily for use with vector arguments, is also a convenient way of generating conditional scalar expressions, that is, expressions with one value, or another, depending on the result of a logical test.
When the arguments of a merge are vectors, parallelization by the compiler is straightforward as an array parallel operation (see p. 964). Less obvious is how the scalar case, as above, is handled. For small-scale parallel (SSP) machines, the natural gain is via speculative evaluation of both of the first two arguments simultaneously with evaluation of the test.

A good compiler should not penalize a scalar machine for use of either the scalar or vector merge construction. The Fortran 90 standard states that “it is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely each operand, if the value of the expression can be determined otherwise.” Therefore, for each test on a scalar machine, only one or the other of the first two argument components need be evaluated.

```
FUNCTION julday(mm,id,iyyy)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: mm,id,iyyy
INTEGER(I4B) :: julday

In this routine julday returns the Julian Day Number that begins at noon of the calendar date specified by month mm, day id, and year iyyy, all integer variables. Positive year signifies A.D.; negative, B.C. Remember that the year after 1 B.C. was 1 A.D.

INTEGER(I4B), PARAMETER :: IGREG=15+31*(10+12*1582)  Gregorian Calendar adopted Oct. 15, 1582.

jy=iyyy
if (jy == 0) call nrerror(’julday: there is no year zero’)
if (jy < 0) jy=jy+1
if (mm > 2) then
  jm=mm+1
else
  jm=mm+13
end if
julday=floor(365.25_sp*jy)+floor(30.6001_sp*jm)+id+1720995
if (id+31*(mm+12*iyyy) >= IGREG) then
  Test whether to change to Gregorian Calendar.
  ja=floor(0.01_sp*jy)
  julday=julday+2-ja+floor(0.25_sp*ja)
end if

END FUNCTION julday
```

```
PROGRAM badluk
USE nrtype
USE nr, ONLY : flmoon,julday
IMPLICIT NONE
INTEGER(I4B) :: ic,icon,idwk,ifrac,im,iyyy,jd,jday,n
INTEGER(I4B) :: iybeg=1900,iyend=2000
The range of dates to be searched.
REAL(SP) :: frac
REAL(SP), PARAMETER :: TIMZON=-5.0_sp/24.0_sp
Time zone -5 is Eastern Standard Time.
write (*,’(1x,a,i5,a,i5)’) ’Full moons on Friday the 13th from’,&
  iybeg,’ to’,iyend
do iyyy=iybeg,iyend
  Loop over each year,
do im=1,12
    Is the 13th a Friday?
    jday=julday(im,13,iyyy)
    idwk=mod(jday+1,7)
  end do
end if
```

⋆⋆⋆
if (idwk == 5) then
    n=12.37_sp*(iyyy-1900+(im-0.5_sp)/12.0_sp)
    This value \( n \) is a first approximation to how many full moons have occurred since 1900. We will feed it into the phase routine and adjust it up or down until we determine that our desired 13th was or was not a full moon. The variable \( \text{icon} \) signals the direction of adjustment.
    icon=0
    do
        call flmoon(n,2,jd,frac) Get date of full moon \( n \).
        ifrac=nint(24.0_sp*(frac+TIMZON)) Convert to hours in correct time
            if (ifrac < 0) then
                jd=jd-1 Convert from Julian Days beginning at noon
                    ifrac=ifrac+24 to civil days beginning at midnight.
            end if
        if (ifrac > 12) then
            jd=jd+1
            ifrac=ifrac-12
        else
            ifrac=ifrac+12
        end if
        if (jd == jday) then
            Did we hit our target day?
            write (*,'(/1x,i2,a,i2,a,i4)') im,'/',13,'/',iyyy
            write (*,'(1x,a,i2,a)') 'Full moon ',ifrac,
                ' hrs after midnight (EST).'
                Don’t worry if you are unfamiliar with FORTRAN’s esoteric input/output
                statements; very few programs in this book do any input/output.
                exit Part of the break-structure, case of a match.
        else
            Didn’t hit it.
            ic=isign(1,jday-jd)
            if (ic == -icon) exit Another break, case of no match.
            icon=ic
            n=n+ic
        end if
    end do
end if
end do
end do
END PROGRAM badluk

\texttt{f90}\texttt{...IGREG=15+31*(10+12*1582) (in julday). ...TIMZON=-5.0_sp/24.0_sp (in badluk)} These are two examples of initialization expressions for “named constants” (that is, \texttt{PARAMETER}s). Because the initialization expressions will generally be evaluated at compile time, Fortran 90 puts some restrictions on what kinds of intrinsic functions they can contain. Although the evaluation of a real expression like \texttt{-5.0_sp/24.0_sp} \texttt{ought} to give identical results at compile time and at execution time, all the way down to the least significant bit, in our opinion the conservative programmer shouldn’t count on strict identity at the level of floating-point roundoff error. (In the special case of \texttt{cross}-compilers, such roundoff-level discrepancies between compile time and run time are almost inevitable.)

* * *
SUBROUTINE caldat(julian,mm,id,iyyy)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: julian
INTEGER(I4B), INTENT(OUT) :: mm,id,iyyy
Inverse of the function julyday given above. Here julian is input as a Julian Day Number, and the routine outputs mm, id, and iyyy as the month, day, and year on which the specified Julian Day started at noon.
INTEGER(I4B) :: ja,jalpha,jb,jc,jd,je
INTEGER(I4B), PARAMETER :: IGREG=2299161
if (julian >= IGREG) then
  jalpha=int(((julian-1867216)-0.25_dp)/36524.25_dp)  Cross-over to Gregorian Calendar produces this correction.
  ja=julian+1+jalpha-int(0.25_dp*jalpha)
else if (julian < 0) then
  ja=julian+36528*(1-julian/36525)  Make day number positive by adding integer number of Julian centuries, then subtract them off at the end.
else
  ja=julian
end if
jb=ja+1524
jc=int(6680.0_dp+((jb-2439870)-122.1_dp)/365.25_dp)
jd=365*jc+int(0.25_dp*jc)
je=int((jb-jd)/30.6001_dp)
id=jb-jd-int(30.6001_dp*je)
mm=je-1
if (mm > 12) mm=mm-12
iyyy=jc-4715
if (mm >= 2) iyyy=iyyy-1
if (iyyy <= 0) iyyy=iyyy-1
if (julian < 0) iyyy=iyyy-100*(1-julian/36525)
END SUBROUTINE caldat
Chapter B2. Solution of Linear Algebraic Equations

SUBROUTINE gaussj(a,b)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,outerand,outerprod,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: a,b

Linear equation solution by Gauss-Jordan elimination, equation (2.1.1). a is an N×N input coefficient matrix. b is an N×M input matrix containing M right-hand-side vectors. On output, a is replaced by its matrix inverse, and b is replaced by the corresponding set of solution vectors.

INTEGER(I4B), DIMENSION(size(a,1)) :: ipiv,indxr,indxc
LOGICAL(LGT), DIMENSION(size(a,1)) :: lpiv
REAL(SP) :: pivinv
REAL(SP), DIMENSION(size(a,1)) :: dumc
INTEGER(I4B), TARGET :: irc(2)
INTEGER(I4B) :: i,l,n
INTEGER(I4B), POINTER :: irow,icol

n=assert_eq(size(a,1),size(a,2),size(b,1),'gaussj')
irow => irc(1)
icol => irc(2)
ipiv=0

DO i=1,n
    lpiv = (lpiv == 0)            ! Begin search for a pivot element.
    irc=maxloc(abs(a),outerand(lpiv,lpiv))
    ipiv(irow)=ipiv(icol)+1
    IF (ipiv(icol) > 1) CALL nrerror('gaussj: singular matrix (1)')
    We now have the pivot element, so we interchange rows, if needed, to put the pivot element on the diagonal. The columns are not physically interchanged, only relabeled: indxc(i), the column of the ith pivot element, is the ith column that is reduced, while indxr(i) is the row in which that pivot element was originally located. If indxr(i) ≠ indxc(i) there is an implied column interchange. With this form of bookkeeping, the solution b's will end up in the correct order, and the inverse matrix will be scrambled by columns.
    IF (irow /= icol) THEN
        CALL swap(a(irow,:),a(icol,:))
        CALL swap(b(irow,:),b(icol,:))
    END IF
    indxr(i)=irow
    We are now ready to divide the pivot row by the pivot element, located at irow and icol.
    indxc(i)=icol
    IF (a(icol,icol) == 0.0) &
        CALL nrerror('gaussj: singular matrix (2)')
pivinv=1.0_sp/a(icol,icol)
a(icol,icol)=1.0
    a(icol,:)=a(icol,:)*pivinv
    b(icol,:)=b(icol,:)*pivinv
    dumc=a(:,icol)
    Next, we reduce the rows, except for the pivot one, of course.

END DO

END
Chapter B2. Solution of Linear Algebraic Equations

It only remains to unscramble the solution in view of the column interchanges. We do this by interchanging pairs of columns in the reverse order that the permutation was built up.

do l=n,1,-1
   call swap(a(:,indxr(l)),a(:,indxc(l)))
end do
END SUBROUTINE gaussj

irow => irc(1) ... icol => irc(2) The maxloc intrinsic returns the location of the maximum value of an array as an integer array, in this case of size 2. Pre-pointing pointer variables to components of the array that will be thus set makes possible convenient references to the desired row and column positions.

irc=maxloc(abs(a),outerand(lpiv,lpiv)) The combination of maxloc and one of the outer... routines from nrutil allows for a very concise formulation. If this task is done with loops, it becomes the ungainly “flying vee,”

\[ \text{aa}=0.0 \]
\[ \text{do i=1,n} \]
\[ \quad \text{if (lpiv(i)) then} \]
\[ \quad \quad \text{do j=1,n} \]
\[ \quad \quad \quad \text{if (lpiv(j)) then} \]
\[ \quad \quad \quad \quad \text{if (abs(a(i,j)) > aa) then} \]
\[ \quad \quad \quad \quad \quad \text{aa}=abs(a(i,j)) \]
\[ \quad \quad \quad \quad \text{irow}=i \]
\[ \quad \quad \quad \quad \text{icol}=j \]
\[ \quad \quad \text{endif} \]
\[ \quad \text{endif} \]
\[ \text{end do} \]
\[ \text{end do} \]
\[ \text{call swap(a(irow,:),a(icol,:))} \]

The swap routine (in nrutil) is concise and convenient. Fortran 90’s ability to overload multiple routines onto a single name is vital here: Much of the convenience would vanish if we had to remember variant routine names for each variable type and rank of object that might be swapped.

Even better, here, than overloading would be if Fortran 90 allowed user-written elemental procedures (procedures with unspecified or arbitrary rank and shape), like the intrinsic elemental procedures built into the language. Fortran 95 will, but Fortran 90 doesn’t.

One quick (if superficial) test for how much parallelism is achieved in a Fortran 90 routine is to count its do-loops, and compare that number to the number of do-loops in the Fortran 77 version of the same routine. Here, in gaussj, 13 do-loops are reduced to 2.

\[ a(1:icol-1,:)=... \]
\[ b(1:icol-1,:)=... \]
\[ a(icol+1:,:)=... \]
\[ b(icol+1:,:)=... \]
Here the same operation is applied to every row of \( a \), and to every row of \( b \), except row number \( i_{col} \). On a massively multiprocessor (MMP) machine it would be better to use a logical mask and do all of \( a \) in a single statement, all of \( b \) in another one. For a small-scale parallel (SSP) machine, the lines as written should saturate the machine’s concurrency, and they avoid the additional overhead of testing the mask.

This would be a good place to point out, however, that linear algebra routines written in Fortran 90 are likely never to be competitive with the hand-coded library routines that are generally supplied as part of MMP programming environments. If you are using our routines instead of library routines written specifically for your architecture, you are wasting cycles!

⋆⋆⋆

SUBROUTINE ludcmp(a, indx, d)

USE nrtype; USE nrutil, ONLY : assert_eq, imaxloc, nrerror, outerprod, swap

IMPLICIT NONE

REAL(SP), DIMENSION(:, :), INTENT(INOUT) :: a
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: indx
REAL(SP), INTENT(OUT) :: d

Given an \( N \times N \) input matrix \( a \), this routine replaces it by the \( LU \) decomposition of a rowwise permutation of itself. On output, \( a \) is arranged as in equation (2.3.14); \( indx \) is an output vector of length \( N \) that records the row permutation effected by the partial pivoting; \( d \) is output as \( \pm 1 \) depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with lubksb to solve linear equations or invert a matrix.

REAL(SP), DIMENSION(size(a,1)) :: vv

vv stores the implicit scaling of each row.

REAL(SP), PARAMETER :: TINY = 1.0e-20_sp

A small number.

INTEGER(I4B) :: j, n, imax

n = assert_eq(size(a,1), size(a,2), size(indx), 'ludcmp')
d = 1.0

No row interchanges yet.

vv = maxval(abs(a),dim=2)

Loop over rows to get the implicit scaling information.

if (any(vv == 0.0)) call nrerror('singular matrix in ludcmp')

There is a row of zeros.

vv = 1.0_sp/vv

Save the scaling.

do j=1,n

imax = (j-1)+imaxloc(vv(j:n)*abs(a(j:n,j)))

Find the pivot row.

if (j /= imax) then

Do we need to interchange rows?

call swap(a(imax,:), a(j,:))

Yes, do so...

d = -d

...and change the parity of \( d \).

dd = d

Also interchange the scale factor.

end if

indx(j) = imax

if (a(j,j) == 0.0) a(j,j) = TINY

If the pivot element is zero the matrix is singular (at least to the precision of the algorithm). For some applications on singular matrices, it is desirable to substitute TINY for zero.

a(j+1:n,j) = a(j+1:n,j)/a(j,j)

Divide by the pivot element.

a(j+1:n,j+1:n) = a(j+1:n,j+1:n) - outerprod(a(j+1:n,j), a(j,j+1:n))

Reduce remaining submatrix.

end do

END SUBROUTINE ludcmp

vv = maxval(abs(a), dim=2)  

A single statement finds the maximum absolute value in each row. Fortran 90 intrinsics like \texttt{maxval} generally “do their thing” in the dimension specified by \texttt{dim} and return a result with a shape corresponding to the other dimensions. Thus, here, \( vv \)’s size is that of the first dimension of \( a \).
Here we see why the `nrutil` routine `imaxloc` is handy: We want the index, in the range 1:n of a quantity to be searched for only in the limited range j:n. Using `imaxloc`, we just add back the proper offset of j-1. (Using only Fortran 90 intrinsics, we could write `imax=(j-1)+sum(maxloc(vv(j:n)*abs(a(j:n,j))))`, but the use of `sum` just to turn an array of length 1 into a scalar seems sufficiently confusing as to be avoided.)

```fortran
a(j+1:n,j+1:n)=a(j+1:n,j+1:n)-outerprod(a(j+1:n,j),a(j,j+1:n))
```

The Fortran 77 version of `ludcmp`, using Crout’s algorithm for the reduction, does not parallelize well: The elements are updated by $O(N^2)$ separate dot product operations in a particular order. Here we use a slightly different reduction, termed “outer product Gaussian elimination” by Golub and Van Loan \[1\], that requires just $N$ steps of matrix-parallel reduction. (See their §3.2.3 and §3.2.9 for the algorithm, and their §3.4.1 to understand how the pivoting is performed.) We use `nrutil`’s routine `outerprod` instead of the more cumbersome pure Fortran 90 construction:

```fortran
spread(a(j+1:n,j),dim=2,ncopies=n-j)*spread(a(j,j+1:n),dim=1,ncopies=n-j)
```

### SUBROUTINE lubksb(a,indx,b)

```fortran
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: a
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
```

Solves the set of $N$ linear equations $A \cdot X = B$. Here the $N \times N$ matrix $A$ is input, not as the original matrix $A$, but rather as its $LU$ decomposition, determined by the routine `ludcmp`. $indx$ is input as the permutation vector of length $N$ returned by `ludcmp`. $b$ is input as the right-hand-side vector $B$, also of length $N$, and returns with the solution vector $X$. $A$ and $indx$ are not modified by this routine and can be left in place for successive calls with different right-hand sides $b$. This routine takes into account the possibility that $b$ will begin with many zero elements, so it is efficient for use in matrix inversion.

```fortran
INTEGER(I4B) :: i,n,ii,ll
REAL(SP) :: summ
n=assert_eq(size(a,1),size(a,2),size(indx),'lubksb')
ii=0
```

When `ii` is set to a positive value, it will become the index of the first nonvanishing element of $b$. We now do the forward substitution, equation (2.3.6). The only new wrinkle is to unscramble the permutation as we go.

```fortran
do i=1,n
  ll=indx(i)
  summ=b(ll)
  b(ll)=b(i)
  if (ii /= 0) then
    summ=summ-dot_product(a(i,ii:i-1),b(ii:i-1))
  else if (summ /= 0.0) then
    ii=i
    A nonzero element was encountered, so from now on we will end if
    have to do the dot product above.
  end if
  b(i)=summ
end do
```

Now we do the backsubstitution, equation (2.3.7).

```fortran
  b(i) = (b(i)-dot_product(a(i,i+1:n),b(i+1:n)))/a(i,i)
end do
END SUBROUTINE lubksb
```

Conceptually, the search for the first nonvanishing element of $b$ (index `ii`) should be moved out of the first do-loop. However, in practice, the need to unscramble the permutation, and also considerations of performance...
on scalar machines, cause us to write this very scalar-looking code. The performance penalty on parallel machines should be minimal.

⋆⋆⋆

Serial and parallel algorithms for tridiagonal problems are quite different. We therefore provide separate routines tridag_ser and tridag_par. In the MODULE nr interface file, one or the other of these (your choice) is given the generic name tridag. Of course, either version will work correctly on any computer; it is only a question of efficiency. See §22.2 for the numbering of the equation coefficients, and for a description of the parallel algorithm.

SUBROUTINE tridag_ser(a,b,c,r,u)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
REAL(SP), DIMENSION(:), INTENT(OUT) :: u

Solves for a vector u of size N the tridiagonal linear set given by equation (2.4.1) using a serial algorithm. Input vectors b (diagonal elements) and r (right-hand sides) have size N, while a and c (off-diagonal elements) are size N – 1.

REAL(SP), DIMENSION(size(b)) :: gam One vector of workspace, gam is needed.
INTEGER(I4B) :: n,j
REAL(SP) :: bet

n=assert_eq((/size(a)+1,size(b),size(c)+1,size(r),size(u)/),'tridag_ser')

bet=b(1)
if (bet == 0.0) call nrerror('tridag_ser: Error at code stage 1')
If this happens then you should rewrite your equations as a set of order N – 1, with u2 trivially eliminated.

u(1)=r(1)/bet

do j=2,n
    gam(j)=c(j-1)/bet
    bet=b(j)-a(j-1)*gam(j)
    if (bet == 0.0) & Algorithm fails; see below routine in Vol. 1.
        call nrerror('tridag_ser: Error at code stage 2')
        u(j)=(r(j)-a(j-1)*u(j-1))/bet
    end do

    nx=j-1
    do j=nx,n,-1
        u(j)=u(j)-gam(j+1)*u(j+1)
    end do

END SUBROUTINE tridag_ser

RECURSIVE SUBROUTINE tridag_par(a,b,c,r,u)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : tridag_ser
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
REAL(SP), DIMENSION(:), INTENT(OUT) :: u

INTEGER(I4B), PARAMETER :: NPAR_TRIDAG=4

REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
REAL(SP), DIMENSION(:), INTENT(OUT) :: u

Solves for a vector u of size N the tridiagonal linear set given by equation (2.4.1) using a parallel algorithm. Input vectors b (diagonal elements) and r (right-hand sides) have size N, while a and c (off-diagonal elements) are size N – 1.

INTEGER(I4B), PARAMETER :: NPAR_TRIDAG=4

n=assert_eq((/size(a)+1,size(b),size(c)+1,size(r),size(u)/),'tridag_par')

if (n < NPAR_TRIDAG) then
    call tridag_ser(a,b,c,r,u)
else
    if (maxval(abs(b(1:n))) == 0.0) & Algorithm fails; see below routine in Vol. 1.

END SUBROUTINE tridag_par
call nrerror('tridag_par: possible singular matrix')
n2=size(y)
mx=size(pivc)
mx=size(x)
piva = a(1:n-1:2)/b(1:n-1:2)
      Zero the odd a's and even c's, giving x, y, z, q.
y(1:mm) = b(2:n-1:2)-piva(1:nm)*c(1:n-2:2)-pivc*a(2:n-1:2)
q(1:mm) = r(2:n-1:2)-piva(1:nm)*r(1:n-2:2)-pivc*r(3:n:2)
if (nm < n2) then
  y(n2) = b(n)-piva(n2)*c(n-1)
  q(n2) = r(n)-piva(n2)*r(n-1)
end if
x = -piva(2:n2)*a(2:n-2:2)
z = -pivc(1:nx)*c(3:n-1:2)
call tridag_par(x,y,z,q,u(2:n:2))
      Recurse and get even u's.
u(1) = (r(1)-c(1)*u(2))/b(1)
      Substitute and get odd u's.
   -c(3:n-1:2)*u(2:n-2:2))/b(3:n-2:2)
if (nm == n2) u(n)=(r(n)-a(n-1)*u(n-1))/b(n)
end if
END SUBROUTINE tridag_par

The serial version tridag_ser is called when the routine has recursed its way down to sufficiently small subproblems. The point at which this occurs is determined by the parameter NPAR_TRIDAG whose optimal value is likely machine-dependent. Notice that tridag_ser must here be called by its specific name, not by the generic tridag (which might itself be overloaded with either tridag_ser or tridag_par).

SUBROUTINE banmul(a,m1,m2,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq, arth
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
INTEGER(I4B), INTENT(IN) :: m1,m2
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(OUT) :: b
Matrix multiply b = A x, where A is band diagonal with m1 rows below the diagonal and m2 rows above. If the input vector x and output vector b are of length N, then the array a(1:N,1:m1+m2+1) stores A as follows: The diagonal elements are in a(1:N,m1+1). Subdiagonal elements are in a(j:N,1:m1) (with j > 1 appropriate to the number of elements on each subdiagonal). Superdiagonal elements are in a(1:j,m1+2:m1+m2+1) with j < N appropriate to the number of elements on each superdiagonal.

INTEGER(I4B) :: n
n=assert_eq(size(a,1),size(b),size(x),'banmul: n')

m=assert_eq(size(a,2),m1+m2+1,'banmul: m')
b=sum(a*eoshift(spread(x,dim=2,ncopies=m), &
      dim=1,shift=arth(-m1,1,m),dim=2))
END SUBROUTINE banmul

This is a good example of Fortran 90 at both its best and its worst: best, because it allows quite subtle combinations of fully parallel operations to be built up; worst, because the resulting code is virtually incomprehensible!
What is going on becomes clearer if we imagine a temporary array \( y \) with a declaration like  
\[
\text{REAL(SP), DIMENSION(size(a,1),size(a,2)) :: y}
\]
Then, the above single line decomposes into  
\[
y=\text{spread}(x,\text{dim}=2,\text{ncopies}=m) \quad \text{[Duplicate } x \text{ into columns of } y.]
\]
\[
y=\text{eoshift}(y,\text{dim}=1,\text{shift}=\text{arth}(-m1,1,m)) \quad \text{[Shift columns by a linear progression.]}
\]
\[
b=\text{sum}(a*y,\text{dim}=2) \quad \text{[Multiply by the band-diagonal elements, and sum.]}
\]
We use here a relatively rare subcase of the \text{eoshift} intrinsic, using a vector value for the \text{shift} argument to accomplish the simultaneous shifting of a bunch of columns, by different amounts (here specified by the linear progression returned by \text{arth}).

If you still don’t see how this accomplishes the multiplication of a band diagonal matrix by a vector, work through a simple example by hand.

SUBROUTINE bandec(a,m1,m2,al,indx,d)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,swap,arth
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: m1,m2
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: al
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: indx
REAL(SP), INTENT(OUT) :: d
REAL(SP), PARAMETER :: TINY=1.0e-20_sp

Given an \( N \times N \) band diagonal matrix \( A \) with \( m1 \) subdiagonal rows and \( m2 \) superdiagonal rows, compactly stored in the array \( a(1:N,1:m1+m2+1) \) as described in the comment for routine \text{banmul}, this routine constructs an \( LU \) decomposition of a rowwise permutation of \( A \). The upper triangular matrix replaces \( a \), while the lower triangular matrix is returned in \( al(1:N,1:m1) \). \text{indx} is an output vector of length \( N \) that records the row permutation effected by the partial pivoting; \( d \) is output as \( \pm 1 \) depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with \text{banbks} to solve band-diagonal sets of equations.

INTEGER(I4B) :: i,k,l,mdum,mm,n
REAL(SP) :: dum

\[
n=\text{assert_eq(size(a,1),size(al,1),size(indx),'bandec: n')}
\]
\[
mm=\text{assert_eq(size(a,2),m1+m2+1,'bandec: mm')}
\]
\[
mdum=\text{assert_eq(size(al,2),m1,'bandec: mdum')}
\]
\[
a(1:m1,:)=\text{eoshift}(a(1:m1,:),\text{dim}=2,\text{shift}=\text{arth}(m1,-1,m1)) \quad \text{Rearrange the storage a bit.}
\]
\[
d=1.0
\]
\[
do k=1,n \\
\quad l=\text{min}(m1+k,n) \\
\quad i=\text{imaxloc(abs(a(k:l,1)))+k-1} \\
\quad \text{dum}=a(i,1)
\]
\[
\quad \text{if (dum == 0.0) } a(k,1)=\text{TINY}
\]
\[
\quad \text{Matrix is algorithmically singular, but proceed anyway with TINY pivot (desirable in some applications)}.
\]
\[
\quad \text{indx}(k)=i \\
\quad \text{if (i /= k) } \text{then}
\]
\[
\quad \quad \text{Interchange rows.}
\quad \quad d=-d
\quad \quad \text{call swap}(a(k,1:mm),a(i,1:mm))
\quad \text{end if}
\]
\[
\quad \text{do i=k+1,l} \\
\quad \quad \text{Do the elimination.}
\quad \quad \text{dum}=a(i,1)/a(k,1)
\quad \quad a(k,i-k)=\text{dum}
\quad \quad a(i,1:mm-1)=a(i,2:mm)-\text{dum}*a(k,2:mm)
\quad \quad a(i,mm)=0.0
\quad \text{end do}
\]
\[
\text{end do}
\]
\[
\text{END SUBROUTINE bandec}
\]
See similar discussion of eoshift for banmul, just above.

\[ a(1:m1,:) = \text{eoshift}(a(1:m1,:),...) \]

See discussion of imaxloc on p. 1017.

Notice that the above is not well parallelized for MMP machines: the outer do-loop is done \( N \) times, where \( N \), the diagonal length, is potentially the largest dimension in the problem. Small-scale parallel (SSP) machines, and scalar machines, are not disadvantaged, because the parallelism of order \( mm = m1 + m2 + 1 \) in the inner loops can be enough to saturate their concurrency.

We don’t know of an \( N \)-parallel algorithm for decomposing band diagonal matrices, at least one that has any reasonably concise expression in Fortran 90. Conceptually, one can view a band diagonal matrix as a block tridiagonal matrix, and then apply the same recursive strategy as was used in tridag_par. However, the implementation details of this are daunting. (We would welcome a user-contributed routine, clear, concise, and with parallelism of order \( N \).)

### SUBROUTINE banbks(a,m1,m2,al,indx,b)

```fortran
USE nrtype; USE nrutil, ONLY : assert_eq, swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: a,al
INTEGER(I4B), INTENT(IN) :: m1,m2
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
```

Given the arrays \( a \), \( al \), and \( indx \) as returned from bandec, and given a right-hand-side vector \( b \), solves the band diagonal linear equations \( A \cdot x = b \). The solution vector \( x \) overwrites \( b \). The other input arrays are not modified, and can be left in place for successive calls with different right-hand sides.

```fortran
INTEGER(I4B) :: i,k,l,mdum,mm,n
n=assert_eq(size(a,1),size(al,1),size(b),size(indx),'banbks: n')
mm=assert_eq(size(a,2),m1+m2+1,'banbks: mm')
mdum=assert_eq(size(al,2),m1,'banbks: mdum')
do k=1,n
  Forward substitution, unscrambling the permuted rows as we go.
  l=min(n,m1+k)
  i=indx(k)
  if (i /= k) call swap(b(i),b(k))
  b(k+1:l)=b(k+1:l)-al(k,1:l-k)*b(k)
end do
```

```fortran
do i=n,1,-1
  Backsubstitution.
  l=min(mm,n-i+1)
  b(i)=(b(i)-dot_product(a(i,2:l),b(1+i:i+l-1)))/a(i,1)
end do
```

END SUBROUTINE banbks

As for bandec, the routine banbks is not parallelized on the large dimension \( N \), though it does give the compiler the opportunity for ample small-scale parallelization inside the loops.

* * *
SUBROUTINE mprove(a,alud,indx,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : lubksb
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: a,alud
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(IN) :: b
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: x

Improves a solution vector \( \mathbf{x} \) of the linear set of equations \( A \cdot \mathbf{X} = B \). The \( N \times N \) matrix \( A \) and the \( N \)-dimensional vectors \( \mathbf{b} \) and \( \mathbf{x} \) are input. Also input is \( \text{alud} \), the \( LU \) decomposition of \( A \) as returned by \text{ludcmp}, and the \( N \)-dimensional vector \( \text{indx} \) also returned by that routine. On output, only \( \mathbf{x} \) is modified, to an improved set of values.

SUBROUTINE svbksb_sp(u,w,v,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: u,v
REAL(SP), DIMENSION(:), INTENT(IN) :: w,b
REAL(SP), DIMENSION(:,), INTENT(OUT) :: x

Solves \( A \cdot \mathbf{X} = B \) for a vector \( \mathbf{X} \), where \( A \) is specified by the arrays \( u, v, w \) as returned by \text{svdcmp}. Here \( u \) is \( M \times N \), \( v \) is \( N \times N \), and \( w \) is of length \( N \). \( b \) is the \( M \)-dimensional input right-hand side. \( \mathbf{x} \) is the \( N \)-dimensional output solution vector. No input quantities are destroyed, so the routine may be called sequentially with different \( b \)'s.

```
assert_eq((/.../),'mprove')
```

This overloaded version of the \text{nrutil} routine \text{assert_eq} makes use of a trick for passing a variable number of scalar arguments to a routine: Put them into an array constructor, (\( {...} \)), and pass the array. The receiving routine can use the \text{size} intrinsic to count them. The technique has some obvious limitations: All the arguments in the array must be of the same type; and the arguments are passed, in effect, by \text{value}, not by address, so they must be, in effect, \text{INTENT(IN)}.

```
r=matmul(real(a,dp),real(x,dp))-real(b,dp)
```

Since Fortran 90's elemental intrinsics operate with the type of their arguments, we can use the \text{real()}\text{...}, \text{dp})'s to force the \text{matmul} matrix multiplication to be done in double precision, which is what we want. In Fortran 77, we would have to do the matrix multiplication with temporary double precision variables, both inconvenient and (since Fortran 77 has no dynamic memory allocation) a waste of memory.

```
tmp=matmul(b,u)/w
elsewhere
    tmp=0.0
end where
```

Calculate \( \text{diag}(1/w_j) \mathbf{U}^T \mathbf{B} \), but replace \( 1/w_j \) by zero if \( w_j = 0 \).
where \( w = 0.0 \)...tmp=...elsewhere...tmp=

Normally, when a where...elsewhere construction is used to set a variable (here tmp) to one or another value, we like to replace it with a merge expression. Here, however, the where is required to guarantee that a division by zero doesn’t occur. The rule is that where will never evaluate expressions that are excluded by the mask in the where line, but other constructions, like merge, might perform speculative evaluation of more than one possible outcome before selecting the applicable one.

Because singular value decomposition is something that one often wants to do in double precision, we include a double-precision version. In \textit{nr}, the single- and double-precision versions are overloaded onto the name \texttt{svbkbs}. 

\begin{verbatim}
SUBROUTINE svbkbs_dp(u,w,v,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
REAL(DP), DIMENSION(:,,:), INTENT(IN) :: u,v
REAL(DP), DIMENSION(:,), INTENT(IN) :: w,b
REAL(DP), DIMENSION(:,), INTENT(OUT) :: x
INTEGER(I4B) :: mdum,ndum
REAL(DP), DIMENSION(size(x)) :: tmp
mdum=assert_eq(size(u,1),size(b),'svbkbs_dp: mdum')
ndum=assert_eq((/size(u,2),size(v,1),size(v,2),size(w),size(x)/),
             'svbkbs_dp: ndum')
where (w /= 0.0)
    tmp=matmul(b,u)/w
elsewhere
    tmp=0.0
end where
x=matmul(v,tmp)
END SUBROUTINE svbkbs_dp
\end{verbatim}

\begin{verbatim}
SUBROUTINE svdcmp_sp(a,w,v)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,outerprod
USE nr, ONLY : pythag
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: w
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
Given an \( M \times N \) matrix \( a \), this routine computes its singular value decomposition, \( A = U \cdot W \cdot V^T \). The matrix \( U \) replaces \( a \) on output. The diagonal matrix of singular values \( W \) is output as the \( N \)-dimensional vector \( w \). The \( N \times N \) matrix \( V \) (not the transpose \( V^T \)) is output as \( v \).
INTEGER(I4B) :: i,its,j,k,l,m,n,nm
REAL(SP) :: anorm,c,f,g,h,s,scale,x,y,z
REAL(SP), DIMENSION(size(a,1)) :: tempm
REAL(SP), DIMENSION(size(a,2)) :: rv1,tempn
m=size(a,1)
n=assert_eq(size(a,2),size(v,1),size(v,2),size(w),'svdcmp_sp')
g=0.0
scale=0.0
do i=1,n
    Householder reduction to bidiagonal form.
    l=i+1
    rv1(i)=scale*g
    g=0.0
    scale=0.0
    if (i <= m) then
\end{verbatim}
scale = sum(abs(a(i:m, i)))
if (scale /= 0.0) then
  a(i:m, i) = a(i:m, i) / scale
  s = dot_product(a(i:m, i), a(i:m, i))
  f = a(i, i)
  g = sign(sqrt(s), f)
  b = f * g
  a(i, i) = f - g
  tempn(l:n) = matmul(a(i:m, i), a(i:m, l:n)) / h
  a(l:m, l:n) = a(l:m, l:n) + outerprod(a(i:m, i), tempn(l:n))
  a(i:m, i) = scale * a(i:m, i)
end if
end if

w(i) = scale * g
g = 0.0
scale = 0.0
if ((i <= m) .and. (i /= n)) then
  scale = sum(abs(a(i, l:n)))
  if (scale /= 0.0) then
    a(i, l:n) = a(i, l:n) / scale
    s = dot_product(a(i, l:n), a(i, l:n))
    f = a(i, l)
    g = sign(sqrt(s), f)
    h = f * g
    a(i, l) = f - g
    rv1(l:n) = a(i, l:n) / h
    tempn(l:n) = matmul(a(l:m, l:n), a(i, l:n))
    a(l:m, l:n) = a(l:m, l:n) + outerprod(tempn(l:m), rv1(l:n))
    a(i, l:n) = scale * a(i, l:n)
  end if
end if
end do

do i = n, 1, -1
  Accumulation of right-hand transformations.
  if (i < n) then
    if (g /= 0.0) then
      v(l:n, i) = (a(i, l:n) / a(i, l)) / g  \ Double division to avoid possible underflow.
      tempn(l:n) = matmul(a(l:m, l:n), v(l:n, l:n))
      v(l:n, l:n) = v(l:n, l:n) + outerprod(v(l:n, i), tempn(l:n))
    end if
    v(i, l:n) = 0.0
    v(l:n, i) = 0.0
  end if
  v(i, i) = 1.0
  g = rv1(i)
  l = i
end do

do i = min(m, n), 1, -1
  Accumulation of left-hand transformations.
  l = i + 1
  g = w(i)
  a(l, i:n) = 0.0
  if (g /= 0.0) then
    g = 1.0_sp / g
    tempn(l:n) = (matmul(a(l:m, i), a(l:m, l:n)) / a(i, i)) * g
    a(l:m, l:n) = a(l:m, l:n) + outerprod(a(l:m, i), tempn(l:n))
  else
    a(i:m, i) = 0.0
  end if
  a(i:m, i) = a(i:m, i) + 1.0_sp
end do

k = n, 1, -1
do its = 1, 30
  do l = k, 1, -1
    Test for splitting.
  end do
end do

Diagonalization of the bidiagonal form: Loop over singular values, and over allowed iterations.
nm=l-1
if ((abs(rv1(l))+anorm) == anorm) exit
Note that rv1(l) is always zero, so can never fall through bottom of loop.
if ((abs(w(nm))+anorm) == anorm) then
  c=0.0
  Cancellation of rv1(l), if l > 1.
s=1.0
  do i=l,k
    f=s*rv1(i)
    rv1(i)=c*rv1(i)
    if ((abs(f)+anorm) == anorm) exit
    g=w(i)
    h=pythag(f,g)
    w(i)=h
    h=1.0_sp/h
    c=(g*h)
    s=-((f*h))
    tempm(1:m)=a(1:m,nm)
    a(1:m,nm)=a(1:m,nm)*c+a(1:m,i)*s
    a(1:m,i)=tempm(1:m)*s+a(1:m,i)*c
  end do
  exit
end if
end do
z=w(k)
if (l == k) then
  Convergence.
  if (z < 0.0) then
    Singular value is made nonnegative.
    w(k)=-z
    v(1:n,k)=-v(1:n,k)
  end if
  exit
end if
if (its == 30) call nrerror('svdcmp_sp: no convergence in svdcmp')
  x=w(l)
  Shift from bottom 2-by-2 minor.
  nm=k-1
  y=w(nm)
  g=rv1(nm)
  h=rv1(k)
  f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0_sp*h*y)
  g=pythag(f,1.0_sp)
  f=((x-z)*(x+z)+h*((y/(f+sign(g,f)))-h))/x
  c=1.0
  Next QR transformation:
s=1.0
  do j=l,nm
    i=j+1
    g=rv1(i)
    y=w(i)
    h=sg
    g=c*sg
    z=pythag(f,h)
    rv1(j)=z
    c=f/z
    s=h/z
    f= (x+c)+(g*s)
    g= (x+s)+(g*c)
    h=sg
    y=y*c
    tempn(1:n)=v(1:n,j)
    v(1:n,j)=v(1:n,j)*c+v(1:n,i)*s
    v(1:n,i)=tempn(1:n)*s+v(1:n,i)*c
    z=pythag(f,h)
    w(j)=z
    Rotation can be arbitrary if z = 0.
    if (z /= 0.0) then
      z=1.0_sp/z
      c=f*z
    end if
  end do
Chapter B2. Solution of Linear Algebraic Equations

The SVD algorithm implemented above does not parallelize very well. There are two parts to the algorithm. The first, reduction to bidiagonal form, can be parallelized. The second, the iterative diagonalization of the bidiagonal form, uses QR transformations that are intrinsically serial. There have been proposals for parallel SVD algorithms [2], but we do not have sufficient experience with them yet to recommend them over the well-established serial algorithm.

Here is an example of an update as in equation (22.1.6). In this case $b_i$ is independent of $i$: It is simply $1/h$. The lines beginning $\text{tempm}(1:m) = \text{matmul}$ about 16 lines down are of a similar form, but with the terms in the opposite order in the $\text{matmul}$.

As with $\text{svbksb}$, single- and double-precision versions of the routines are overloaded onto the name $\text{svdcmp}$ in $\text{nr}$.

SUBROUTINE svdcmp_sp(a, w, v)
USE nrtype; USE nrutil, ONLY : assert_eq, nrerror, outerprod
USE nr, ONLY : pythag
IMPLICIT NONE
REAL(DP), DIMENSION(:,,:), INTENT(INOUT) :: a
REAL(DP), DIMENSION(:,), INTENT(OUT) :: w
REAL(DP), DIMENSION(:,,:), INTENT(OUT) :: v
INTEGER(I4B) :: i, its, j, k, l, m, n, nm
REAL(DP) :: anorm, c, f, g, h, s, scale, x, y, z
REAL(DP), DIMENSION(size(a,1)) :: tempm
REAL(DP), DIMENSION(size(a,2)) :: rv1, tempn
m = size(a,1)
n = assert_eq(size(a,1),size(a,2),size(v,1),size(v,2), size(w),'svdcmp_sp')
g = 0.0
scale = 0.0
DO i = 1, m
   rv1(i) = scale + g
   g = 0.0
   scale = 0.0
   IF (i <= m)
      scale = sum(abs(a(i:m, i)))
   IF (scale /= 0.0)
      a(i:m, i) = a(i:m, i)/scale
      s = dot_product(a(i:m, i), a(i:m, i))
      f = a(i, i)
      g = sign(sqrt(s), f)
h=f*g-s
a(i,i)=f-g
tempn(1:n)=matmul(a(1:m,i),a(1:m,1:n))/h
a(1:m,1:n)=a(1:m,1:n)+outerprod(a(1:m,i),tempn(1:n))
a(i,1)=scale*a(i,1)
end if
end if
w(i)=scale*g
g=0.0
scale=0.0
if ((i < m) .and. (i /= n)) then
    scale=norm(abs(a(i,1:n)))
    if (scale /= 0.0) then
        a(1,1)=a(1,1)/scale
        s=dot_product(a(1,1:i),a(1,1:i))
        f=a(1,1)
        g=sign(sqrt(s),f)
        h=f*g-s
        rv1(1:n)=a(1,1:n)/h
        tempn(1:m)=matmul(a(1:m,1:n),a(1:m,1:n))
        a(1:m,1:n)=a(1:m,1:n)+outerprod(tempn(1:m),rv1(1:n))
        a(1,1:n)=scale*a(1,1:n)
    end if
end if
end do
anorm=maxval(abs(w)+abs(rv1))
do i=n,1,-1
    if (i < n) then
        if (g /= 0.0) then
            v(l:n,i)=(a(i,l:n)/a(i,i))/g
            tempn(l:n)=matmul(a(i,l:n),v(l:n,l:n))
            v(l:n,l:n)=v(l:n,l:n)+outerprod(v(l:n,i),tempn(l:n))
        end if
    end if
    v(i,1:n)=0.0
    v(l:n,i)=0.0
end if
v(i,i)=1.0
g=rv1(i)
l=1
end do
do i=min(m,n),1,-1
    l=1+i
    a(1,1:n)=0.0
    if (g /= 0.0) then
        g=1.0_dp/g
        tempn(1:n)=matmul(a(1:m,1:n),a(1:m,1:n))/a(1,1)
        a(1:m,1:n)=a(1:m,1:n)+outerprod(a(1:m,1:n),tempn(1:n))
        a(1,1)=a(1,1)*g
    else
        a(1,1)=0.0
    end if
    a(i,i)=a(i,i)+1.0_dp
end if
end do
do k=n,1,-1
    do its=1,30
        nm=1-l
        if ((abs(rv1(l))+anorm) == anorm) exit
        if ((abs(w(nm))+anorm) == anorm) then
            c=0.0
            s=1.0
            do i=1,k
f=s*rv1(i)
rv1(i)=c*rv1(i)
if ((abs(f)+anorm) == anorm) exit
g=w(i)
h=pythag(f,g)
v(i)=h
h=1.0_dp/h
c= (g*h)
s=-(f*h)
tempm(1:m)=a(1:m,nm)
a(1:m,nm)=a(1:m,nm)*c+a(1:m,i)*s
a(1:m,i)=tempm(1:m)*s+a(1:m,i)*c
end do
exit
end if
end do
z=w(k)
if (l == k) then
if (z < 0.0) then
w(k)=-z
v(1:n,k)=-v(1:n,k)
end if
exit
end if
if (its == 30) call nrerror('svdcmp_dp: no convergence in svdcmp')
x=w(l)
nm=k-1
y=w(nm)
g=rv1(nm)
h=rv1(k)
f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0_dp*h*y)
g=pythag(f,1.0_dp)
f=((x-z)*(x+z)+h*((y/(f+sign(g,f)))-h))/x
c=1.0_dp
s=1.0_dp
do j=l,nm
i=j+1
g=rv1(i)
y=w(i)
h=s*g
g=c*g
z=pythag(f,h)
rv1(j)=z
c=f/z
s=h/z
f= (x*c)+(g*s)
g= (x*s)+(g*c)
b=y=s
y=y*c
tempn(1:n)=v(1:n,j)
v(1:n,j)=v(1:n,j)*c+v(1:n,i)*s
v(1:n,i)=tempn(1:n)*s+v(1:n,i)*c
z=pythag(f,h)
v(j)=z
if (z /= 0.0) then
z=1.0_dp/z
c=f*z
s=h*z
end if
f= (c*e)+(s*y)
x=-(s*e)+(c*y)
tempm(1:m)=a(1:m,j)
a(1:m,j)=a(1:m,j)*c+a(1:m,i)*s
a(1:m,i)=tempm(1:m)*s+a(1:m,i)*c
FUNCTION pythag_sp(a,b)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: pythag_sp
Computes \((a^2 + b^2)^{1/2}\) without destructive underflow or overflow.
REAL(SP) :: absa,absb
absa=abs(a)
absb=abs(b)
if (absa > absb) then
pythag_sp=absa*sqrt(1.0_sp+(absb/absa)**2)
else
  if (absb == 0.0) then
    pythag_sp=0.0
  else
    pythag_sp=absb*sqrt(1.0_sp+(absa/absb)**2)
  end if
end if
END FUNCTION pythag_sp

FUNCTION pythag_dp(a,b)
USE nrtype
IMPLICIT NONE
REAL(DP), INTENT(IN) :: a,b
REAL(DP) :: pythag_dp
REAL(DP) :: absa,absb
absa=abs(a)
absb=abs(b)
if (absa > absb) then
pythag_dp=absa*sqrt(1.0_dp+(absb/absa)**2)
else
  if (absb == 0.0) then
    pythag_dp=0.0
  else
    pythag_dp=absb*sqrt(1.0_dp+(absa/absb)**2)
  end if
end if
END FUNCTION pythag_dp

* * *

end do
rv1(l)=0.0
rv1(k)=f
w(k)=x
end do
END SUBROUTINE svdcmp_dp
SUBROUTINE cyclic(a,b,c,alpha,beta,r,x)
USE nrtype; USE nrutil, ONLY : assert, assert_eq
USE nr, ONLY: tridag
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN):: a,b,c,r
REAL(SP), INTENT(IN) :: alpha,beta
REAL(SP), DIMENSION(:), INTENT(OUT):: x

Solves the "cyclic" set of linear equations given by equation (2.7.9). a, b, c, and r are input vectors, while x is the output solution vector, all of the same size. alpha and beta are the corner entries in the matrix. The input is not modified.

INTEGER(I4B) :: n
REAL(SP) :: fact,gamma
REAL(SP), DIMENSION(size(x)) :: bb,u,z

n=assert_eq((/size(a),size(b),size(c),size(r),size(x)/),'cyclic')
call assert(n > 2, 'cyclic arg')
gamma=-b(1)
b(1)=b(1)-gamma
bb(1)=b(1)+alpha*beta/gamma
bb(2:n-1)=b(2:n-1)
call tridag(a(2:n),bb,c(1:n-1),r,x)

u(1)=gamma
u(n)=alpha
u(2:n-1)=0.0

call tridag(a(2:n),bb,c(1:n-1),u,z)

fact=(x(1)+beta*x(n)/gamma)/(1.0_sp+z(1)+beta*z(n)/gamma)
x=x-fact*z

END SUBROUTINE cyclic

The parallelism in cyclic is in tridag. Users with multiprocessor machines will want to be sure that, in nrutil, they have set the name tridag to be overloaded with tridag_par instead of tridag_ser.

* * *

The routines sprsin, sprsax, sprstx, sprstp, and sprsdiag give roughly equivalent functionality to the corresponding Fortran 77 routines, but they are not plug compatible. Instead, they take advantage of (and illustrate) several Fortran 90 features that are not present in Fortran 77.

In the module nrtype we define a TYPE sprs2_sp for two-dimensional sparse, square, matrices, in single precision, as follows

TYPE sprs2_sp
  INTEGER(I4B) :: n,len
  REAL(SP), DIMENSION(:,), POINTER :: val
  INTEGER(I4B), DIMENSION(:,), POINTER :: irow
  INTEGER(I4B), DIMENSION(:,), POINTER :: jcol
END TYPE sprs2_sp

This has much less structure to it than the "row-indexed sparse storage mode" used in Volume 1. Here, a sparse matrix is just a list of values, and corresponding lists giving the row and column number that each value is in. Two integers n and len give, respectively, the underlying size (number of rows or columns) in the full matrix, and the number of stored nonzero values. While the previously used row-indexed scheme can be somewhat more efficient for serial machines, it does not parallelize conveniently, while this one does (though with some caveats; see below).
SUBROUTINE sprsin_sp(a,thresh,sa)
USE nrtype; USE nrutil, ONLY : arth,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: a
REAL(SP), INTENT(IN) :: thresh
TYPE(sprs2_sp), INTENT(OUT) :: sa
Converts a square matrix a to sparse storage format as sa. Only elements of a with magnitude $\geq$ thresh are retained.
INTEGER(I4B) :: n,len
LOGICAL(LGT), DIMENSION(size(a,1),size(a,2)) :: mask
n=assert_eq(size(a,1),size(a,2),'sprsin_sp')
mask=abs(a)>thresh
len=count(mask) How many elements to store?
allocate(sa%val(len),sa%irow(len),sa%jcol(len))
sa%nn=n
sa%len=len
sa%val=pack(a,mask) Grab the values, row, and column numbers.
sa%irow=pack(spread(arth(1,1,n),2,n),mask)
sa%jcol=pack(spread(arth(1,1,n),1,n),mask)
END SUBROUTINE sprsin_sp

SUBROUTINE sprsin_dp(a,thresh,sa)
USE nrtype; USE nrutil, ONLY : arth,assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,,:), INTENT(IN) :: a
REAL(DP), INTENT(IN) :: thresh
TYPE(sprs2_dp), INTENT(OUT) :: sa
INTEGER(I4B) :: n,len
LOGICAL(LGT), DIMENSION(size(a,1),size(a,2)) :: mask
n=assert_eq(size(a,1),size(a,2),'sprsin_dp')
mask=abs(a)>thresh
len=count(mask) len=how many elements to store?
allocate(sa%val(len),sa%irow(len),sa%jcol(len))
sa%nn=n
sa%len=len
sa%val=pack(a,mask)
sa%irow=pack(spread(arth(1,1,n),2,n),mask)
sa%jcol=pack(spread(arth(1,1,n),1,n),mask)
END SUBROUTINE sprsin_dp

Note that the routines sprsin_sp and sprsin_dp — single and double precision versions of the same algorithm — are overloaded onto the name sprsin in module nr. We supply both forms because the routine linbcg below, works in double precision.

sa%irow=pack(spread(arth(1,1,n),2,n),mask) The trick here is to use the same mask, abs(a)>thresh, in three consecutive pack expressions, thus guaranteeing that the corresponding elements of the array argument get selected for packing. The first time, we get the desired matrix element values. The second time (above code fragment), we construct a matrix with each element having the value of its row number. The third time, we construct a matrix with each element having the value of its column number.
SUBROUTINE sprsax_sp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq, scatter_add
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION (:), INTENT(IN) :: x
REAL(SP), DIMENSION (:), INTENT(OUT) :: b
    INTEGER(I4B) :: ndum
    ndum=assert_eq(sa%n,size(x),size(b),'sprsax_sp')
    b=0.0_sp
    call scatter_add(b,sa%val*x(sa%jcol),sa%irow)
    Each sparse matrix entry adds a term to some component of b.
END SUBROUTINE sprsax_sp

SUBROUTINE sprsax_dp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq, scatter_add
IMPLICIT NONE
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION (:), INTENT(IN) :: x
REAL(DP), DIMENSION (:), INTENT(OUT) :: b
    INTEGER(I4B) :: ndum
    ndum=assert_eq(sa%n,size(x),size(b),'sprsax_dp')
    b=0.0_dp
    call scatter_add(b,sa%val*x(sa%jcol),sa%irow)
END SUBROUTINE sprsax_dp

call scatter_add(b,sa%val*x(sa%jcol),sa%irow) Since more than one component of the middle vector argument will, in general, need to be added into the same component of b, we must resort to a call to the nrutil routine scatter_add to achieve parallelism. However, this parallelism is achieved only if a parallel version of scatter_add is available! As we have discussed previously (p. 984), Fortran 90 does not provide any scatter-with-combine (here, scatter-with-add) facility, insisting instead that indexed operations yield non-colliding addresses. Luckily, almost all parallel machines do provide such a facility as a library program. In HPF, for example, the equivalent of scatter_add is SUM_SCATTER.

The call to scatter_add above is equivalent to the do-loop

\[
\begin{align*}
    b = 0.0 \\
    \text{do } k = 1, sa\%len \\
    \quad b(sa%irow(k)) = b(sa%irow(k)) + sa%val(k) \times (sa%jcol(k)) \\
    \text{end do}
\end{align*}
\]

SUBROUTINE sprstx_sp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq, scatter_add
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION (:), INTENT(IN) :: x
REAL(SP), DIMENSION (:), INTENT(OUT) :: b
    INTEGER(I4B) :: ndum
    ndum=assert_eq(sa%n,size(x),size(b),'sprstx_sp')
    b=0.0_sp
    call scatter_add(b,sa%val*x(sa%irow),sa%jcol)
    Each sparse matrix entry adds a term to some component of b.
END SUBROUTINE sprstx_sp
SUBROUTINE sprstx_dp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq, scatter_add
IMPLICIT NONE
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION (:), INTENT(IN) :: x
REAL(DP), DIMENSION (:), INTENT(OUT) :: b
INTEGER(I4B) :: ndum
ndum=assert_eq(sa%n,size(x),size(b),'sprstx_dp')
b=0.0_dp
call scatter_add(b,sa%val*x(sa%irow),sa%jcol)
END SUBROUTINE sprstx_dp

Precisely the same comments as for sprsax apply to sprstx. The call
to scatter_add is here equivalent to

\[
b=0.0
\]
\[
do k=1,sa%len
\]
\[
  b(sa%jcol(k))=b(sa%jcol(k))+sa%val(k)*x(sa%irow(k))
\]
\[
end do
\]

SUBROUTINE sprstp(sa)
USE nrtype
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(INOUT) :: sa
Replaces sa, in sparse matrix format, by its transpose.
INTEGER(I4B), DIMENSION(:), POINTER :: temp
temp=>sa%irow
We need only swap the row and column pointers.
sa%irow=>sa%jcol
sa%jcol=>temp
END SUBROUTINE sprstp

SUBROUTINE sprsdiag_sp(sa,b)
USE nrtype; USE nrutil, ONLY : array_copy, assert_eq
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION(:), INTENT(OUT) :: b
Extracts the diagonal of a matrix sa in sparse matrix format into a vector b.
REAL(SP), DIMENSION(size(b)) :: val
INTEGER(I4B) :: k,1,ndum,nerr
INTEGER(I4B), DIMENSION(size(b)) :: i
LOGICAL(LGT), DIMENSION(:), ALLOCATABLE :: mask
ndum=assert_eq(sa%n,size(b),'sprsdiag_sp')
l=sa%len
allocate(mask(l))
mask = (sa%irow(1:l) == sa%jcol(1:l))  Find diagonal elements.
call array_copy(pack(sa%val(1:l),mask),val,k,nerr)  Grab the values...
i(1:k)=pack(sa%irow(1:l),mask)  ...and their locations.
deallocate(mask)
b=0.0  Zero b because zero values not stored in sa.
b(i(1:k))=val(1:k)  Scatter values into correct slots.
END SUBROUTINE sprsdiag_sp
SUBROUTINE sprsdiag_dp(sa,b)
USE nrtype; USE nrutil, ONLY : array_copy, assert_eq
IMPLICIT NONE
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(dp), DIMENSION(:), INTENT(OUT) :: b
INTEGER(4B) :: k, ndum, nerr
INTEGER(4B), DIMENSION(size(b)) :: i
LOGICAL(LGT), DIMENSION(:), ALLOCATABLE :: mask
ndum=assert_eq(sa%n,size(b),'sprsdiag_dp')
l=sa%n
allocate(mask(l))
mask = (sa%irow(1:l) == sa%jcol(1:l))
call array_copy(pack(sa%val(1:l),mask),val,k,nerr)
i(1:k)=pack(sa%irow(1:l),mask)
deallocate(mask)
b = 0.0
b(i(1:k))=val(1:k)
END SUBROUTINE sprsdiag_dp

We use the nrutil routine array_copy because we don't know in advance how many nonzero diagonal elements will be selected by mask. Of course we could count them with a count(mask), but this is an extra step, and inefficient on scalar machines.

Using the same mask, we pick out the corresponding locations of the diagonal elements. No need to use array_copy now, since we know the value of k.

Finally, we can put each element in the right place. Notice that if the sparse matrix is ill-formed, with more than one value stored for the same diagonal element (which should not happen!) then the vector subscript i(1:k) is a "many-one section" and its use on the left-hand side is illegal.

* * *

SUBROUTINE linbcg(b,x,itol,tol,itmax,iter,err)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : atimes, asolve, snrm
IMPLICIT NONE
REAL(dp), DIMENSION(:), INTENT(IN) :: b
REAL(dp), DIMENSION(:,), INTENT(INOUT) :: x
INTEGER(4B), INTENT(IN) :: itol, itmax
REAL(dp), INTENT(INOUT) :: tol, itmax
REAL(dp), INTENT(IN) :: tol, itmax
REAL(dp), INTENT(IN) :: x
 REAL(dp), INTENT(IN) :: tol, itmax
REAL(dp), INTENT(INOUT) :: tol, itmax
REAL(dp), PARAMETER :: EPS=1.0e-14_dp

Solves \( Ax = b \) for \( x \), given \( b \) of the same length, by the iterative biconjugate gradient method. On input \( x \) should be set to an initial guess of the solution (or all zeros); \( itol \) is 1, 2, 3, or 4, specifying which convergence test is applied (see text); \( itmax \) is the maximum number of allowed iterations; and \( tol \) is the desired convergence tolerance. On output, \( x \) is reset to the improved solution, \( iter \) is the number of iterations actually taken, and \( err \) is the estimated error. The matrix \( A \) is referenced only through the user-supplied routines atimes, which computes the product of either \( A \) or its transpose on a vector; and asolve,
which solves \( \tilde{A} \cdot x = b \) or \( \tilde{A}^T \cdot x = b \) for some preconditioner matrix \( \tilde{A} \) (possibly the trivial diagonal part of \( A \)).

```fortran
INTEGER(I4B) :: n
REAL(DP) :: ak,akden,bk,bknum,bnrm,dxnrm,xnrm,zm1nrm,znrm
REAL(DP), DIMENSION(size(b)) :: p,pp,r,rr,z,zz
n=assert_eq(size(b),size(x),'linbcg')
iter=0
call atimes(x,r,0) ! Calculate initial residual. Input to atimes is x(1:n), output is r(1:n); the final 0 indicates that the matrix (not its transpose) is to be used.
r=b-r
rr=r!
call atimes(r,rr,0)
!
! Uncomment this line to get the "minimum residual" variant of the algorithm.
select case(itol)
  ! Calculate norms for use in stopping criterion, and initialize z.
  case(1)
    bnrm=snrm(b,itol)
call asolve(r,z,0) ! Input to asolve is r(1:n), output is z(1:n); the final 0 indicates that the matrix \( \tilde{A} \) (not its transpose) is to be used.
    bnrm=snrm(z,itol)
call asolve(r,z,0)
  case(3:4)
    call asolve(b,z,0)
    bnrm=snrm(z,itol)
call asolve(r,z,0)
    znrm=snrm(z,itol)
  case default
    call nrerror('illegal itol in linbcg')
end select
!
!
! Main loop.
!
! if (iter > itmax) exit
iter=iter+1
!
call asolve(rr,zz,1) ! Final 1 indicates use of transpose matrix \( \tilde{A}^T \).
!
! Calculate coefficient bk and direction vectors p and pp.
bknum=dot_product(z,rr)
if (iter == 1) then
  p=z
  pp=zz
else
  bk=bknum/bkden
  p=bk*p+z
  pp=bk*pp+zz
end if
!
call atimes(p,z,0)
!
! Calculate coefficient ak, new iterate x, and new residuals r and rr.
ak=bknum/akden
!
call atimes(pp,zz,1)
x=x+ak*p
r=r-ak*z
rr=rr-ak*zz
!
call solve(r,z,0)
!
! Solve \( \tilde{A} \cdot z = r \) and check stopping criterion.
select case(itol)
  case(1)
    err=snrm(r,itol)/bnrm
  case(2)
    err=snrm(z,itol)/bnrm
  case(3:4)
    zm1nrm=znrm
    znrm=snrm(z,itol)
    if (abs(zm1nrm-znrm) > EPS*znrm) then
      dxnrm=abs(ak)*snrm(p,itol)
      err=znrm/abs(zm1nrm-znrm)+dxnrm
    else
      err=znrm/bnrm
    end if
end select
!
! Error may not be accurate, so loop again.
! cycle
```
case default...call nrerror('illegal itol in linbcg') It's always a good idea to trap errors when the value of a case construction is supplied externally to the routine, as here.

FUNCTION snrm(sx,itol)
USE nrtype
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: sx
INTEGER(I4B), INTENT(IN) :: itol
REAL(DP) :: snrm
Compute one of two norms for a vector sx, as signaled by itol. Used by linbcg.
if (itol <= 3) then
    snrm=sqrt(dot_product(sx,sx)) Vector magnitude norm.
else
    snrm=maxval(abs(sx)) Largest component norm.
end if
END FUNCTION snrm

SUBROUTINE atimes(x,r,itrnsp)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : sprsax,sprstx DOUBLE PRECISION versions of sprsax and sprstx.
USE xlinbcg_data The matrix is accessed through this module.
REAL(DP), DIMENSION(:), INTENT(IN) :: x
REAL(DP), DIMENSION(:), INTENT(OUT) :: r
INTEGER(I4B), INTENT(IN) :: itrnsp
INTEGER(I4B) :: n
n=assert_eq(size(x),size(r),'atimes')
if (itrnsp == 0) then
call sprsax(sa,x,r)
else
call sprstx(sa,x,r)
end if
END SUBROUTINE atimes
SUBROUTINE asolve(b,x,itrnsp)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : sprsdiag DOUBLE PRECISION version of sprsdiag.
USE xlinbcg_data This matrix is accessed through this module.
REAL(DP), DIMENSION(:), INTENT(IN) :: b
REAL(DP), DIMENSION(:), INTENT(OUT) :: x
INTEGER(I4B), INTENT(IN) :: itrnsp
INTEGER(I4B) :: ndum
ndum=assert_eq(size(b),size(x),'asolve')
call sprsdiag(sa,x)
The matrix $A$ is taken to be the diagonal part of $A$. Since the transpose matrix has the same
diagonal, the flag itrnsp is not used.
if (any(x == 0.0)) call nrerror('asolve: singular diagonal matrix')
x=b/x
END SUBROUTINE asolve

The routines atimes and asolve are examples of user-supplied routines
that interface linbcg to a user-supplied method for multiplying the
user's sparse matrix by a vector, and for solving the preconditioner matrix
equation. Here, we have used these routines to connect linbcg to the sparse matrix
machinery developed above. If we were instead using the different sparse matrix
machinery of Volume 1, we would modify atimes and asolve accordingly.

USE xlinbcg_data This user-supplied module is assumed to have sa (the
sparse matrix) in it.

FUNCTION vander(x,q)
USE nrtype; USE nrutil, ONLY : assert_eq,outerdiff
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: x,q
REAL(DP), DIMENSION(size(x)) :: vander
Solves the Vandermonde linear system $\sum_{i=1}^{N} x^{k-1}w_i = q_k$ ($k = 1, \ldots, N$). Input consists
of the vectors $x$ and $q$ of length $N$. The solution $w$ (also of length $N$) is returned in vander.
REAL(DP), DIMENSION(size(x)) :: c
REAL(DP), DIMENSION(size(x)) :: a
INTEGER(I4B) :: i,n
n=assert_eq(size(x),size(q),'vander')
if (n == 1) then
 vander(1)=q(1)
else
 c(:)=0.0
 c(n)=x(1)
 do i=2,n
 c(n+1-i:n-1)=c(n+1-i:n-1)-x(i)*c(n+2-i:n)
c(n)=c(n)-x(i)
d end do
 a(:,1)=-c(1)/x(:)
do i=2,n
 a(:,i)=-(c(i)-a(:,i-1))/x(:)
d end do
end if
vander(:)=matmul(a,q)/vander(:)
END FUNCTION vander
Here is an example of the coding of equation (22.1.4). Since in this case the product is over the second index \((n \times x_n)\), we have \(\text{dim}=2\) in the product.

```fortran
FUNCTION toeplz(r,y)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: r,y
REAL(SP), DIMENSION(size(y)) :: toeplz
Solves the Toeplitz system \(
\sum_{j=1}^{N} R(N+i-j)x_j = y_i \quad (i=1,\ldots,N)
\)
need not be symmetric. \(y\) (of length \(N\)) and \(x\) (of length \(2N-1\)) are input arrays; the solution \(x\) (of length \(N\)) is returned in \(toeplz\).
INTEGER(14B) :: m,m1,n,ndum
REAL(SP) :: sd,sgd,sgn,shn,sxn
REAL(SP), DIMENSION(size(y)) :: g,h,t
n=size(y)
ndum=assert_eq(2*n-1,size(r),'toeplz: ndum')
if (r(n) == 0.0) call nrerror('toeplz: initial singular minor')
toeplz(1)=y(1)/r(n) Initialize for the recursion.
if (n == 1) RETURN

\(g(1)=r(n-1)/r(n)\)
\(h(1)=r(n+1)/r(n)\)
do m=1,n
Main loop over the recursion.
m1=m+1
\(sxn=y(m1)+\text{dot_product}(r(n+1:n+m),toeplz(m:1:-1))\)
Compute numerator and denominator for \(x\).
sdn=\(r(n)+\text{dot_product}(r(n+1:n+m),g(1:m))\)
if (sd == 0.0) exit
\(toeplz(m1)=sxn/sdn\) whence \(x\).
toeplz(1:m)=toeplz(1:m)-toeplz(m1)*g(1:m)
if (m1 == n) RETURN

\(sgn=r(n-m1)+\text{dot_product}(r(n-m:n-1),g(1:m))\)
\(shn=r(n+m1)+\text{dot_product}(r(n+m:n+1:-1),h(1:m))\)
\(sgd=r(n)+\text{dot_product}(r(n-m:n-1),h(1:m))\)
Compute numerator and denominator for \(G\) and \(H\).
if (sgd == 0.0) exit
\(g(m1)=sgn/sgd\) whence \(G\) and \(H\).
h(m1)=shn/sgd
\(t(1:m)=g(1:m)\)
\(g(1:m)=g(1:m)-g(m1)*h(1:m1)\)
h(1:m)=h(1:m)-h(m1)*t(1:m1)
end do Back for another recurrence.
if (m > n) call nrerror('toeplz: sanity check failed in routine')
call nrerror('toeplz: singular principal minor')
END FUNCTION toeplz
```

```fortran
SUBROUTINE choldc(a,p)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: p
Given an \(N \times N\) positive-definite symmetric matrix \(a\), this routine constructs its Cholesky decomposition, \(A = L \cdot L^T\). On input, only the upper triangle of \(a\) need be given; it is not modified. The Cholesky factor \(L\) is returned in the lower triangle of \(a\), except for its diagonal elements, which are returned in \(p\), a vector of length \(N\).
INTEGER(14B) :: i,n
REAL(SP) :: summ
n=assert_eq(size(a,1),size(a,2),size(p),'choldc')
do i=1,n
```

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. ... or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).
SUBROUTINE choldc(a,p,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: p,b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x

Solves the set of \( N \) linear equations \( A \cdot x = b \), where \( a \) is a positive-definite symmetric matrix, \( a \) (\( N \times N \)) and \( p \) (of length \( N \)) are input as the output of the routine choldc. Only the lower subdiagonal portion of \( a \) is accessed. \( b \) is the input right-hand-side vector, of length \( N \). The solution vector, also of length \( N \), is returned in \( x \). \( a \) and \( p \) are not modified and can be left in place for successive calls with different right-hand sides. \( b \) is not modified unless you identify \( b \) and \( x \) in the calling sequence, which is allowed.

INTEGER(I4B) :: i,n
n=assert_eq((/size(a,1),size(a,2),size(p),size(b),size(x)/),'cholsl')
do i=1,n
Solve \( L \cdot y = b \), storing \( y \) in \( x \).
   x(i)=(b(i)-dot_product(a(i,1:i-1),x(1:i-1)))/p(i)
end do
do i=n,1,-1
Solve \( L^T \cdot x = y \).
   x(i)=(x(i)-dot_product(a(i+1:n,i),x(i+1:n)))/p(i)
end do
END SUBROUTINE cholds

SUBROUTINE cholsl(a,p,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: p,b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x

Solves the set of \( N \) linear equations \( A \cdot x = b \), where \( a \) is a positive-definite symmetric matrix, \( a \) (\( N \times N \)) and \( p \) (of length \( N \)) are input as the output of the routine choldc. Only the lower subdiagonal portion of \( a \) is accessed. \( b \) is the input right-hand-side vector, of length \( N \). The solution vector, also of length \( N \), is returned in \( x \). \( a \) and \( p \) are not modified and can be left in place for successive calls with different right-hand sides. \( b \) is not modified unless you identify \( b \) and \( x \) in the calling sequence, which is allowed.

INTEGER(I4B) :: i,n
n=assert_eq((/size(a,1),size(a,2),size(p),size(b),size(x)/),'cholsl')
do i=1,n
Solve \( L \cdot y = b \), storing \( y \) in \( x \).
   x(i)=(b(i)-dot_product(a(i,1:i-1),x(1:i-1)))/p(i)
end do
do i=n,1,-1
Solve \( L^T \cdot x = y \).
   x(i)=(x(i)-dot_product(a(i+1:n,i),x(i+1:n)))/p(i)
end do
END SUBROUTINE chols

SUBROUTINE qrdcmp(a,c,d,sing)
USE nrtype; USE nrutil, ONLY : assert_eq,outerprod,vabs
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: c,d
LOGICAL(LGT), INTENT(OUT) :: sing

Constructs the QR decomposition of the \( n \times n \) matrix \( a \). The upper triangular matrix \( R \) is returned in the upper triangle of \( a \), except for the diagonal elements of \( R \), which are returned in the \( n \)-dimensional vector \( d \). The orthogonal matrix \( Q \) is represented as a product of \( n-1 \) Householder matrices \( Q_1 \cdots Q_{n-1} \), where \( Q_j = I - u_j \otimes u_j / c_j \). The \( i \)th component of \( u_j \) is zero for \( i = 1, \ldots, j-1 \) while the nonzero components are returned in \( a(i,j) \) for \( i = j, \ldots, n \). \( \text{sing} \) returns as true if singularity is encountered during the decomposition, but the decomposition is still completed in this case.

INTEGER(I4B) :: k,n
REAL(SP) :: scale,sigma
n=assert_eq((/size(a,1),size(a,2),size(c),size(d)/),'qrdcmp')
sing=.false.
do k=1,n-1
   scale=maxval(abs(a(k:n,k)))
   if (scale == 0.0) then
      Singular case.
      sing=.true.
      c(k)=0.0
      d(k)=0.0
   else
      Form \( Q_k \) and \( Q_k \cdot A \).
      a(k:n,k)=a(k:n,k)/scale
      sigma=sign(vabs(a(k:n,k)),a(k,k))
      a(k,k)=a(k,k)+sigma
      c(k)=sigma*a(k,k)
   end if
end do
END SUBROUTINE qrdcmp
Chapter B2. Solution of Linear Algebraic Equations

d(k)=scale*sigma
a(k:n,k+1:n)=a(k:n,k+1:n)-outerprod(a(k:n,k),&matmul(a(k:n,k),a(k:n,k+1:n)))/c(k)
end if
end do
d(n)=a(n,n)
if (d(n) == 0.0) sing=.true.
END SUBROUTINE qrdcmp

SUBROUTINE qrsolv(a,c,d,b)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : rsolv
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: c,d
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
Solves the set of n linear equations $A \cdot x = b$. The $n \times n$ matrix $a$ and the $n$-dimensional vectors $c$ and $d$ are input as the output of the routine qrdcmp and are not modified. $b$ is input as the right-hand-side vector of length $n$, and is overwritten with the solution vector on output.
INTEGER(I4B) :: j,n
REAL(SP) :: tau
n=assert_eq((/size(a,1),size(a,2),size(b),size(c),size(d)/),'qrsolv')
do j=1,n-1
   Form $Q^T \cdot b$.
   tau=dot_product(a(j:n,j),b(j:n))/c(j)
   b(j:n)=b(j:n)-tau*a(j:n,j)
end do
call rsolv(a,d,b)
Solve $R \cdot x = Q^T \cdot b$.
END SUBROUTINE qrsolv

SUBROUTINE rsolv(a,d,b)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: d
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
Solves the set of n linear equations $R \cdot x = b$, where $R$ is an upper triangular matrix stored in $a$ and $d$. The $n \times n$ matrix $a$ and the vector $d$ of length $n$ are input as the output of the routine qrdcmp and are not modified. $b$ is input as the right-hand-side vector of length $n$, and is overwritten with the solution vector on output.
INTEGER(I4B) :: i,n
n=assert_eq(size(a,1),size(a,2),size(b),size(d),'rsolv')
do i=n,1,-1
   $b(i)=(b(i)-\text{dot_product}(a(i,i+1:n),b(i+1:n)))/d(i)$
end do
END SUBROUTINE rsolv
SUBROUTINE qrupdt(r,qt,u,v)
USE nrtype; USE nrutil, ONLY : assert_eq,ifirstloc
USE nr, ONLY : rotate,pythag
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: r,qt
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: u
REAL(SP), DIMENSION(:,), INTENT(IN) :: v

Given the QR decomposition of some $n \times n$ matrix, calculates the QR decomposition of the matrix $Q \cdot (R + u \otimes v)$. Here $r$ and $qt$ are $n \times n$ matrices, $u$ and $v$ are $n$-dimensional vectors. Note that $Q^T$ is input and returned in $qt$.

INTEGER(I4B) :: i,k,n
n=assert_eq((/size(r,1),size(r,2),size(qt,1),size(qt,2),size(u),size(v)/),'qrupdt')
k=n+1-ifirstloc(u(n:1:-1) /= 0.0)
if (k < 1) k=1
DO i=k-1,1,-1
   Transform $R + u \otimes v$ to upper Hessenberg.
   call rotate(r,qt,i,u(i),-u(i+1))
   u(i)=pythag(u(i),u(i+1))
END DO
r(1,:)=r(1,:)+u(1)*v
DO i=1,k-1
   Transform upper Hessenberg matrix to upper triangular.
   call rotate(r,qt,i,r(i,i),-r(i+1,i))
END DO
END SUBROUTINE qrupdt

SUBROUTINE rotate(r,qt,i,a,b)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), TARGET, INTENT(INOUT) :: r,qt
INTEGER(I4B), INTENT(IN) :: i
REAL(SP), INTENT(IN) :: a,b

Given $n \times n$ matrices $r$ and $qt$, carry out a Jacobi rotation on rows $i$ and $i+1$ of each matrix.

$a$ and $b$ are the parameters of the rotation: 
\[
\cos \theta = \frac{a}{\sqrt{a^2 + b^2}}, \quad \sin \theta = \frac{b}{\sqrt{a^2 + b^2}}.
\]

REAL(SP), DIMENSION(size(r,1)) :: temp
INTEGER(I4B) :: n
REAL(SP) :: c,fact,s
n=assert_eq(size(r,1),size(r,2),size(qt,1),size(qt,2),'rotate')
if (a == 0.0) then
   Avoid unnecessary overflow or underflow.
   c=0.0
else if (abs(a) > abs(b)) then
   fact=b/a
   c=sign(1.0_sp/sqrt(1.0_sp+fact**2),a)
   s=fact*c
else
   fact=a/b
   s=sign(1.0_sp/sqrt(1.0_sp+fact**2),b)
   c=fact*s
end if

temp(i:n)=r(i,i:n)  \ Premultiply $r$ by Jacobi rotation.
r(i,i:n)=c*temp(i:n)-s*r(i+1,i:n)
r(i+1,i:n)=s*temp(i:n)+c*r(i+1,i:n)
temp=qt(i,:)
qt(i,:)=c*temp-s*qt(i+1,:)
qt(i+1,:)=s*temp+c*qt(i+1,:)
END SUBROUTINE rotate

The function ifirstloc in nrutil returns the first occurrence of .true. in a logical vector. See the discussion of the analogous routine imaxloc on p. 1017.
CITED REFERENCES AND FURTHER READING:
Chapter B3. Interpolation and Extrapolation

SUBROUTINE polint(xa,ya,x,y,dy)
USE nttype; USE nrtutil, ONLY : assert_eq,iminloc,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: y,dy
Given arrays xa and ya of length N, and given a value x, this routine returns a value y,
and an error estimate dy. If \( P(x) \) is the polynomial of degree \( N - 1 \) such that \( P(x_a) = y_a, \ i = 1, \ldots, N \), then the returned value \( y = P(x) \).
INTEGER(I4B) :: m,n,ns
REAL(SP), DIMENSION(size(xa)) :: c,d,den,ho

\[ n = \text{assert_eq}(\text{size}(xa),\text{size}(ya),'polint') \]

\[ c = y_a \]
\[ d = y_a \]
\[ ho = xa-x \]
\[ ns = \text{iminloc}(|x-xa|) \]
\[ y = y_a(ns) \] This is the initial approximation to y.
\[ ns = ns-1 \]
\[ \text{do } m = 1, n-1 \]
\[ \text{den}(1:n-m) = ho(1:n-m) - ho(1+m:n) \] we loop over the current c’s and d’s and up-
\[ \text{if} \ (\text{any}(\text{den}(1:n-m) == 0.0)) \ & \] date them.
\[ \text{call nrerror('polint: calculation failure')} \]
This error can occur only if two input xa’s are (to within roundoff) identical.
\[ \text{den}(1:n-m) = c(2:n-m+1) - d(1:n-m) \] Here the c’s and d’s are updated.
\[ d(1:n-m) = ho(1+m:n) \ast \text{den}(1:n-m) \]
\[ c(1:n-m) = ho(1:n-m) \ast \text{den}(1:n-m) \]
\[ \text{if} \ (2*ns < n-m) \text{ then } \]
After each column in the tableau is completed, we decide
\[ \text{dy} = c(ns+1) \] which correction, c or d, we want to add to our accu-
\[ \text{else} \]
\[ \text{dy} = d(ns) \] mulating value of y, i.e., which path to take through
\[ \text{ns} = ns-1 \]
the tableau—forking up or down. We do this in such a way as to take the most “straight line” route through the
tableau to its apex, updating ns accordingly to keep track of
where we are. This route keeps the partial approxima-
tions centered (insofar as possible) on the target x. The
last dy added is thus the error indication.

\[ \text{end do} \]
\[ \text{end do} \]
\[ \text{END SUBROUTINE polint} \]

SUBROUTINE ratint(xa,ya,x,y,dy)
USE nttype; USE nrtutil, ONLY : assert_eq,iminloc,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: y,dy
Given arrays xa and ya of length N, and given a value x, this routine returns a value y,
and an error estimate dy. The value returned is that of the diagonal rational function,
evaluated at x, that passes through the N points \((x_a, y_a), i = 1 \ldots N\).
INTEGER(I4B) :: m,n,ns
REAL(SP), DIMENSION(size(xa)) :: c,d,dd,h,t
REAL(SP), PARAMETER :: TINY=1.0e-25_sp
A small number.
n=assert_eq(size(xa),size(ya),'ratint')
h=xa-x
ns=minloc(abs(h))
y=ya(ns)
if (x == xa(ns)) then
dy=0.0
RETURN
end if

The TINY part is needed to prevent
a rare zero-over-zero condition.
c=ya
d=ya+TINY
ns=ns-1
do m=1,n-1
t(1:n-m)=(xa(1:n-m)-x)*d(1:n-m)/h(1+m:n)

h will never be zero, since this was

tested in the initializing loop.

This error condition indicates that
the interpolating function has a
pole at the requested value of x.

dd(1:n-m)=t(1:n-m)-c(2:n-m+1)
d(1:n-m)=c(2:n-m+1)*dd(1:n-m)
c(1:n-m)=t(1:n-m)*dd(1:n-m)
if (2*ns < n-m) then

dy=c(ns+1)
else

dy=d(ns)

end if

end do

END SUBROUTINE ratint

⋆⋆⋆

SUBROUTINE spline(x,y,yp1,ypn,y2)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : tridag
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(IN) :: yp1,ypn
REAL(SP), DIMENSION(:), INTENT(OUT) :: y2

Given arrays x and y of length N containing a tabulated function, i.e.,
yi = f(xi), with x1 < x2 < ... < xN, and given values yp1 and yp2 for the first derivative of the interpolating
function at points 1 and N, respectively, this routine returns an array y2 of length N
that contains the second derivatives of the interpolating function at the tabulated points
xi. If yp1 and/or yp2 are equal to \(1 \times 10^{30}\) or larger, the routine is signaled to set the

corresponding boundary condition for a natural spline, with zero second derivative on that
boundary.

INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(x)) :: a,b,c,r

n=assert_eq(size(x),size(y),size(y2),'spline')
c(1:n-1)=x(2:n)-x(1:n-1)
r(1:n-1)=6.0_sp*((y(2:n)-y(1:n-1))/c(1:n-1))
r(2:n-1)=r(2:n-1)-r(1:n-2)
a(2:n-1)=c(1:n-2)
b(2:n-1)=2.0_sp*(c(2:n-1)+a(2:n-1))
b(1)=1.0
b(n)=1.0
if (yp1 > 0.99e30_sp) then
The lower boundary condition is set either to be “nat-
ural”
r(1)=0.0
else

or else to have a specified first derivative.

r(1)=(3.0_sp/(x(2)-x(1)))*(y(2)-y(1))/(x(2)-x(1))-yp1)

end do

end do
The upper boundary condition is set either to be "natural"
if \( \text{ypn} > 0.99\times 10^3 \) then
\[
\begin{align*}
r(n) &= 0.0 \\
a(n) &= 0.0
\end{align*}
\]
else
\[
\begin{align*}
r(n) &= \frac{-3.0 \times 10^3}{x(n) - x(n-1)} \left( y(n) - y(n-1) \right) \\
a(n) &= 0.5
\end{align*}
\]
end if

```
c(1)=0.5
end if
```

```fortran
FUNCTION splint(xa,ya,y2a,x)
USE nrtype; USE nrutil, ONLY : assert_eq, nrerror
USE nr, ONLY: locate
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya,y2a
REAL(SP), INTENT(IN) :: x
REAL(SP) :: splint
INTEGER(I4B) :: khi,klo,n
REAL(SP) :: a,b,h
n=assert_eq(size(xa),size(ya),size(y2a),’splint’)
klo=max(min(locate(xa,x),n-1),1)

We will find the right place in the table by means of locate’s bisection algorithm. This is optimal if sequential calls to this routine are at random values of \( x \). If sequential calls are in order, and closely spaced, one would do better to store previous values of klo and khi and test if they remain appropriate on the next call. We have

\[
\begin{align*}
khi &= klo + 1 \\
klo \text{ and khi now bracket the input value of } x.
\end{align*}
\]

\[
\begin{align*}
h &= xa(khi) - xa(klo)
\end{align*}
\]

if \( h == 0.0 \) call nrerror(’bad xa input in splint’) \quad The xa’s must be distinct.
\[
\begin{align*}
a &= (xa(khi) - x) / h \\
b &= (xa(xa(klo)) + y2a(khi) + (a**3-a)*y2a(klo) + (b**3-b)*y2a(khi)*h**2)/6.0
\end{align*}
\]

END FUNCTION splint
```
n = size(xx)
ascnd = (xx(n) >= xx(1))
jl = 0
ju = n+1
do
  if (ju - jl <= 1) exit
  jm = (ju + jl) / 2
  if (ascnd .eqv. (x >= xx(jm))) then
    jl = jm
  else
    ju = jm
  end if
end do
if (x == xx(1)) then
  locate = 1
else if (x == xx(n)) then
  locate = n-1
else
  locate = jl
end if
END FUNCTION locate

The use of bisection is perhaps a sin on a genuinely parallel machine, but
(since the process takes only logarithmically many sequential steps) it is at
most a small sin. One can imagine a “fully parallel” implementation like,

k = iminloc(abs(x-xx))
if ((x < xx(k)) .equiv. (xx(1) < xx(n))) then
  locate = k-1
else
  locate = k
end if

Problem is, unless the number of physical (not logical) processors participating in
the iminloc is larger than N, the length of the array, this “parallel” code turns a
log N algorithm into one scaling as N, quite an unacceptable inefficiency. So we
prefer to be small sinners and bisect.

SUBROUTINE hunt(xx,x,jlo)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: jlo
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: xx

Given an array xx(1:N), and given a value x, returns a value jlo such that
x is between xx(jlo) and xx(jlo+1). xx must be monotonic, either increasing or decreasing. jlo = 0
or jlo = N is returned to indicate that x is out of range. jlo on input is taken as the
initial guess for jlo on output.

INTEGER(I4B) :: n,inc,jhi,jm
LOGICAL :: ascnd
n = size(xx)
ascnd = (xx(n) >= xx(1))
if (jl0 .or. jlo > n) then
  jlo = 0
else
  inc = 1
  if (x >= xx(jlo) .equiv. ascnd) then
    Hunt up:
    do
      jhi = jlo + inc
      if (jhi > n) then
        Done hunting, since off end of table.
      else
        if (x < xx(jhi)) then
          jlo = jhi
        else
          jm = (jlo + jhi) / 2
          if (ascnd .equiv. (x >= xx(jm))) then
            jl = jm
          else
            ju = jm
          end if
          do
            if (ju - jl <= 1) exit
            jm = (ju + jl) / 2
            if (ascnd .equiv. (x >= xx(jm))) then
              jl = jm
            else
              ju = jm
            end if
          end do
          if (x == xx(1)) then
            locate = 1
          else if (x == xx(n)) then
            locate = n-1
          else
            locate = jl
          end if
        end do
      end do
    end if
  else
    Hunt down:
    do
      jlo = jhi - inc
      if (jlo < 0) then
        jlo = 0
      else
        jm = (jlo + jhi) / 2
        if (ascnd .equiv. (x < xx(jm))) then
          jl = jm
        else
          ju = jm
        end if
        do
          if (ju - jl <= 1) exit
          jm = (ju + jl) / 2
          if (ascnd .equiv. (x < xx(jm))) then
            jl = jm
          else
            ju = jm
          end if
        end do
        if (x == xx(n)) then
          locate = n-1
        else
          locate = jl
        end if
      end do
    end do
  end if
end if
END SUBROUTINE hunt
Chapter B3. Interpolation and Extrapolation

1047

\[ jhi = n + 1 \]

\[ \text{exit} \]

else

\[ \text{if } (x < xx(jhi) \text{ .eqv. ascnd) exit} \]

\[ jlo = jhi \]

\[ \text{inc = inc} \times \text{inc} \]

\[ \text{end if} \]

\[ \text{end do} \]

and try again.

else

\[ \text{Hunt down:} \]

\[ jhi = jlo \]

\[ \text{do} \]

\[ jlo = jhi - \text{inc} \]

\[ \text{if } (jlo < 1) \text{ then} \]

\[ jlo = 0 \]

\[ \text{exit} \]

else

\[ \text{if } (x \geq xx(jlo) \text{ .eqv. ascnd) exit} \]

\[ jhi = jlo \]

\[ \text{inc = inc} \times \text{inc} \]

\[ \text{end if} \]

\[ \text{end do} \]

and try again.

end if

end if

Done hunting, value bracketed.

\[ \text{do} \]

Hunt is done, so begin the final bisection phase:

\[ \text{if } (jhi - jlo \leq 1) \text{ then} \]

\[ \text{if } (x = xx(n)) jlo = n - 1 \]

\[ \text{if } (x = xx(1)) jlo = 1 \]

\[ \text{exit} \]

else

\[ jm = \frac{jhi + jlo}{2} \]

\[ \text{if } (x \geq xx(jm) \text{ .eqv. ascnd}) \text{ then} \]

\[ jlo = jm \]

else

\[ jhi = jm \]

end if

end if

end do

END SUBROUTINE hunt

⋆⋆⋆

FUNCTION polcoe(x,y)

USE nrtype; USE nrutil, ONLY : assert_eq, outerdiff

IMPLICIT NONE

REAL(SP), DIMENSION(:,), INTENT(IN) :: x, y

REAL(SP), DIMENSION(size(x)) :: polcoe

Given same-size arrays \( x \) and \( y \) containing a tabulated function \( y_i = f(x_i) \), this routine returns a same-size array of coefficients \( c_j \), such that \( y_i = \sum_j c_j x_i^{j-1} \).

INTEGER(I4B) :: i, k, n

REAL(SP), DIMENSION(size(x)) :: s

REAL(SP), DIMENSION(size(x), size(x)) :: a

n = assert_eq(size(x), size(y), 'polcoe')

s=0.0

s(n)=x(1)

do i=2, n

s(n+1-i:n-1)=s(n+1-i:n-1)-x(i)*s(n+2-i:n)

s(n)=s(n)-x(i)

end do

a=outerdiff(x,x) \quad \text{Make vector } w_j = \prod_{j \neq k} (x_j - x_k), \text{ using polcoe for temporary storage.}

polcoe=product(a, dim=2, mask=a /= 0.0)
Now do synthetic division by \( x - x_j \). The division for all \( x_j \) can be done in parallel (on a parallel machine), since the : in the loop below is over \( j \).

\[
a(:,1) = \frac{s(1)}{x(:)}
\]

\[
do \quad k=2,n
\]

\[
a(:,k) = \frac{s(k) - a(:,k-1)}{x(:)}
\]

\[
end do
\]

\[
s = y/polcoe
\]

\[
polcoe = \text{matmul}(s,a)
\]

Solve linear system.

END FUNCTION polcoe

For a description of the coding here, see §22.3, especially equation (22.3.9). You might also want to compare the coding here with the Fortran 77 version, and also look at the description of the method on p. 84 in Volume 1. The Fortran 90 implementation here is in fact much closer to that description than is the Fortran 77 method, which goes through some acrobatics to roll the synthetic division and matrix multiplication into a single set of two nested loops. The price we pay, here, is storage for the matrix \( a \). Since the degree of any useful polynomial is not a very large number, this is essentially no penalty.

Also worth noting is the way that parallelism is brought to the required synthetic division. For a single such synthetic division (e.g., as accomplished by the nrutil routine poly_term), parallelism can be obtained only by recursion. Here things are much simpler, because we need a whole bunch of simultaneous and independent synthetic divisions; so we can just do them in the obvious, data-parallel, way.

FUNCTION polcof(xa,ya)
USE nrtype; USE nrutil, ONLY : assert_eq,iminloc
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
REAL(SP), DIMENSION(size(xa)) :: polcof
Given same-size arrays \( xa \) and \( ya \) containing a tabulated function \( ya_i = f(xa_i) \), this routine returns a same-size array of coefficients \( c_j \) such that \( ya_i = \sum_j c_j xa_i^{j-1} \).
INTEGER(I4B) :: j,k,m,n
REAL(SP) :: dy
REAL(SP), DIMENSION(size(xa)) :: x,y
n=assert_eq(size(xa),size(ya),'polcof')
x=xa
y=ya
do j=1,n
m=n+1-j
    call polint(x(1:m),y(1:m),0.0_sp,polcof(j),dy)
    Use the polynomial interpolation routine of §3.1 to extrapolate to \( x = 0 \).
    k=iminloc(abs(x(1:m)))
    Find the remaining \( x_k \) of smallest absolute value, where \( x(1:m) /= 0.0 \).
    y(1:m)=y(1:m)-polcof(j)/x(1:m)
    reduce all the terms,
    y(k:m-1)=y(k+1:m)
    and eliminate \( x_k \).
    x(k:m-1)=x(k+1:m)
end do
END FUNCTION polcof

* * *

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. For a description of the coding here, see §22.3, especially equation (22.3.9). You might also want to compare the coding here with the Fortran 77 version, and also look at the description of the method on p. 84 in Volume 1. The Fortran 90 implementation here is in fact much closer to that description than is the Fortran 77 method, which goes through some acrobatics to roll the synthetic division and matrix multiplication into a single set of two nested loops. The price we pay, here, is storage for the matrix \( a \). Since the degree of any useful polynomial is not a very large number, this is essentially no penalty.

Also worth noting is the way that parallelism is brought to the required synthetic division. For a single such synthetic division (e.g., as accomplished by the nrutil routine poly_term), parallelism can be obtained only by recursion. Here things are much simpler, because we need a whole bunch of simultaneous and independent synthetic divisions; so we can just do them in the obvious, data-parallel, way.

FUNCTION polcof(xa,ya)
USE nrtype; USE nrutil, ONLY : assert_eq,iminloc
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
REAL(SP), DIMENSION(size(xa)) :: polcof
Given same-size arrays \( xa \) and \( ya \) containing a tabulated function \( ya_i = f(xa_i) \), this routine returns a same-size array of coefficients \( c_j \) such that \( ya_i = \sum_j c_j xa_i^{j-1} \).
INTEGER(I4B) :: j,k,m,n
REAL(SP) :: dy
REAL(SP), DIMENSION(size(xa)) :: x,y
n=assert_eq(size(xa),size(ya),'polcof')
x=xa
y=ya
do j=1,n
m=n+1-j
    call polint(x(1:m),y(1:m),0.0_sp,polcof(j),dy)
    Use the polynomial interpolation routine of §3.1 to extrapolate to \( x = 0 \).
    k=iminloc(abs(x(1:m)))
    Find the remaining \( x_k \) of smallest absolute value, where \( x(1:m) /= 0.0 \).
    y(1:m)=y(1:m)-polcof(j)/x(1:m)
    reduce all the terms,
    y(k:m-1)=y(k+1:m)
    and eliminate \( x_k \).
    x(k:m-1)=x(k+1:m)
end do
END FUNCTION polcof

* * *
SUBROUTINE polin2(x1a,x2a,ya,x1,x2,y,dy)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: x1a,x2a
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: ya
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), INTENT(OUT) :: y,dy

Given arrays x1a of length M and x2a of length N of independent variables, and an M × N array of function values ya, tabulated at the grid points defined by x1a and x2a, and given values x1 and x2 of the independent variables, this routine returns an interpolated function value y, and an accuracy indication dy (based only on the interpolation in the x1 direction, however).

INTEGER(I4B) :: j,m,ndum
REAL(SP), DIMENSION(size(x1a)) :: ymtmp
REAL(SP), DIMENSION(size(x2a)) :: yntmp
m=assert_eq(size(x1a),size(ya,1),'polin2: m')
ndum=assert_eq(size(x2a),size(ya,2),'polin2: ndum')
do j=1,m
  yntmp=ya(j,:)
  call polint(x2a,yntmp,x2,ymtmp(j),dy)
end do
call polint(x1a,ymtmp,x1,y,dy)
END SUBROUTINE polin2

⋆⋆⋆

SUBROUTINE bcucof(y,y1,y2,y12,d1,d2,c)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: d1,d2
REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
REAL(SP), DIMENSION(4,4), INTENT(OUT) :: c

Given arrays y, y1, y2, and y12, each of length 4, containing the function, gradients, and cross derivative at the four grid points of a rectangular grid cell (numbered counterclockwise from the lower left), and given d1 and d2, the length of the grid cell in the 1- and 2-directions, this routine returns the 4 × 4 table c that is used by routine bcuint for bicubic interpolation.

REAL(SP), DIMENSION(16) :: x
REAL(SP), DIMENSION(16,16) :: wt
DATA wt /1,0,-3,2,4*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,
  -6,2,0,-6,4,2*0,-2,6,0,-9,6,
  -6,4,0,-1,0,-3,2,8*0,-1,0,3,-2,1,0,-3,6,
  -3,2,0,-2,0,2,-4,2*0,-2,0,2,-4,2*0,-2,0,1,-2,1,5*0,
  -3,2,0,-2,0,2,-4,2*0,-2,0,2,-4,2*0,-2,0,1,-2,1,5*0,
  -2,0,1,-2,1,5*0,1,0,-3,2,8*0,-1,0,3,-2,1,0,-3,6,
  -6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6,2*0,-3,0,9,-6,2,0,-6,4,2*0,-2,6,0,-9,6

x=(1:4)=y
x(5:8)=y1*d1
x(9:12)=y2*d2
x(13:16)=y12*d1*d2
x=reshape(x,(4,4,:),order=(/2,1/))
c=matmul(wt,x)
c=reshape(c,(/4,4/),order=(/2,1/))
END SUBROUTINE bcucof
It is a powerful technique to combine the `matmul` intrinsic with `reshape`'s of the input or output. The idea is to use `matmul` whenever the calculation can be cast into the form of a linear mapping between input and output objects. Here the `order=(/2,1/)` parameter specifies that we want the packing to be by rows, not by Fortran's default of columns. (In this two-dimensional case, it's the equivalent of applying `transpose`.)

```fortran
x = matmul(wt, x) ... c = reshape(x, (/4,4/), order=(/2,1/))
```

SUBROUTINE `bcuint(y,y1,y2,y12,x1l,x1u,x2l,x2u,x1,x2,ansy,ansy1,ansy2)`

```fortran
SUBROUTINE bcuint(y,y1,y2,y12,x1l,x1u,x2l,x2u,x1,x2,ansy,ansy1,ansy2) USE nrtype; USE nrutil, ONLY : nrerror USE nr, ONLY : bcucof IMPLICIT NONE REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12 REAL(SP), INTENT(IN) :: x1l,x1u,x2l,x2u,x1,x2 REAL(SP), INTENT(OUT) :: ansy,ansy1,ansy2 ! Bicubic interpolation within a grid square. Input quantities are y,y1,y2,y12 (as described in bcucof): x1l and x1u, the lower and upper coordinates of the grid square in the 1-direction; x2l and x2u likewise for the 2-direction; and x1,x2, the coordinates of the desired point for the interpolation. The interpolated function value is returned as ansy, and the interpolated gradient values as ansy1 and ansy2. This routine calls bcucof.

```fortran
INTEGER(I4B) :: i
REAL(SP) :: t,u
REAL(SP), DIMENSION(4,4) :: c
```

```fortran
call bcucof(y,y1,y2,y12,x1u-x1l,x2u-x2l,c) ! Get the c's.
```

```fortran
if (x1u == x1l .or. x2u == x2l) call & nrerror('bcuint: problem with input values - boundary pair equal?')
```

```fortran
t=(x1-x1l)/(x1u-x1l)
```

```fortran
u=(x2-x2l)/(x2u-x2l)
```

```fortran
ansy=0.0
ansy2=0.0
ansy1=0.0
```

```fortran
do i=4,1,-1
    Equation (3.6.6).
    ansy=t*ansy+((c(i,4)*u+c(i,3))*u+c(i,2))*u+c(i,1)
    ansy2=t*ansy2+(3.0_sp*c(i,4)*u+2.0_sp*c(i,3))*u+c(i,2)
    ansy1=u*ansy1+(3.0_sp*c(4,i)*t+2.0_sp*c(3,i))*t+c(2,i)
end do
```

```fortran
ansy1=ansy1/(x1u-x1l)
ansy2=ansy2/(x2u-x2l)
END SUBROUTINE bcuint
```

***

SUBROUTINE `splie2(x1a,x2a,ya,y2a)`

```fortran
SUBROUTINE splie2(x1a,x2a,ya,y2a) USE nrtype; USE nrutil, ONLY : assert_eq USE nr, ONLY : spline IMPLICIT NONE REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya REAL(SP), DIMENSION(:,:), INTENT(OUT) :: y2a ! Given an M \times N tabulated function ya, and N tabulated independent variables x2a, this routine constructs one-dimensional natural cubic splines of the rows of ya and returns the second derivatives in the M \times N array y2a. (The array x1a is included in the argument list merely for consistency with routine splin2.)

INTEGER(I4B) :: j,m,ndum m=assert_eq(size(x1a),size(ya,1),size(y2a,1),'splie2: m') ndum=assert_eq(size(x2a),size(ya,2),size(y2a,2),'splie2: ndum') do j=1,m
    call spline(x2a,ya(j,:),1.0e30_sp,1.0e30_sp,y2a(j,:))
end do
END SUBROUTINE splie2
```
Values $1 \times 10^{10}$ signal a natural spline.

FUNCTION splin2(x1a,x2a,ya,y2a,x1,x2)
USE nrtype, USE nrutil, ONLY : assert_eq
USE nr, ONLY : spline, splint
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: ya,y2a
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP) :: splin2

Given $x_{1a}$, $x_{2a}$, $y_a$ as described in splie2 and $y_{2a}$ as produced by that routine; and given
a desired interpolating point $x_1, x_2$; this routine returns an interpolated function value by
bicubic spline interpolation.

INTEGER(I4B) :: j,m,ndum
REAL(SP), DIMENSION(size(x1a)) :: yytmp,y2tmp2
m=assert_eq(size(x1a),size(ya,1),size(y2a,1),'splin2: m')
ndum=assert_eq(size(x2a),size(ya,2),size(y2a,2),'splin2: ndum')
do j=1,m
    yytmp(j)=splint(x2a,ya(j,:),y2a(j,:),x2)
end do

Perform $m$ evaluations of the row splines constructed by splie2, using the one-dimensional
spline evaluator splint.

END FUNCTION splin2
Chapter B4. Integration of Functions

SUBROUTINE trapzd(func,a,b,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE

This routine computes the \( n \)th stage of refinement of an extended trapezoidal rule. \( \text{func} \) is input as the name of the function to be integrated between limits \( a \) and \( b \), also input. When called with \( n=1 \), the routine returns as \( s \) the crudest estimate of \( \int_{a}^{b} f(x) \, dx \). Subsequent calls with \( n=2,3,... \) (in that sequential order) will improve the accuracy of \( s \) by adding \( 2^{n-2} \) additional interior points. \( s \) should not be modified between sequential calls.

REAL(SP) :: del,fsum
INTEGER(I4B) :: it
if (n == 1) then
    s=0.5_sp*(b-a)*sum(func( (/ a,b /) ))
else
    it=2**(n-2)
    del=(b-a)/it
    fsum=sum(func(arth(a+0.5_sp*del,del,it)))
    s=0.5_sp*(s+del*fsum)
end if
END SUBROUTINE trapzd

While most of the quadrature routines in this chapter are coded as functions, trapzd is a subroutine because the argument \( s \) that returns the function value must also be supplied as an input parameter. We could change the subroutine into a function by declaring \( s \) to be a local variable with the SAVE attribute. However, this would prevent \( s \) from being able to use the routine recursively to do multidimensional quadrature (see quad3d on p. 1065). When \( s \) is left as an argument, a fresh copy is created on each recursive call. As a SAVE’d variable, by contrast, its value would get overwritten on each call, and the code would not properly “re-entrant.”

\[ s=0.5_sp*(b-a)*\text{sum(func( }/ \ a,b /) ) \]  
Note how we use the \( (/.../ \) construct to supply a set of scalar arguments to a vector function.

* * *
FUNCTION qtrap(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : trapzd
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qtrap
INTERFACE
  FUNCTION func(x)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: func
  END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20
REAL(SP), PARAMETER :: EPS=1.0e-6_sp

Returns the integral of the function func from a to b. The parameter EPS should be set to
the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
allowed number of steps. Integration is performed by the trapezoidal rule.

REAL(SP) :: olds
INTEGER(I4B) :: j
olds=0.0
Initial value of olds is arbitrary.
do j=1,JMAX
  call trapzd(func,a,b,qtrap,j)
  if (j > 5) then
    Avoid spurious early convergence.
    if (abs(qtrap-olds) < EPS*abs(olds) .or. &
      (qtrap == 0.0 .and. olds == 0.0)) RETURN
  end if
  olds=qtrap
end do

call nrerror('qtrap: too many steps')
END FUNCTION qtrap

* * *

FUNCTION qsimp(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : trapzd
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qsimp
INTERFACE
  FUNCTION func(x)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: func
  END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20
REAL(SP), PARAMETER :: EPS=1.0e-6_sp

Returns the integral of the function func from a to b. The parameter EPS should be set to
the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
allowed number of steps. Integration is performed by Simpson’s rule.

INTEGER(I4B) :: j
REAL(SP) :: os,ost,st
ost=0.0
os= 0.0
do j=1,JMAX
  call trapzd(func,a,b,ost,qtrap)
  qsimp=(4.0_sp*ost-st)/3.0_sp
  Compare equation (4.2.4).
  if (j > 5) then
    Avoid spurious early convergence.
    if (abs(qsimp-os) < EPS*abs(os) .or. &
      (qsimp == 0.0 .and. os == 0.0)) RETURN
  end if
  os=qsimp
end if
os=qsimp
FUNCTION qromb(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : polint, trapzd
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qromb
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20, JMAXP=JMAX+1, K=5, KM=K-1
REAL(SP), PARAMETER :: EPS=1.0e-6_sp

Returns the integral of the function \( \text{func} \) from \( a \) to \( b \). Integration is performed by Romberg's method of order \( K \), where, e.g., \( K=2 \) is Simpson's rule.

Parameters:
- \( \text{EPS} \) is the fractional accuracy desired, as determined by the extrapolation error estimate;
- \( \text{JMAX} \) limits the total number of steps;
- \( K \) is the number of points used in the extrapolation.

REAL(SP), DIMENSION(JMAXP) :: h,s
These store the successive trapezoidal approximations and their relative stepsizes.

INTEGER(I4B) :: j
h(1)=1.0

do j=1,JMAX
    call trapzd(func,a,b,s(j),j)
    if (j >= K) then
        call polint(h(j-KM:j),s(j-KM:j),0.0_sp,qromb,dqromb)
        if (abs(dqromb) <= EPS*abs(qromb)) RETURN
    end if

    s(j+1)=s(j)
    h(j+1)=0.25_sp*h(j)
end do

call nrerror('qromb: too many steps')
END FUNCTION qromb

SUBROUTINE midpnt(func,a,b,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE

This routine computes the \( n \)th stage of refinement of an extended midpoint rule. \( \text{func} \) is input as the name of the function to be integrated between limits \( a \) and \( b \), also input.
called with $n=1$, the routine returns as $s$ the crudest estimate of $\int_{a}^{b} f(x) \, dx$. Subsequent calls with $n=2,3,\ldots$ (in that sequential order) will improve the accuracy of $s$ by adding $(2/3) \times 3^{n-1}$ additional interior points. $s$ should not be modified between sequential calls.

```fortran
REAL(SP) :: del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
if (n == 1) then
  s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
  it=3**(n-2)
  del=(b-a)/(3.0_sp*it)  
  x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,1.0_sp*del,2*del,
                  x(2:2*it-1:2)+2.0_sp*del
  s=s/3.0_sp+del*sum(func(x))
end if
END SUBROUTINE midpnt
```

midpnt is a subroutine and not a function for the same reasons as trapzd. This is also true for the other mid... routines below.

$s=(b-a)*\sum( (/0.5_sp*(a+b)/) )$  Here we use (/.../) to pass a single scalar argument to a vector function.

```fortran
FUNCTION qromo(func,a,b,choose)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qromo
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
SUBROUTINE choose(funk,aa,bb,s,n)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
FUNCTION funk(x)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: funk
END FUNCTION funk
END INTERFACE
END SUBROUTINE choose
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=14,JMAXP=JMAX+1,K=5,KM=K-1
REAL(SP), PARAMETER :: EPS=1.0e-6
Romberg integration on an open interval. Returns the integral of the function func from a to b, using any specified integrating subroutine choose and Romberg's method. Normally choose will be an open formula, not evaluating the function at the endpoints. It is assumed that choose triples the number of steps on each call, and that its error series contains only
```
even powers of the number of steps. The routines midpnt, midinf, midsq, midsq, and midexp are possible choices for choose. The parameters have the same meaning as in qromo.

```fortran
REAL(SP), DIMENSION(JMAXP) :: h, s
REAL(SP) :: dqromo
INTEGER(I4B) :: j
h(1)=1.0
do j=1,JMAX
  call choose(func,a,b,s(j),j)
  if (j >= K) then
    call polint(h(j-KM:j),s(j-KM:j),0.0_sp,qromo,dqromo)
    if (abs(dqromo) <= EPS*abs(qromo)) RETURN
  end if
  s(j+1)=s(j)
  h(j+1)=h(j)/9.0_sp
  This is where the assumption of step tripling and an even
end do
error series is used.
END FUNCTION qromo
```

```fortran
SUBROUTINE midinf(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth, assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa, bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
FUNCTION funk(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: funk
END FUNCTION funk
END INTERFACE
This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement
of the integral of funk from aa to bb, except that the function is evaluated at evenly spaced
points in 1/x rather than in x. This allows the upper limit bb to be as large and positive
as the computer allows, or the lower limit aa to be as large and negative, but not both.
aa and bb must have the same sign.
REAL(SP) :: a, b, del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
call assert(aa*bb > 0.0, 'midinf args')
b=1.0_sp/aa
These two statements change the limits of integration ac-
ordinarily.
a=1.0_sp/bb
if (n == 1) then
  From this point on, the routine is exactly identical to midpnt.
s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
  it=3***(n-2)
  del=(b-a)/(3.0_sp*it)
x(1:2:it-1)=arth(a+0.5_sp*del,3.0_sp*del, it)
x(2:2:it:2)=x(1:2:it-1)+2.0_sp*del
s=s+3.0_sp*del*sum(func(x))
end if
CONTAINS
FUNCTION func(x) 
This internal function effects the change of variable.
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
func=funk(1.0_sp/x)/x**2
END FUNCTION func
END SUBROUTINE midinf
```
FUNCTION func(x)  The change of variable could have been effected by a statement function in midinf itself. However, the statement function is a Fortran 77 feature that is deprecated in Fortran 90 because it does not allow the benefits of having an explicit interface, i.e., a complete set of specification statements. Statement functions can always be coded as internal subprograms instead.

SUBROUTINE midsql(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
END INTERFACE
This routine is an exact replacement for midpnt, i.e., returns as \( s \) the \( n \)th stage of refinement of the integral of \( \text{funk} \) from \( aa \) to \( bb \), except that it allows for an inverse square-root singularity in the integrand at the lower limit \( aa \).

REAL(SP) :: a,b,del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
b=sqrt(bb-aa)  These two statements change the limits of integration accordingly.
a=0.0
if (n == 1) then
    From this point on, the routine is exactly identical to midpnt.
    \( s=(b-a)\times\text{sum}(\text{func}(\,(/0.5\_sp\times(a+b)/)\,)) \)
else
    \( it=3\times\text{pow}(n-2,2) \)
    \( del=(b-a)/(3.0\_sp\times it) \)
    \( x(1:2\times it-1:2)=\text{arth}(a+0.5\_sp\times del,3.0\_sp\times del,it) \)
    \( x(2:2\times it:2)=x(1:2\times it-1:2)+2.0\_sp\times del \)
    \( s=s/3.0\_sp+del\times\text{sum}(\text{func}(x)) \)
end if
CONTAINS
    FUNCTION func(x)  This internal function effects the change of variable.
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    func=2.0\_sp\times x\times\text{func}(aa+x**2)
END FUNCTION func
END SUBROUTINE midsql

SUBROUTINE midsqu(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
END INTERFACE
This routine is an exact replacement for midpnt, i.e., returns as \( s \) the \( n \)th stage of refinement of the integral of \( \text{funk} \) from \( aa \) to \( bb \), except that it allows for an inverse square-root singularity in the integrand at the upper limit \( bb \).

REAL(SP) :: a,b,del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
b=sqrt(bb-aa)  
These two statements change the limits of integration accordingly.
a=0.0
if (n == 1) then  
From this point on, the routine is exactly identical to midpnt.
s=(b-a)*sum(func( (0.5_sp*(a+b)/ ) ))
else
  it=3**(n-2)
  del=(b-a)/(3.0_sp*it)
  x(1:2:it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)
  x(2:2:it:2)=x(1:2:it-1:2)+2.0_sp*del
  s=s/3.0_sp+del*sum(func(x))
end if
CONTAINS
  FUNCTION func(x)  
  This internal function effects the change of variable.
  REAL(SP), DIMENSION(:), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: func
  func=2.0_sp*x*funk(bb-x**2)
END FUNCTION func
END SUBROUTINE midsqu

SUBROUTINE midsqub(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
  FUNCTION funk(x)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: funk
  END FUNCTION funk
END INTERFACE

This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement of the integral of funk from aa to bb, except that bb is assumed to be infinite (value passed not actually used). It is assumed that the function funk decreases exponentially rapidly at infinity.
REAL(SP) :: a,b,del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
b=exp(-aa)  
These two statements change the limits of integration accordingly.
a=0.0
if (n == 1) then  
From this point on, the routine is exactly identical to midpnt.
s=(b-a)*sum(func( (0.5_sp*(a+b)/ ) ))
else
  it=3**(n-2)
  del=(b-a)/(3.0_sp*it)
  x(1:2:it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)
  x(2:2:it:2)=x(1:2:it-1:2)+2.0_sp*del
  s=s/3.0_sp+del*sum(func(x))
end if
CONTAINS
  FUNCTION func(x)  
  This internal function effects the change of variable.
  REAL(SP), DIMENSION(:), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: func
  func=funk(-log(x))/x
END FUNCTION func
END SUBROUTINE midsqub
SUBROUTINE gauleg(x1,x2,x,w)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), DIMENSION(,:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-14_dp

Given the lower and upper limits of integration x1 and x2, this routine returns arrays x and w
of length N containing the abscissas and weights of the Gauss-Legendre N-point quadrature
formula. The parameter EPS is the relative precision. Note that internal computations are
done in double precision.

INTEGER(I4B) :: its,j,m,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(DP) :: xl,xm
REAL(DP), DIMENSION((size(x)+1)/2) :: p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION((size(x)+1)/2) :: unfinished
n=assert_eq(size(x),size(w),'gauleg')

INTEGER(I4B) :: its,j,m,n

m=(n+1)/2
xm=0.5_dp*(x2+x1)
xl=0.5_dp*(x2-x1)
z=cos(PI_D*(arth(1,1,m)-0.25_dp)/(n+0.5_dp))
unfinished=.true.
do its=1,MAXIT

Newton's method carried out simultane-
ously on the roots.
end where

p1=1.0
p2=0.0
end where

do j=1,n
Loop up the recurrence relation to get
the Legendre polynomials evaluated
at z.

where (unfinished)
p3=p1
p2=p1
p1=((2.0_dp*j-1.0_dp)*z*p2-(j-1.0_dp)*p3)/j
end where

end do

p1 now contains the desired Legendre polynomials. We next compute pp, the derivatives,
by a standard relation involving also p2, the polynomials of one lower order.

where (unfinished)
pp=n*(z*p1-p2)/(z*z-1.0_dp)
z1=z
z=z1-p1/pp
Newton's method.
end where

if (.not. any(unfinished)) exit
end do

if (its == MAXIT+1) call nrerror('too many iterations in gauleg')
x(1:m)=xm-xl*z
x(n:n-m+1:-1)=xm+xl*z
w(1:m)=2.0_dp*xl/((1.0_dp-z**2)*pp**2)
w(n:n-m+1:-1)=w(1:m)
END SUBROUTINE gauleg

Often we have an iterative procedure that has to be applied until all
components of a vector have satisfied a convergence criterion. Some
components of the vector might converge sooner than others, and it is
inefficient on a small-scale parallel (SSP) machine to continue iterating on those
components. The general structure we use for such an iteration is exemplified by
the following lines from gauleg:

LOGICAL(LGT), DIMENSION((size(x)+1)/2) :: unfinished

unfinished= true.
do its=1,MAXIT

...
where (unfinished)
...
  unfinished=(abs(z-z1) > EPS)
end where
if (.not. any(unfinished)) exit
end do
if (its == MAXIT+1) call nrerror('too many iterations in gaulag')

We use the logical mask unfinished to control which vector components are processed inside the where. The mask gets updated on each iteration by testing whether any further vector components have converged. When all have converged, we exit the iteration loop. Finally, we check the value of its to see whether the maximum allowed number of iterations was exceeded before all components converged.

The logical expression controlling the where block (in this case unfinished) gets evaluated completely on entry into the where, and it is then perfectly fine to modify it inside the block. The modification affects only the next execution of the where.

On a strictly serial machine, there is of course some penalty associated with the above scheme: after a vector component converges, its corresponding component in unfinished is redundantly tested on each further iteration, until the slowest-converging component is done. If the number of iterations required does not vary too greatly from component to component, this is a minor, often negligible, penalty. However, one should be on the alert against algorithms whose worst-case convergence could differ from typical convergence by orders of magnitude. For these, one would need to implement a more complicated packing-unpacking scheme. (See discussion in Chapter B6, especially introduction, p. 1083, and notes for factrl, p. 1087.)

SUBROUTINE gaulag(x,w,alf)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: alf
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-13_dp

Given alf, the parameter $\alpha$ of the Laguerre polynomials, this routine returns arrays $x$ and $w$ of length $N$ containing the abscissas and weights of the $N$-point Gauss-Laguerre quadrature formula. The abscissas are returned in ascending order. The parameter EPS is the relative precision. Note that internal computations are done in double precision.

INTEGER(I4B) :: its,j,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(SP), DIMENSION(size(x)) :: rhs,r2,r3,theta
REAL(SP), PARAMETER :: C1=9.084064e-01_sp,C2=5.214976e-02_sp,&
  C3=2.579930e-03_sp,C4=3.986126e-03_sp
REAL(SP), DIMENSION(size(x)) :: p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION(size(x)) :: unfinished

n=assert_eq(size(x),size(w),'gaulag')
anu=4.0_sp*n+2.0_sp*alf+2.0_sp
r3=rh**(1.0_sp/3.0_sp)
theta=r3*(C1+r2*(C2+r2*(C3+r2*C4)))
z=anu*cos(theta)**2
unfinished=.true.
do ite=1,MAXIT
  Newton’s method carried out simultaneously on roots.
  where (unfinished)
  p1=1.0
Loop up the recurrence relation to get the Laguerre polynomials evaluated at $z$.

\[
p_1 = \frac{(2.0_dp\cdot j-1.0_dp+\alpha-z)p_2-(j-1.0_dp+\alpha)p_3}{j}
\]

$p_1$ now contains the desired Laguerre polynomials. We next compute $pp$, the derivatives, by a standard relation involving also $p_2$, the polynomials of one lower order.

\[
\text{Newton's formula.}
\]

\[
pp = \frac{n*p_1-(n+\alpha)*p_2}{z}
\]

\[
z = z_1 - p_1/pp
\]

\[
\text{if (.not. any(unfinished)) exit}
\]

The key difficulty in parallelizing this routine starting from the Fortran 77 version is that the initial guesses for the roots of the Laguerre polynomials were given in terms of previously determined roots. This prevents one from finding all the roots simultaneously. The solution is to come up with a new approximation to the roots that is a simple explicit formula, like the formula we used for the Legendre roots in \texttt{gauleg}.

We start with the approximation to $L_n^\alpha(x)$ given in equation (10.15.8) of [1]. We keep only the first term and ask when it is zero. This gives the following prescription for the $k$th root $x_k$ of $L_n^\alpha(x)$: Solve for $\theta$ the equation

\[
2\theta - \sin 2\theta = \frac{4n - 4k + 3}{4n + 2\alpha + 2}\pi
\]  

(B4.1)

Since $1 \leq k \leq n$ and $\alpha > -1$, we can always find a value such that $0 < \theta < \pi/2$. Then the approximation to the root is

\[
x_k = (4n + 2\alpha + 2)\cos^2 \theta
\]  

(B4.2)

This typically gives 3-digit accuracy, more than enough for the Newton iteration to be able to refine the root. Unfortunately equation (B4.1) is not an explicit formula for $\theta$. (You may recognize it as being of the same form as Kepler’s equation in mechanics.) If we call the right-hand side of (B4.1) $y$, then we can get an explicit formula by working out the power series for $y^{1/3}$ near $\theta = 0$ (using a computer algebra program). Next invert the series to give $\theta$ as a function of $y^{1/3}$. Finally, economize the series (see §5.11). The result is the concise approximation

\[
\theta = 0.9084064y^{1/3} + 5.214976 \times 10^{-2}y + 2.579930 \times 10^{-3}y^{5/3}
\]

\[+ 3.986126 \times 10^{-3}y^{7/3}
\]  

(B4.3)
SUBROUTINE gauher(x,w)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-13_dp,PIM4=0.7511255444649425_dp

This routine returns arrays \( x \) and \( w \) of length \( N \) containing the abscissas and weights of the \( N \)-point Gauss-Hermite quadrature formula. The abscissas are returned in descending order. Note that internal computations are done in double precision.

Parameters: EPS is the relative precision, \( \frac{1}{\pi^{1/4}} \).

INTEGER(I4B) :: its,j,m,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(SP) :: anu
REAL(SP), PARAMETER :: C1=9.084064e-01_sp,C2=5.214976e-02_sp,&
C3=2.579930e-01_sp,C4=3.986126e-03_sp
REAL(SP), DIMENSION((size(x)+1)/2) :: rhs,r2,r3,theta
REAL(DP), DIMENSION((size(x)+1)/2) :: p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION((size(x)+1)/2) :: unfinished
n=assert_eq(size(x),size(w),'gauher')
m=(n+1)/2

The roots are symmetric about the origin, so we have to find only half of them.

rhs=arth(3,4,m)*PI/anu
r3=rhs**(1.0_sp/3.0_sp)
r2=r3**2
theta=r3*(C1+r2*(C2+r2*(C3+r2*C4)))))
z=sqrt(anu)*cos(theta)

Initial approximations to the roots.

unfinished=.true.
do its=1,MAXIT
Newton’s method carried out simultaneously on the roots.
where (unfinished)
p1=PIM4
p2=0.0
end where
do j=1,n
Loop up the recurrence relation to get the Hermite polynomials evaluated at \( z \).
where (unfinished)
p3=p2
p2=p1
p1=z*sqrt(2.0_dp/j)*p2-sqrt(real(j-1,dp)/real(j,dp))*p3
end where
do
p1 now contains the desired Hermite polynomials. We next compute \( pp \), the derivatives, by the relation (4.5.21) using \( p2 \), the polynomials of one lower order.
where (unfinished)
pp=sqrt(2.0_dp*n)*p2
z1=z
z=z1-p1/pp
Newton’s formula.
where (unfinished)
end where
end do
if (its == MAXIT+1) call nrerror('too many iterations in gauher')
x(1:m)=p1
x(n:n-m+1:-1)=-p1 and its symmetric counterpart.
w(1:m)=2.0_dp/pp**2
w(n:n-m+1:-1)=w(1:m) and its symmetric counterpart.
END SUBROUTINE gauher

Once again we need an explicit approximation for the polynomial roots, this time for \( H_n(x) \). We can use the same approximation scheme as for \( L_m^0(x) \), since

\begin{align}
H_{2m}(x) &\propto L_m^{-1/2}(x^2), & H_{2m+1}(x) &\propto x L_m^{1/2}(x^2)
\end{align}

(B4.4)
Equations (B4.1) and (B4.2) become
\[ 2\theta - \sin 2\theta = \frac{4k - 1}{2n + 1} \pi \]
\[ x_k = \sqrt{2n + 1} \cos \theta \]

Here \( k = 1, 2, \ldots, m \) where \( m = [(n + 1)/2] \), and \( k = 1 \) is the largest root. The negative roots follow from symmetry. The root at \( x = 0 \) for odd \( n \) is included in this approximation.

```fortran
SUBROUTINE gaujac(x,w,alf,bet)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: alf,bet
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-14_dp

Given \( \alpha \) and \( \beta \), the parameters \( \alpha \) and \( \beta \) of the Jacobi polynomials, this routine returns arrays \( x \) and \( w \) of length \( N \) containing the abscissas and weights of the \( N \)-point Gauss-Jacobi quadrature formula. The abscissas are returned in descending order. The parameter \( \text{EPS} \) is the relative precision. Note that internal computations are done in double precision.

INTEGER(I4B) :: its,j,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(DP) :: alfbet,a,c,temp
REAL(DP), DIMENSION(size(x)) :: b,p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION(size(x)) :: unfinished
n=assert_eq(size(x),size(w),'gaujac')

Initial approximations to the roots go into \( z \).
\[ z = \cos(\pi(\text{arth}(1,1,n)-0.25_dp+0.5_dp*\alpha)/(n+0.5_dp*(\alpha+1.0_dp))) \]

Newton’s method carried out simultaneously on the roots.
\[ \text{temp}=2.0_dp+\alpha \beta \]
\[ \text{where (unfinished)} \]
\[ \text{p1}=(\alpha-\beta+\text{temp}z)/2.0_dp \]
\[ \text{p2}=1.0 \]
\[ \text{end where} \]
\[ \text{do j}=2,n \]
\[ \text{Loop up the recurrence relation to get the Jacobi polynomials evaluated at } z. \]
\[ a=2*j*(j+\alpha+\beta) \]
\[ \text{temp}=(j-0.0_dp+\alpha)*b*p2-c*p3)/a \]
\[ \text{end where} \]
\[ \text{p1 now contains the desired Jacobi polynomials. We next compute pp, the derivatives, by a standard relation involving also p2, the polynomials of one lower order.} \]
\[ \text{where (unfinished)} \]
\[ \text{pp}=(n*(\alpha-\beta-temp*z)+p1+2.0_dp*(n+0.5_dp*(\alpha+1.0_dp)-p2)/(temp*(1.0_dp-z*z)) \]
\[ \text{Newton’s formula.} \]
\[ \text{unfinished}=(abs(z-z1) > EPS) \]
\[ \text{end where} \]
\[ \text{if (.not. any(unfinished)) exit} \]
\[ \text{end do} \]
\[ \text{if (its == MAXIT+1) call nrerror(’too many iterations in gaujac’) } \]
\[ x=z \]
Store the root and the weight.
```
Now we need an explicit approximation for the roots of the Jacobi polynomials $P_n^{(\alpha,\beta)}(x)$. We start with the asymptotic expansion (10.14.10) of \cite{1}. Setting this to zero gives the formula
\begin{equation}
x = \cos \left[ \frac{k - 1/4 + \alpha/2}{n + (\alpha + \beta + 1)/2} \pi \right]
\end{equation}

This is better than the formula (22.16.1) in \cite{2}, especially at small and moderate $n$.

\begin{verbatim}
SUBROUTINE gaucof(a,b,amu0,x,w)
USE nrtype; USE nrutil, ONLY : assert_eq,unit_matrix
USE nr, ONLY : eigsrt,tqli
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
REAL(SP), INTENT(IN) :: amu0
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
Computes the abscissas and weights for a Gaussian quadrature formula from the Jacobi matrix. On input, $a$ and $b$ of length $N$ are the coefficients of the recurrence relation for the set of monic orthogonal polynomials. The quantity $\mu_0 \equiv \int_a^b W(x) \, dx$ is input as amu0. The abscissas are returned in descending order in array $x$ of length $N$, with the corresponding weights in $w$, also of length $N$. The arrays a and b are modified. Execution can be speeded up by modifying tqli and eigsrt to compute only the first component of each eigenvector.

REAL(SP), DIMENSION(size(a),size(a)) :: z
INTEGER(I4B) :: n
n=assert_eq(size(a),size(b),size(x),size(w),'gaucof')
b(2:n)=sqrt(b(2:n))
call unit_matrix(z)
call tqli(a,b,z)
call eigsrt(a,z)
x=a
w=amu0*z(1,:)**2
Equation (4.5.27).
END SUBROUTINE gaucof
\end{verbatim}
relation for the chosen basis of orthogonal polynomials. The \( 2N \) modified moments \( \nu_j \) are input in \( \text{anu} \) for \( j = 0, \ldots, 2N - 1 \). The first \( N \) coefficients are returned in \( a \) and \( b \).

\[
\begin{align*}
\text{INTEGER}(4B) : & \ k, n, \text{ndum} \\
\text{REAL(SP), DIMENSION(2*size(a)+1,2*size(b)+1)} : & \ \text{sig} \\
n = & \text{assert_eq(size(a),size(b),'orthog: n')} \\
\text{ndum} = & \text{assert_eq(2*n,size(alpha)+1,size(anu),size(beta)+1,'orthog: ndum')} \\
\text{sig}(1,3:2*n) = & 0.0 \\
\text{Initialization, Equation (4.5.33)}. \\
\text{sig}(2,2:2*n+1) = & \text{anu}(1:2*n) \\
a(1) = & \text{alpha}(1) + \text{anu}(2)/\text{anu}(1) \\
b(1) = & 0.0 \\
\text{do} k = 3, n+1 \\
\text{Equation (4.5.34).} \\
\text{sig}(k,k:2*n-k+3) = & \text{sig}(k-1,k:2*n-k+3) + (\text{alpha}(k-1:2*n-k+2) & \\
& - a(k-2))*\text{sig}(k-1,k+1:2*n-k+4) + b(k-2)*\text{sig}(k-2,k:2*n-k+4) & \\
& + \text{beta}(k-1:2*n-k+2) + \text{beta}(k-1,k:2*n-k+2) & \\
a(k-1) = & \text{alpha}(k-1) + \text{sig}(k,k+1)/\text{sig}(k,k) - \text{sig}(k-1,k)/\text{sig}(k-1,k-1) & \\
b(k-1) = & \text{sig}(k,k)/\text{sig}(k,k-1) & \\
\text{end do} \\
\end{align*}
\]

END SUBROUTINE orthog

As discussed in Volume 1, multidimensional quadrature can be performed by calling a one-dimensional quadrature routine along each dimension. If the same routine is used for all such calls, then the calls are recursive. The file quad3d.f90 contains two modules, quad3d_qgaus_mod and quad3d_qromb_mod. In the first, the basic one-dimensional quadrature routine is a 10-point Gaussian quadrature routine called qgaus and three-dimensional quadrature is performed by calling quad3d_qgaus. In the second, the basic one-dimensional routine is qromb of §4.3 and the three-dimensional routine is quad3d_qromb. The Gaussian quadrature is simpler but its accuracy is not controllable. The Romberg integration lets you specify an accuracy, but is apt to be very slow if you try for too much accuracy. The only difference between the stand-alone version of trapzd and the version included here is that we have to add the keyword RECURSIVE. The only changes from the stand-alone version of qromb are: We have to add RECURSIVE; we remove trapzd from the list of routines in USE nr; we increase EPS to \( 3 \times 10^{-6} \). Even this value could be too ambitious for difficult functions. You may want to set JMAX to a smaller value than 20 to avoid burning up a lot of computer time. Some people advocate using a smaller EPS on the inner quadrature (over \( z \) in our routine) than on the outer quadratures (over \( x \) or \( y \)). That strategy would require separate copies of qromb.

MODULE quad3d_qgaus_mod
USE nrtype
PRIVATE
PUBLIC quad3d_qgaus
REAL(SP) :: xsav,ysav
INTERFACE
FUNCTION func(x,y,z)
\text{The three-dimensional function to be integrated.} \\
\text{USE nrtype} \\
\text{REAL(SP), INTENT(IN)} :: x,y \\
\text{REAL(SP), DIMENSION(size(z))} :: func
END FUNCTION func
FUNCTION y1(x)
\text{User-supplied functions.} \\
\text{USE nrtype} \\
\text{REAL(SP), DIMENSION(size(x))} :: x \\
\text{REAL(SP)} :: z \\
\text{REAL(SP), INTENT(IN)} :: x
\text{END FUNCTION y1(x)} \\
\end{align*}
\]
REAL(SP), INTENT(IN) :: x
REAL(SP) :: y1
END FUNCTION y1

FUNCTION y2(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: y2
END FUNCTION y2

FUNCTION z1(x,y)
USE nrtype
REAL(SP), INTENT(IN) :: x,y
REAL(SP) :: z1
END FUNCTION z1

FUNCTION z2(x,y)
USE nrtype
REAL(SP), INTENT(IN) :: x,y
REAL(SP) :: z2
END FUNCTION z2

END INTERFACE

The routine quad3d_gaus returns as $ss$ the integral of a user-supplied function $func$ over a three-dimensional region specified by the limits $x_1$, $x_2$, and by the user-supplied functions $y_1$, $y_2$, $z_1$, and $z_2$, as defined in (4.6.2). Integration is performed by calling gaus recursively.

CONTAINS

FUNCTION h(x)
This is $H$ of eq. (4.6.5).
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: h
INTEGER(I4B) :: i
do i=1,size(x)
xsav=x(i)
h(i)=qgaus(g,y1(xsav),y2(xsav))
end do
END FUNCTION h

FUNCTION g(y)
This is $G$ of eq. (4.6.4).
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(size(y)) :: g
INTEGER(I4B) :: j
do j=1,size(y)
ysav=y(j)
g(j)=qgaus(f,z1(xsav,ysav),z2(xsav,ysav))
end do
END FUNCTION g

FUNCTION f(z)
The integrand $f(x,y,z)$ evaluated at fixed $x$ and $y$.
REAL(SP), DIMENSION(:,), INTENT(IN) :: z
REAL(SP), DIMENSION(size(z)) :: f
f=func(xsav,ysav,z)
END FUNCTION f

RECURSIVE FUNCTION qgaus(func,a,b)
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qgaus
END INTERFACE

FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func

END INTERFACE

REAL(SP) :: x,xr
REAL(SP), DIMENSION(5) :: dx, w = (/ 0.2955242247_sp,0.2692667193_sp,& 0.2190863625_sp,0.1494513491_sp,0.0666713443_sp /),& x = (/ 0.1488743389_sp,0.4333953941_sp,0.6794095682_sp,&
0.8650633666_sp,0.9739065285_sp /

xm=0.5_sp*(b+a)
xr=0.5_sp*(b-a)
dx(:)=xr*x(:)
qgaus=xr*sum(w(:)*(func(xm+dx)+func(xm-dx)))
END FUNCTION qgaus

SUBROUTINE quad3d_qgaus(x1,x2,ss)
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), INTENT(OUT) :: ss
ss=qgaus(h,x1,x2)
END SUBROUTINE quad3d_qgaus

END MODULE quad3d_qgaus_mod

PRIVATE...PUBLIC quad3d_qgaus  By default, all module entities are accessible by a routine that uses the module (unless we restrict the USE statement with ONLY). In this module, the user needs access only to the routine quad3d_qgaus; the variables xsav, ysav and the procedures f, g, h, and qgaus are purely internal. It is good programming practice to prevent duplicate name conflicts or data overwriting by limiting access to only the desired entities. Here the PRIVATE statement with no variable names resets the default from PUBLIC. Then we include in the PUBLIC statement only the function name we want to be accessible.

REAL(SP) :: xsav,ysav  In Fortran 90, we generally avoid declaring global variables in COMMON blocks. Instead, we give them complete specifications in a module. A deficiency of Fortran 90 is that it does not allow pointers to functions. So here we have to use the fixed-name function func for the function to be integrated over. If we could have a pointer to a function as a global variable, then we would just set the pointer to point to the user function (of any name) in the calling program. Similarly the functions y1, y2, z1, and z2 could also have any name.

CONTAINS  Here follow the internal subprograms f, g, h, qgaus, and quad3d_qgaus. Note that such internal subprograms are all “visible” to each other, i.e., their interfaces are mutually explicit, and do not require INTERFACE statements.

RECURSIVE SUBROUTINE qgaus(func,a,b,ss)  The RECURSIVE keyword is required for the compiler to process correctly any procedure that is invoked again in its body before the return from the first call has been completed. While some compilers may let you get away without explicitly informing them that a routine is recursive, don’t count on it!

MODULE quad3d_qromb_mod
Alternative to quad3d_qgaus_mod that uses qromb to perform each one-dimensional integration.

USE nrtype
PRIVATE
PUBLIC quad3d_qromb
REAL(SP) :: xsav,ysav
INTERFACE
    FUNCTION func(x,y,z)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(:), INTENT(IN) :: z
    REAL(SP), DIMENSION(size(z)) :: func
    END FUNCTION func

    FUNCTION y1(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: y1
END FUNCTION y1

FUNCTION y2(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: y2
END FUNCTION y2

FUNCTION z1(x,y)
USE nrtype
REAL(SP), INTENT(IN) :: x,y
REAL(SP) :: z1
END FUNCTION z1

FUNCTION z2(x,y)
USE nrtype
REAL(SP), INTENT(IN) :: x,y
REAL(SP) :: z2
END FUNCTION z2

END INTERFACE

CONTAINS

FUNCTION h(x)
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: h
INTEGER(I4B) :: i
DO i=1,size(x)
  xsav=x(i)
  h(i)=qromb(g,y1(xsav),y2(xsav))
END DO
END FUNCTION h

FUNCTION g(y)
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(size(y)) :: g
INTEGER(I4B) :: j
DO j=1,size(y)
  ysav=y(j)
  g(j)=qromb(f,z1(xsav,ysav),z2(xsav,ysav))
END DO
END FUNCTION g

FUNCTION f(z)
REAL(SP), DIMENSION(:), INTENT(IN) :: z
REAL(SP), DIMENSION(size(z)) :: f
f=func(xsav,ysav,z)
END FUNCTION f

RECURSIVE FUNCTION qromb(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: func
INTERFACES
  FUNCTION func(x)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: func
  END FUNCTION func
END INTERFACE

INTERFACE
  FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE

PARAMETER :: JMAX=20, JMAXP=JMAX+1, K=5, KM=K-1
REAL(SP), PARAMETER :: EPS=3.0e-6_sp
REAL(SP) :: dqromb
INTEGER(I4B) :: j
h(1)=1.0
do j=1,JMAX
  call trapzd(func,a,b,s(j),j)
  if (j >= K) then
    call polint(h(j-KM:j),s(j-KM:j),0.0_sp,qromb,dqromb)
    if (abs(dqromb) <= EPS*abs(qromb)) RETURN
  end if
  s(j+1)=s(j)
  h(j+1)=0.25_sp*h(j)
end do
end FUNCTION qromb

RECURSIVE SUBROUTINE trapzd(func,a,b,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
  FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
  END FUNCTION func
END INTERFACE
REAL(SP) :: del,fsum
INTEGER(I4B) :: it
if (n == 1) then
  s=0.5_sp*(b-a)*sum(func( (/ a,b /) ))
else
  it=2**(n-2)
  del=(b-a)/it
  fsum=sum(func(arth(a+0.5_sp*del,del,it)))
  s=0.5_sp*(s+del*fsum)
end if
END SUBROUTINE trapzd

SUBROUTINE quad3d_qromb(x1,x2,ss)
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), INTENT(OUT) :: ss
ss=qromb(h,x1,x2)
END SUBROUTINE quad3d_qromb
END MODULE quad3d_qromb_mod

MODULE quad3d_qromb_mod
The only difference between this module and the
previous one is that all calls to qgaus are replaced by calls to qromb and that the
routine qgaus is replaced by qromb and trapzd.

CITED REFERENCES AND FURTHER READING:
Erdélyi, A., Magnus, W., Oberhettinger, F., and Tricomi, F.G. 1953, Higher Transcendental
Chapter B5. Evaluation of Functions

SUBROUTINE eulsum(sum,term,jterm)
USE nrtype; USE nrutil, ONLY: poly_term, reallocate
IMPLICIT NONE
REAL(SP), INTENT(INOUT) :: sum
REAL(SP), INTENT(IN) :: term
INTEGER(I4B), INTENT(IN) :: jterm

Incorporates into sum the jterm'th term, with value term, of an alternating series. sum is input as the previous partial sum, and is output as the new partial sum. The first call to this routine, with the first term in the series, should be with jterm=1. On the second call, term should be set to the second term of the series, with sign opposite to that of the first call, and jterm should be 2. And so on.

REAL(SP), DIMENSION(:), POINTER, SAVE :: wksp
INTEGER(I4B), SAVE :: nterm
LOGICAL(LGT), SAVE :: init=.true.

if (init) then
  init=.false.
nullify(wksp)
end if
if (jterm == 1) then
  nterm=1
  wksp=>reallocate(wksp,100)
  wksp(1)=term
  sum=0.5_sp*term
  Return first estimate.
else
  if (nterm+1 > size(wksp)) wksp=>reallocate(wksp,2*size(wksp))
  wksp(2:nterm+1)=0.5_sp*wksp(1:nterm)
  Update saved quantities by van Wijngaarden's algorithm.
  wksp(1:nterm+1)=poly_term(wksp(1:nterm+1),0.5_sp)
  if (abs(wksp(nterm+1)) <= abs(wksp(nterm))) then
    Favorable to increase p,
    sum=sum+0.5_sp*wksp(nterm+1)
    nterm=nterm+1
    and the table becomes longer.
  else
    Favorable to increase n,
    sum=sum+wksp(nterm+1)
    the table doesn't become longer.
  end if
end if
END SUBROUTINE eulsum

This routine uses the function reallocate in nrutil to define a temporary workspace and then, if necessary, enlarge the workspace without destroying the earlier contents. The pointer wksp is declared with the SAVE attribute. Since Fortran 90 pointers are born “in limbo,” we cannot immediately test whether they are associated or not. Hence the code if (init)...nullify(wksp). Then the line wksp=>reallocate(wksp,100) allocates an array of length 100 and points wksp to it. On subsequent calls to eulsum, if nterm ever gets bigger than the size of wksp, the call to reallocate doubles the size of wksp and copies the old contents into the new storage.

You could achieve the same effect as the code if (init)...nullify(wksp)... wksp=>reallocate(wksp,100) with a simple allocate(wksp,100). You would then use
reallocate only for increasing the storage if necessary. Don’t! The advantage of the above scheme becomes clear if you consider what happens if eulsum is invoked twice by the calling program to evaluate two different sums. On the second invocation, when jterm = 1 again, you would be allocating an already allocated pointer. This does not generate an error — it simply leaves the original target inaccessible. Using reallocate instead not only allocates a new array of length 100, but also detects that wksp had already been associated. It dutifully (and wastefully) copies the first 100 elements of the old wksp into the new storage, and, more importantly, deallocates the old wksp, reclaiming its storage. While only two invocations of eulsum without intervening deallocation of memory would not cause a problem, many such invocations might well. We believe that, as a general rule, the potential for catastrophe from reckless use of allocate is great enough that you should always deallocate whenever storage is no longer required.

The unnecessary copying of 100 elements when eulsum is invoked a second time could be avoided by making init an argument. It hardly seems worth it to us.

For Fortran 90 neophytes, note that unlike in C you have to do nothing special to get the contents of the storage a pointer is addressing. The compiler figures out from the context whether you mean the contents, such as wksp(1:n term), or the address, such as both occurrences of wksp in wksp=>reallocate(wksp,100).

wksp(1:n term+1)=poly_term(wksp(1:n term+1),0.5_sp)  The poly_term function in nrutil tabulates the partial sums of a polynomial, or, equivalently, performs the synthetic division of a polynomial by a monomial.

Small-scale parallelism in eulsum is achieved straightforwardly by the use of vector constructions and poly_term (which parallelizes recursively).

The routine is not written to take advantage of data parallelism in the (infrequent) case of wanting to sum many different series simultaneously; nor, since wksp is a SAVEd variable, can it be used in many simultaneous instances on a MIMD machine. (You can easily recode these generalizations if you need them.)

SUBROUTINE ddpoly(c,x,pd)
USE nrtype; USE nrutil, ONLY : arth,cumprod,poly_term
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(:), INTENT(OUT) :: pd

Given the coefficients of a polynomial of degree N_c − 1 as an array c(1:N_c) with c(1) being the constant term, and given a value x, this routine returns the polynomial evaluated at x as pd(1) and N_d − 1 derivatives as pd(2:N_d).

INTEGER(I4B) :: i,nc,nd
REAL(SP), DIMENSION(size(pd)) :: fac
REAL(SP), DIMENSION(size(c)) :: d
nc=size(c)
nd=size(pd)
d(nc:i:-1)=poly_term(c(nc:i:-1),x)
do i=2,min(nd,nc)
da(nc:i:-1)=poly_term(d(nc:i:-1),x)
end do
pd=d(1:nd)
fac=cumprod(arth(1.0_sp,1.0_sp,nd))  After the first derivative, factorial constants come in.
pd(3:nd)=fac(2:nd-1)*pd(3:nd)
END SUBROUTINE ddpoly
The poly_term function in nrutil tabulates the partial sums of a polynomial, or, equivalently, performs synthetic division. See §22.3 for a discussion of why ddpoly is coded this way.

fac=cumprod(arth(1.0_sp,1.0_sp,nd))  Here the function arth from nrutil generates the sequence 1, 2, 3, ..., n. The function cumprod then tabulates the cumulative products, thus making a table of factorials.

Notice that ddpoly doesn’t need an argument to pass N_d, the number of output terms desired by the user: It gets that information from the length of the array pd that the user provides for it to fill. It is a minor curiosity that pd, declared as INTENT(OUT), can thus be used, on the sly, to pass some INTENT(IN) information.

(A Fortran 90 brain teaser could be: A subroutine with only INTENT(OUT) arguments can be called to print any specified integer. How is this done?)

SUBROUTINE poldiv(u,v,q,r)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: u,v
REAL(SP), DIMENSION(:), INTENT(OUT) :: q,r
Given the N coefficients of a polynomial in u, and the N_v coefficients of another polynomial in v, divide the polynomial u by the polynomial v (“u”/“v”) giving a quotient polynomial whose coefficients are returned in q, and a remainder polynomial whose coefficients are returned in r. The arrays q and r are of length N, but only the first N - N_v + 1 elements of q and the first N_v - 1 elements of r are used. The remaining elements are returned as zero.

INTEGER(I4B) :: i,n,nv
n=assert_eq(size(u),size(q),size(r),'poldiv')
nv=size(v)
r(:)=u(:)
q(:)=0.0
do i=n-nv,0,-1
  q(i+1)=r(nv+i)/v(nv)
  r(i+1:nv+i-1)=r(i+1:nv+i-1)-q(i+1)*v(1:nv-1)
end do
r(nv:n)=0.0
END SUBROUTINE poldiv

FUNCTION ratval_s(x,cof,mm,kk)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(DP), INTENT(IN) :: x  Note precision! Change to REAL(SP) if desired.
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
REAL(DP) :: ratval_s
Given mm, kk, and cof(1:mm+kk+1), evaluate and return the rational function (cof(1) + cof(2)x + ... + cof(mm+1)x^mm)/(1 + cof(mm+2)x + ... + cof(mm+kk+1)x^kk).
ratval_s=poly(x,cof(1:mm+1))/(1.0_dp+x*poly(x,cof(mm+2:mm+kk+1)))
END FUNCTION ratval_s

This simple routine uses the function poly from nrutil to evaluate the numerator and denominator polynomials. Single- and double-precision versions, ratval_s and ratval_v, are overloaded onto the name ratval when the module nr is used.
FUNCTION ratval_v(x,cof,mm,kk)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
REAL(DP), DIMENSION(size(x)) :: ratval_v
ratval_v=poly(x,cof(1:mm+1))/(1.0_dp+x*poly(x,cof(mm+2:mm+kk+1)))
END FUNCTION ratval_v

The routines recur1 and recur2 are new in this volume, and do not have Fortran 77 counterparts. First- and second-order linear recurrences are implemented as trivial do-loops on strictly serial machines. On parallel machines, however, they pose different, and quite interesting, programming challenges. Since many calculations can be decomposed into recurrences, it is useful to have general, parallelizable routines available. The algorithms behind recur1 and recur2 are discussed in §22.2.

RECURSIVE FUNCTION recur1(a,b) RESULT(u)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a)) :: u
INTEGER(I4B), PARAMETER :: NPAR_RECUR1=8
Given vectors a of size n and b of size n-1, returns a vector u that satisfies the first order linear recurrence $u_1 = a_1$, $u_j = a_j + b_{j-1}u_{j-1}$, for $j = 2,...,n$. Parallelization is via a recursive evaluation.
INTEGER(I4B) :: n,j
n=assert_eq(size(a),size(b)+1,'recur1')
if (n < NPAR_RECUR1) then
   Do short vectors as a loop.
   do j=2,n
      u(j)=a(j)+b(j-1)*u(j-1)
   end do
else
   Otherwise, combine coefficients and recurse on the even components, then evaluate all the odd components in parallel.
u(2:n:2)=recur1(a(2:n:2)+a(1:n-1:2)*b(1:n-1:2), &
b(3:n-1:2)*b(2:n-2:2))
u(3:n:2)=a(3:n:2)+b(2:n-1:2)*u(2:n-1:2)
end if
END FUNCTION recur1

When a recursive function invokes itself only indirectly through a sequence of function calls, then the function name can be used for the result just as in a nonrecursive function. When the function invokes itself directly, however, as in recur1, another name must be used for the result. If you are hazy on the syntax for RESULT, see the discussion of recursion in §21.5.
FUNCTION recur2(a,b,c)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c
REAL(SP), DIMENSION(size(a)) :: recur2

Given vectors a of size n and b and c of size n−2, returns a vector u that satisfies the second order linear recurrence

\[ u_1 = a_1, \quad u_2 = a_2, \quad u_j = a_j + b_{j-2}u_{j-1} + c_{j-2}u_{j-2}, \text{ for } j = 3, \ldots, n. \]

Parallelization is via conversion to a first order recurrence for a two-dimensional vector.

INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(a)-1) :: a1,a2,u1,u2
REAL(SP), DIMENSION(size(a)-2) :: b11,b12,b21,b22

n=assert_eq(size(a),size(b)+2,size(c)+2,'recur2')
a1(1)=a(1)
a2(1)=a(2)
a1(2:n-1)=0.0
a2(2:n-1)=a(3:n)
b11(1:n-2)=0.0
b12(1:n-2)=1.0
b21(1:n-2)=c(1:n-2)
b22(1:n-2)=b(1:n-2)
call recur1_v(a1,a2,b11,b12,b21,b22,u1,u2)
recur2(1:n-1)=u1(1:n-1)
recur2(n)=u2(n-1)
CONTAINS

RECURSIVE SUBROUTINE recur1_v(a1,a2,b11,b12,b21,b22,u1,u2)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a1,a2,b11,b12,b21,b22
REAL(SP), DIMENSION(:), INTENT(OUT) :: u1,u2
INTEGER(I4B), PARAMETER :: NPAR_RECUR2=8

Used by recur2 to evaluate first order vector recurrence. Routine is a two-dimensional vector version of recur1, with matrix multiplication replacing scalar multiplication.

INTEGER(I4B) :: n,j,nn,nn1
REAL(SP), DIMENSION(size(a1)/2) :: aa1,aa2
REAL(SP), DIMENSION(size(a1)/2-1) :: bb11,bb12,bb21,bb22
n=assert_eq(size(a1)/2,size(a2),size(b11)+1,size(b12)+1,size(b21)+1,&
    size(b22)+1,size(u1),size(u2))
u1(1)=a1(1)
u2(1)=a2(1)
if (n < NPAR_RECUR2) then
    Do short vectors as a loop.
do j=2,n
    u1(j)=a1(j)+b11(j-1)*u1(j-1)+b12(j-1)*u2(j-1)
    u2(j)=a2(j)+b21(j-1)*u1(j-1)+b22(j-1)*u2(j-1)
end do
else
    Otherwise, combine coefficients and recurse on the even components, then evaluate all the odd components in parallel.
n=n/2
nn=nn-1
aa1(1:nn)=a1(2:n:2)+b11(1:n-1:2)*u1(1:n-1:2)+&
b12(1:n-1:2)*u2(1:n-1:2)
aa2(1:nn)=a2(2:n:2)+b21(1:n-1:2)*u1(1:n-1:2)+&
b22(1:n-1:2)*u2(1:n-1:2)
bb11(1:nn)=b11(3:n-2:2)*b11(2:n-2:2)+&
b12(3:n-2:2)*b12(2:n-2:2)
bb12(1:nn)=b11(3:n-2:2)*b12(2:n-2:2)+&
b21(3:n-2:2)*b22(2:n-2:2)
bb21(1:nn)=b21(3:n-2:2)*b11(2:n-2:2)+&
b22(3:n-2:2)*b21(2:n-2:2)
bb22(1:nn)=b21(3:n-2:2)*b22(2:n-2:2)
call recur1_v(aa1,aa2,bb11,bb12,bb21,bb22,u1(2:n:2),u2(2:n:2))
u1(3:n:2)=a1(3:n:2)+b11(2:n-1:2)*u1(2:n-1:2)+&
end if
CONTAINS

FUNCTION dfridr(func,x,h,err)
USE nrtype; USE nrutil, ONLY : assert, geop, iminloc
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,h
REAL(SP), INTENT(OUT) :: err
REAL(SP) :: dfridr
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: NTAB=10
REAL(SP), PARAMETER :: CON=1.4_sp, CON2=CON*CON, BIG=huge(x), SAFE=2.0

Returns the derivative of a function func at a point x by Ridders' method of polynomial extrapolation. The value h is input as an estimated initial stepsize; it need not be small, but rather should be an increment in x over which func changes substantially. An estimate of the error in the derivative is returned as err.

Parameters: Stepwise is decreased by CON at each iteration. Max size of tableau is set by NTAB. Return when error is SAFE worse than the best so far.

INTEGER(I4B), PARAMETER :: NTAB=10
REAL(SP) :: h, err
REAL(SP), DIMENSION(NTAB-1) :: errt, fac
REAL(SP), DIMENSION(NTAB,NTAB) :: a
CALL assert(h /= 0.0, 'dfridr arg')
h=CON
a(1,1)=(func(x+h)-func(x-h))/2.0_sp*h
err=BIG
fac(1:NTAB-1)=geop(CON2,CON2,NTAB-1)
DO i=2,NTAB
     SUCCESSIVE COLUMNS IN THE NEVILLE TABLEAU WILL GO TO SMALLER STEPSIZES AND HIGHER ORDERS OF EXTRAPOLATION.
     a(i,1)=(func(x+h)-func(x-h))/2.0_sp*h
     err=max(abs(a(2:i,i)-a(1:i-1,i)),abs(a(2:i,i)-a(1:i-1,i-1)))
     IF (err(ierrmin) <= err) THEN
         IF error is decreased, save the improved answer.
         err=err(ierrmin)
         dfridr=a(1+ ierrmin,1)
     ELSE
         IF higher order is worse by a significant factor SAFE, then quit early.
     END IF
END DO
END FUNCTION dfridr
The function `iminloc` in `nrutil` is useful when you need to know the index of the smallest element in an array.

\[
ierrmin=\text{iminloc}(\text{errt}(1:i-1))
\]

\[
\text{FUNCTION chebft(a,b,n,func)}
\]

\[
\text{USE nrtype; USE nrutil, ONLY : arth,outerprod}
\]

\[
\text{IMPLICIT NONE}
\]

\[
\text{REAL(SP), INTENT(IN) :: a,b}
\]

\[
\text{INTEGER(I4B), INTENT(IN) :: n}
\]

\[
\text{REAL(SP), DIMENSION(n) :: chebft}
\]

\[
\text{INTERFACE}
\]

\[
\text{FUNCTION func(x)}
\]

\[
\text{USE nrtype}
\]

\[
\text{IMPLICIT NONE}
\]

\[
\text{REAL(SP), DIMENSION(:,), INTENT(IN) :: x}
\]

\[
\text{REAL(SP), DIMENSION(size(x)) :: func}
\]

\[
\text{END FUNCTION func}
\]

\[
\text{END INTERFACE}
\]

Chebyshev fit: Given a function \(\text{func}\), lower and upper limits of the interval \([a,b]\), and a maximum degree \(n\), this routine computes the \(n\) coefficients \(c_k\) such that

\[
\text{func}(x) \approx \sum_{k=1}^{n} c_k T_{k-1}(y) - c_1/2,
\]

where \(y\) and \(x\) are related by (5.8.10). This routine is to be used with moderately large \(n\) (e.g., 30 or 50), the array of \(c\)'s subsequently to be truncated at the smaller value \(m\) such that \(c_{m+1}\) and subsequent elements are negligible.

\[
\text{REAL(DP) :: bma,bpa}
\]

\[
\text{REAL(DP), DIMENSION(n) :: theta}
\]

\[
bma=0.5_dp*(b-a)
\]

\[
bpa=0.5_dp*(b+a)
\]

\[
\theta(:)=\pi_D*\text{arth}(0.5_dp,1.0_dp,n)/n
\]

\[
\text{chebft(:)=matmul(cos(outerprod(arth(0.0_dp,1.0_dp,n),theta)), &}
\]

\[
\text{func(real(cos(theta)*bma+bpa,sp)))*2.0_dp/n}
\]

We evaluate the function at the \(n\) points required by (5.8.7). We accumulate the sum in double precision for safety.

\[
\text{END FUNCTION chebft}
\]

\[
\text{FUNCTION chebev_s(a,b,c,x)}
\]

\[
\text{USE nrtype; USE nrutil, ONLY : nrerror}
\]

\[
\text{IMPLICIT NONE}
\]

\[
\text{REAL(SP), INTENT(IN) :: a,b,x}
\]

\[
\text{REAL(SP), DIMENSION(:,), INTENT(IN) :: c}
\]

\[
\text{REAL(SP) :: chebev_s}
\]

Chebyshev evaluation: All arguments are input. \(c\) is an array of length \(M\) of Chebyshev coefficients, the first \(M\) elements of \(c\) output from `chebft` (which must have been called
with the same \(a\) and \(b\). The Chebyshev polynomial \[\sum_{k=1}^{M} c_k T_{k-1}(y) - c_1/2\] is evaluated at a point \(y = \left[ x - (b + a)/2 \right]/\left[ (b - a)/2 \right]\), and the result is returned as the function value.

```fortran
INTEGER(I4B) :: j,m
REAL(SP) :: d,dd,sv,y,y2
if ((x-a)*(x-b) > 0.0) call nrerror('x not in range in chebev_s')
    m=size(c)
    d=0.0
    dd=0.0
    y=(2.0_sp*x-a-b)/(b-a)
    y2=2.0_sp*y
    do j=m,2,-1
        sv=d
        d=y2*d-dd+c(j)
        dd=sv
    end do
    chebev_s=y*d-dd+0.5_sp*c(1)
END FUNCTION chebev_s
```

The name `chebev` is overloaded with scalar and vector versions. `chebev_v` is essentially identical to `chebev_s` except for the declarations of the variables. Fortran 90 does the appropriate scalar or vector arithmetic in the body of the routine, depending on the type of the variables.

```fortran
FUNCTION chebev_v(a,b,c,x)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c,x
REAL(SP), DIMENSION(size(x)) :: chebev_v
INTEGER(I4B) :: j,m
REAL(SP), DIMENSION(size(x)) :: d,dd,sv,y,y2
if (any((x-a)*(x-b) > 0.0)) call nrerror('x not in range in chebev_v')
    m=size(c)
    d=0.0
    dd=0.0
    y=(2.0_sp*x-a-b)/(b-a)
    y2=2.0_sp*y
    do j=m,2,-1
        sv=d
        d=y2*d-dd+c(j)
        dd=sv
    end do
    chebev_v=y*d-dd+0.5_sp*c(1)
END FUNCTION chebev_v
```

The name `chebev` is essentially identical to `chebev_s` except for the declarations of the variables. Fortran 90 does the appropriate scalar or vector arithmetic in the body of the routine, depending on the type of the variables.

```fortran
FUNCTION chder(a,b,c)
USE nrtype; USE nrutil, ONLY : arth,cumsum
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chder
    This routine returns an array of length \(N\) containing the Chebyshev coefficients of the derivative of the function whose coefficients are in the array \(c\). Input are \(a, b, c\), as output
```

```fortran
FUNCTION chder(a,b,c)
USE nrtype; USE nrutil, ONLY : arth,cumsum
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chder
    This routine returns an array of length \(N\) containing the Chebyshev coefficients of the derivative of the function whose coefficients are in the array \(c\). Input are \(a, b, c\), as output
```
Chapter B5. Evaluation of Functions

from routine `chebft` §5.8. The desired degree of approximation $N$ is equal to the length of $c$ supplied.

```fortran
INTEGER(I4B) :: n
REAL(SP) :: con
REAL(SP), DIMENSION(size(c)) :: temp

n=size(c)
temp(1)=0.0
temp(2:n)=2.0_sp*arth(n-1,-1,n-1)*c(n:2:-1)
chder(n:1:-2)=cumsum(temp(1:n:2))
chder(n-1:1:-2)=cumsum(temp(2:n:2))

END FUNCTION chder
```

```
FUNCTION chint(a,b,c)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chint

INTEGER(I4B) :: n
REAL(SP) :: con

n=size(c)
con=0.25_sp*(b-a)
chint(2:n-1)=con*(c(1:n-2)-c(3:n))/arth(1,1,n-2)
chint(n)=con*c(n-1)/(n-1)
chint(1)=2.0_sp*(sum(chint(2:n:2))-sum(chint(3:n:2)))

END FUNCTION chint
```

```
FUNCTION chebpc(c)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chebpc

INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(c)) :: dd,sv

n=size(c)
chebpc=0.0

dd=0.0
chebpc(1)=c(n)
do j=n-1,2,-1
    sv(2:n-j+1)=chebpc(2:n-j+1)
    chebpc(2:n-j+1)=2.0_sp*chebpc(1:n-j)-dd(2:n-j+1)
    dd(2:n-j+1)=sv(2:n-j+1)
end do

```

If you look at equation (5.9.1) for the Chebyshev coefficients of the integral of a function, you will see $c_{i-1}$ and $c_{i+1}$ and be tempted to use `eoshift`. We think it is almost always better to use array sections instead, as in the code above, especially if your code will ever run on a serial machine.

\* \* \*

FUNCTION chebpc(c)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chebpc

Chebyshev polynomial coefficients. Given a coefficient array $c$ of length $N$, this routine returns a coefficient array $d$ of length $N$ such that $\sum_{k=1}^{N} d_k T_k - c_1/2$. The method is Clenshaw's recurrence (5.8.11), but now applied algebraically rather than arithmetically.

```
INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(c)) :: dd,sv

n=size(c)
chebpc=0.0

dd=0.0
chebpc(1)=c(n)
do j=n-1,2,-1
    sv(2:n-j+1)=chebpc(2:n-j+1)
    chebpc(2:n-j+1)=2.0_sp*chebpc(1:n-j)-dd(2:n-j+1)
    dd(2:n-j+1)=sv(2:n-j+1)
end do
```
sv(1)=chebpc(1)
chebpc(1)=dd(1)+c(j)
end do
chebpc(2:n)=chebpc(1:n-1)-dd(2:n)
chebpc(1)=dd(1)+0.5 * c(1)
END FUNCTION chebpc

* * *

SUBROUTINE pcshft(a,b,d)
USE nrtype; USE nrutil, ONLY : geop
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d

Polynomial coefficient shift. Given a coefficient array d of length N, this routine generates a coefficient array g of the same length such that \[ \sum_{k=1}^{N} d_k y^{k-1} = \sum_{k=1}^{N} g_k x^{k-1}, \] where x and y are related by (5.8.10), i.e., the interval -1 < y < 1 is mapped to the interval a < x < b. The array y is returned in d.

INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(d)) :: dd
REAL(SP) :: x
n=size(d)
d=d*geop(1.0_sp,2.0_sp/(b-a),n)
x=-0.5 * (a+b)
d(1)=dd(n)
d(2:n)=0.0
do j=n-1,1,-1
  d(2:n+1-j)=d(2:n+1-j)*x+d(1:n-j)
d(1)=d(1)*x+dd(j)
end do
END SUBROUTINE pcshft

There is a subtle, but major, distinction between the synthetic division algorithm used in the Fortran 77 version of pcshft and that used above. In the Fortran 77 version, the synthetic division (translated to Fortran 90 notation) is

\[
d(1:n)=dd(1:n)
do j=1,n-1
do k=n,1,-1
  d(k)=x*d(k+1)+d(k)
end do
end do
\]

while, in Fortran 90, it is

\[
d(1)=dd(n)
d(2:n)=0.0
do j=n-1,1,-1
  d(2:n+1-j)=d(2:n+1-j)*x+d(1:n-j)
d(1)=d(1)*x+dd(j)
end do
\]

As explained in §22.3, these are algebraically — but not algorithmically — equivalent. The inner loop in the Fortran 77 version does not parallelize, because each k value uses the result of the previous one. In fact, the k loop is a synthetic division, which can be parallelized recursively (as in the nrutil routine poly slurf), but not simply...
vectors. In the Fortran 90 version, since not one but \( n-1 \) successive synthetic divisions are to be performed (by the outer loop), it is possible to reorganize the calculation to allow vectorization.

```
FUNCTION pccheb(d)
    USE nrtype; USE nrutil, ONLY : arth, cumprod, geop
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: d
    REAL(SP), DIMENSION(size(d)) :: pccheb

    Inverse of routine chebpc: given an array of polynomial coefficients \( d \), returns an equivalent array of Chebyshev coefficients of the same length.

    INTEGER(I4B) :: k, n
    REAL(SP), DIMENSION(size(d)) :: denom, numer, pow
    n = size(d)
    pccheb(1) = 2.0_sp * d(1)
    pow = geop(1.0_sp, 2.0_sp, n)
    numer(1) = 1.0
    denom(1) = 1.0
    denom(2:(n+3)/2) = cumprod(arth(1.0_sp, 1.0_sp, (n+1)/2))
    pccheb(2:n) = 0.0
    do k = 2, n
        Loop over orders of \( x \) in the polynomial.
        numer(2:(k+3)/2) = cumprod(arth(k-1.0_sp, -1.0_sp, (k+1)/2))
        pccheb(k:1:-2) = pccheb(k:1:-2) + d(k) / pow(k-1) * numer(1:(k+1)/2) / denom(1:(k+1)/2)
    end do
END FUNCTION pccheb
```

```
SUBROUTINE pade(cof, resid)
    USE nrtype
    USE nr, ONLY : lubksb, ludcmp, mprove
    IMPLICIT NONE
    REAL(DP), DIMENSION(:), INTENT(INOUT) :: cof DP for consistency with ratval.
    REAL(SP), INTENT(OUT) :: resid

    Given \( \text{cof}(1:2N+1) \), the leading terms in the power series expansion of a function, solve the linear Padé equations to return the coefficients of a diagonal rational function approximation to the same function, namely
    \[
    \left( \text{cof}(1) + \text{cof}(2)x + \cdots + \text{cof}(N+1)x^N \right) / \left( 1 + \text{cof}(N+2)x + \cdots + \text{cof}(2N+1)x^N \right).
    \]
    The value \( \text{resid} \) is the norm of the residual vector; a small value indicates a well-converged solution.

    INTEGER(I4B) :: k, n
    INTEGER(I4B), DIMENSION((size(cof)-1)/2) :: indx
    REAL(SP), PARAMETER :: BIG = 1.0e30_sp
    REAL(SP) :: d, rr, rrold
    REAL(SP), DIMENSION((size(cof)-1)/2) :: x, y, z
    REAL(SP), DIMENSION((size(cof)-1)/2, (size(cof)-1)/2) :: q, qlu
    n = (size(cof)-1)/2
    x = cof(n+2:2*n+1)
    Set up matrix for solving.
    y = x
    do k = 1, n
        \( q(:,k) = \text{cof}(n+2-k:2*n+1-k) \)
        end do
    qlu = q
    call ludcmp(qlu, indx, d)
    call lubksb(qlu, indx, x)
    \( \text{rr} = \text{BIG} \)
    rr = rr - rr
    \( \text{rrold} = \text{rr} \)
    Solve by \( LU \) decomposition and backsubstitution.
    Important to use iterative improvement, since the Padé equations tend to be ill-conditioned.
```
Chapter B5. Evaluation of Functions

1081

SUBROUTINE pade

⋆⋆⋆

SUBROUTINE ratlsq(func,a,b,mm,kk,cof,dev)

USE nrtype; USE nrutil, ONLY : arth,geop
USE nr, ONLY : ratval,svbksb,svdcmp
IMPLICIT NONE
REAL(DP), INTENT(IN) :: a,b
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(:), INTENT(OUT) :: cof
REAL(DP), INTENT(OUT) :: dev
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(DP), DIMENSION(:), INTENT(IN) :: x
REAL(DP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: NPFAC=8,MAXIT=5
REAL(DP), PARAMETER :: BIG=1.0e30_dp

Returns in cof(1:mm+kk+1) the coefficients of a rational function approximation to the function func in the interval (a,b). Input quantities mm and kk specify the order of the numerator and denominator, respectively. The maximum absolute deviation of the approximation (insofar as is known) is returned as dev. Note that double-precision versions of svdcmp and svbksb are called.

INTEGER(I4B) :: it,ncof,npt,*npth
REAL(DP) :: devmax,e,theta
REAL(DP), DIMENSION((mm+kk+1)*NPFAC) :: bb,ee,fs,wt,xs
REAL(DP), DIMENSION(mm+kk+1) :: coff,w
REAL(DP), DIMENSION(mm+kk+1,mm+kk+1) :: v
REAL(DP), DIMENSION((mm+kk+1)*NPFAC,mm+kk+1) :: u,temp
ncof=mm+kk+1
npt=NPFAC*ncof
npth=npt/2
dev=BIG
theta=PIO2_D/(npt-1)
x[1:npt-1]=a+(b-a)*sin(theta*arth(0,1,npt-1))**2
	Now fill arrays with mesh ascissas and function values. At each end, use formula that minimizes roundoff sensitivity in xs.
x[npt]=b-(b-a)*sin(theta*arth(npt-npth,-1,npt-npth+1))**2
fs=func(xs)
w[1]=1.0	In later iterations we will adjust these weights to combat the largest deviations.
e=0.0
do it=1,MAXIT
	Loop over iterations.
bb=wt*sign(e,ee)
	Key idea here: Fit to fn(x)+e where the deviation is positive, to fn(x)-e where it is negative. Then e is supposed to become an approximation to the equal-ripple deviation.
temp=geop(spread(1.0_dp,1,npt),xs,ncof)

...
Note that vector form of \text{geop} (returning matrix) is being used.

\begin{align*}
u((1:mm+1)) &= \text{temp}(:,1:mm+1) \ast \text{spread}(\text{wt},2:mm+1) \\
\text{Set up the "design matrix" for the least squares fit.}\end{align*}

\begin{align*}
u((1:mm+2:ncof)) &= \text{temp}(:,2:ncof-mm) \ast \text{spread}(\text{bb},2,ncof-mm-1) \\
\text{call svdcmp}(u,u,v) \\
\text{Singular Value Decomposition. In especially singular or difficult cases, one might here edit the singular values } &u(1:ncof), \text{ replacing small values by zero.}\end{align*}

\begin{align*}
\text{call svbksb}(u,w,v,bb) \\
e &= \text{ratval}(xs,coff,mm,kk)-fs \\
\text{Tabulate the deviations and revise the weights.} \\
devmax &= \maxval(wt) \\
\text{if (devmax } \leq \text{ dev }) \text{ then} \\
\text{end if} \\
\text{write(*,10) it,devmax} \\
\text{end do} \\
10 \text{ format (' ratlsq iteration=',i2, ' max error=',1p,e10.3) \text{ END SUBROUTINE ratlsq}
\end{align*}

\begin{align*}
temp &= \text{geop}(\text{spread}(1.0_dp,1,npt),xs,ncof) \\
\text{The design matrix } u_{ij} \text{ is defined for } i = 1, \ldots, npts \text{ by} \\
&u_{ij} = \begin{cases} \\
&w_j x_i^{j-1}, \quad j = 1, \ldots, m + 1 \\
&-b_j x_i^{j-m-2}, \quad j = m + 2, \ldots, n \\
\end{cases} \quad (B5.12)
\end{align*}

The first case in equation (B5.12) is computed in parallel by constructing the matrix temp equal to

\[
\begin{bmatrix}
1 & x_1 & x_1^2 & \cdots \\
1 & x_2 & x_2^2 & \cdots \\
1 & x_3 & x_3^2 & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{bmatrix}
\]

and then multiplying by the matrix \text{spread}(\text{wt},2:mm+1), which is just

\[
\begin{bmatrix}
w_1 & w_1 & w_1 & \cdots \\
w_2 & w_2 & w_2 & \cdots \\
w_3 & w_3 & w_3 & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{bmatrix}
\]

(\text{Remember that multiplication using } \ast \text{ means element-by-element multiplication, not matrix multiplication.)} \text{ A similar construction is used for the second part of the design matrix.}
Chapter B6. Special Functions

A Fortran 90 intrinsic function such as $\sin(x)$ is both *generic* and *elemental*. Generic means that the argument $x$ can be any of multiple intrinsic data types and kind values (in the case of $\sin$, any real or complex kind). Elemental means that $x$ need not be a scalar, but can be an array of any rank and shape, in which case the calculation of $\sin$ is performed independently for each element.

Ideally, when we implement more complicated special functions in Fortran 90, as we do in this chapter, we would make them, too, both generic and elemental. Unfortunately, the language standard does not completely allow this. User-defined elemental functions are prohibited in Fortran 90, though they will be allowed in Fortran 95. And, there is no fully automatic way of providing for a single routine to allow arguments of multiple data types or kinds — nothing like C++’s “class templates,” for example.

However, don’t give up hope! Fortran 90 does provide a powerful mechanism for overloading, which can be used (perhaps not always with a maximum of convenience) to *simulate* both generic and elemental function features. In most cases, when we implement a special function with a scalar argument, $\text{gammln}(x)$ say, we will also implement a corresponding vector-valued function of vector argument that evaluates the special function for each component of the vector argument. We will then overload the scalar and vector version of the function onto the same function name. For example, within the $\text{nr}$ module are the lines

```fortran
INTERFACE gammln
  FUNCTION gammln_s(xx)
    USE nrtype
    REAL(SP), INTENT(IN) :: xx
    REAL(SP) :: gammln_s
  END FUNCTION gammln_s

  FUNCTION gammln_v(xx)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xx
    REAL(SP), DIMENSION(size(xx)) :: gammln_v
  END FUNCTION gammln_v
END INTERFACE
```

which can be included by a statement like “USE $\text{nr}$, ONLY: $\text{gammln}$,” and then allow you to write $\text{gammln}(x)$ without caring (or even thinking about) whether $x$ is a scalar or a vector. If you want arguments of even higher rank (matrices, and so forth), you can provide these yourself, based on our models, and overload them, too.

That takes care of “elemental”; what about “generic”? Here, too, overloading provides an acceptable, if not perfect, solution. Where double-precision versions of special functions are needed, you can in many cases easily construct them from our provided routines by changing the variable kinds (and any necessary convergence
parameters), and then additionally overload them onto the same generic function names. (In general, in the interest of brevity, we will not ourselves do this for the functions in this chapter.)

At first meeting, Fortran 90’s overloading capability may seem trivial, or merely cosmetic, to the Fortran 77 programmer; but one soon comes to rely on it as an important conceptual simplification. Programming at a “higher level of abstraction” is usually more productive than spending time “bogged down in the mud.” Furthermore, the use of overloading is generally fail-safe: If you invoke a generic name with arguments of shapes or types for which a specific routine has not been defined, the compiler tells you about it.

We won’t reprint the module nr’s interface blocks for all the routines in this chapter. When you see routines named something-s and something-v, below, you can safely assume that the generic name something is defined in the module nr and overloaded with the two specific routine names. A full alphabetical listing of all the interface blocks in nr is given in Appendix C2.

Given our heavy investment, in this chapter, in overloadable vector-valued special function routines, it is worth discussing whether this effort is simply a stopgap measure for Fortran 90, soon to be made obsolete by Fortran 95’s provision of user-definable ELEMENTAL procedures. The answer is “not necessarily,” and takes us into some speculation about the future of SIMD, versus MIMD, computing.

Elemental procedures, while applying the same executable code to each element, do not insist that it be feasible to perform all the parallel calculations in lockstep. That is, elemental procedures can have tests and branches (if-then-else constructions) that result in different elements being calculated by totally different pieces of code, in a fashion that can only be determined at run time. For true 100% MIMD (multiple instruction, multiple data) machines, this is not a problem: individual processors do the individual element calculations asynchronously.

However, virtually none of today’s (and likely tomorrow’s) largest-scale parallel supercomputers are 100% MIMD in this way. While modern parallel supercomputers increasingly have MIMD features, they continue to reward the use of SIMD (single instruction, multiple data) code with greater computational speed, often because of hardware pipelining or vector processing features within the individual processors. The use of Fortran 90 (or, for that matter Fortran 95) in a data-parallel or SIMD mode is thus by no means superfluous, or obviated by Fortran 95’s ELEMENTAL construction.

The problem we face is that parallel calculation of special function values often doesn’t fit well into the SIMD mold: Since the calculation of the value of a special function typically requires the convergence of an iterative process, as well as possible branches for different values of arguments, it cannot in general be done efficiently with “lockstep” SIMD programming.

Luckily, in particular cases, including most (but not all) of the functions in this chapter, one can in fact make reasonably good parallel implementations with the SIMD tools provided by the language. We will in fact see a number of different tricks for accomplishing this in the code that follows.

We are interested in demonstrating SIMD techniques, but we are not completely impractical. None of the data-parallel implementations given below are too inefficient on a scalar machine, and some may in fact be faster than Fortran 95’s ELEMENTAL
alternative, or than do-loops over calls to the scalar version of the function. On a scalar machine, how can this be? We have already, above, hinted at the answer: (i) most modern scalar processors can overlap instructions to some degree, and data-parallel coding often provides compilers with the ability to accomplish this more efficiently; and (ii) data-parallel code can sometimes give better cache utilization.

```fortran
FUNCTION gammln_s(xx)
USE nrtype; USE nrutil, ONLY : arth, assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xx
REAL(SP) :: gammln_s

Returns the value \( \ln[\Gamma(xx)] \) for \( xx > 0 \).

REAL(DP) :: tmp,x
Internal arithmetic will be done in double precision, a nicety that you can omit if five-figure accuracy is good enough.

REAL(DP) :: stp = 2.5066282746310005_dp
REAL(DP), DIMENSION(6) :: coef = (/76.18009172947146_dp,&
-86.50532032941677_dp,24.01409824083091_dp,&
-1.231739572450155_dp,0.120865097366179e-2_dp,&
-0.5395239384953e-5_dp/)
call assert(xx > 0.0, 'gammln_s arg')
x=xx
tmp=x+5.5_dp
tmp=(x+0.5_dp)*log(tmp)-tmp
gammln_s=tmp+log(stp*(1.000000000190015_dp+&
sum(coef(:)/arth(x+1.0_dp,1.0_dp,size(coef))))/x)
END FUNCTION gammln_s

FUNCTION gammln_v(xx)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
INTEGER(I4B) :: i
REAL(SP), DIMENSION(:), INTENT(IN) :: xx
REAL(SP), DIMENSION(size(xx)) :: gammln_v
REAL(DP), DIMENSION(size(xx)) :: ser,tmp,x,y
REAL(DP) :: stp = 2.5066282746310005_dp
REAL(DP), DIMENSION(6) :: coef = (/76.18009172947146_dp,&
-86.50532032941677_dp,24.01409824083091_dp,&
-1.231739572450155_dp,0.120865097366179e-2_dp,&
-0.5395239384953e-5_dp/)
if (size(xx) == 0) RETURN
call assert(all(xx > 0.0), 'gammln_v arg')
x=xx
tmp=x+5.5_dp
tmp=(x+0.5_dp)*log(tmp)-tmp
ser=1.000000000190015_dp
y=x
do i=1,size(coef)
y=y+1.0_dp
ser=ser+coef(i)/y
end do
gammln_v=tmp+log(stp*ser/x)
END FUNCTION gammln_v
```
We use the nrutil routine assert for functions that have restrictions on the allowed range of arguments. One could instead have used an if statement with a call to nrerror; but we think that the uniformity of using assert, and the fact that its logical arguments read the “desired” way, not the “erroneous” way, make for a clearer programming style. In the vector version, the assert line is:

call assert(all(xx > 0.0), 'gammln_v arg')

Notice that the scalar and vector versions achieve parallelism in quite different ways, something that we will see many times in this chapter. In the scalar case, parallelism (at least small-scale) is achieved through constructions like

\[
\text{sum(coef(:)/arth(x+1.0_dp,1.0_dp,size(coef)))}
\]

Here vector utilities construct the series \(x+1, x+2, \ldots\) and then sum a series with these terms in the denominators and a vector of coefficients in the numerators. (This code may seem terse to Fortran 90 novices, but once you get used to it, it is quite clear to read.)

In the vector version, by contrast, parallelism is achieved across the components of the vector argument, and the above series is evaluated sequentially as a do-loop. Obviously the assumption is that the length of the vector argument is much longer than the very modest number (here, 6) of terms in the sum.

\[
\star \quad \star \quad \star
\]

FUNCTION factrl_s(n)
USE nrtyp; USE nrutil, ONLY : arth,assert,cumprod
USE nr, ONLY : gammln
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP) :: factrl_s

Returns the value \(n!\) as a floating-point number.

INTEGER(I4B), SAVE :: ntop=0
INTEGER(I4B), PARAMETER :: NMAX=32
REAL(SP), DIMENSION(NMAX), SAVE :: a

Table of stored values.
call assert(n >= 0, 'factrl_s arg')
if (n < ntop) then
  factrl_s=a(n+1)
else if (n < NMAX) then
  ntop=NMAX
  a(1)=1.0
  a(2:NMAX)=cumprod(arth(1.0_sp,1.0_sp,NMAX-1))
  factrl_s=a(n+1)
else
  factrl_s=exp(gammln(n+1.0_sp))
end if
END FUNCTION factrl_s

\[
\text{cumprod(arth(1.0_sp,1.0_sp,NMAX-1))}
\]

By now you should recognize this as an idiom for generating a vector of consecutive factorials. The routines cumprod and arth, both in nrutil, are both capable of being parallelized, e.g., by recursion, so this idiom is potentially faster than an in-line do-loop.
FUNCTION factrl_v(n)
USE nrtype; USE nrutil, ONLY : arth, assert, cumprod
USE nr, ONLY : gammln
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
REAL(SP), DIMENSION(size(n)) :: factrl_v
LOGICAL(LGT), DIMENSION(size(n)) :: mask
INTEGER(I4B), SAVE :: ntop=0
INTEGER(I4B), PARAMETER :: NMAX=32
REAL(SP), DIMENSION(NMAX), SAVE :: a
call assert(all(n >= 0), 'factrl_v arg')
if (ntop == 0) then
  ntop=NMAX
  a(1)=1.0
  a(2:NMAX)=cumprod(arth(1.0_sp,1.0_sp,NMAX-1))
end if
mask = (n >= NMAX)
factrl_v=unpack(exp(gammln(pack(n,mask)+1.0_sp)),mask,0.0_sp)
where (.not. mask) factrl_v=a(n+1)
END FUNCTION factrl_v

unpack(exp(gammln(pack(n,mask)+1.0_sp)),mask,0.0_sp) Here we meet
the first of several solutions to a common problem: How shall we get
answers, from an external vector-valued function, for just a subset
of vector arguments, those defined by a mask? Here we use what we call the “pack-
unpack” solution: Pack up all the arguments using the mask, send them to the
function, and unpack the answers that come back. This packing and unpacking is
not without cost (highly dependent on machine architecture, to be sure), but we hope
to “earn it back” in the parallelism of the external function.

where (.not. mask) factrl_v=a(n+1) In some cases we might take care of the
.not.mask case directly within the unpack construction, using its third (“FIELD=”)
argument to provide the not-unpacked values. However, there is no guarantee that
the compiler won’t evaluate all components of the “FIELD=” array, if it finds it
efficient to do so. Here, since the index of a(n+1) would be out of range, we can’t
do it this way. Thus the separate where statement.

* * *

FUNCTION bico_s(n,k)
USE nrtype
USE nr, ONLY : factln
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n,k
REAL(SP) :: bico_s

Returns the binomial coefficient \(\binom{n}{k}\) as a floating-point number.

bico_s=nint(exp(factln(n)-factln(k)-factln(n-k)))

The nearest-integer function cleans up roundoff error for smaller values of n and k.
END FUNCTION bico_s
FUNCTION bico_v(n,k)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : factln
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: n,k
REAL(SP), DIMENSION(size(n)) :: bico_v
INTEGER(I4B) :: ndum
ndum=assert_eq(size(n),size(k),'bico_v')
bico_v=nint(exp(factln(n)-factln(k)-factln(n-k)))
END FUNCTION bico_v

FUNCTION factln_s(n)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : gammln
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP) :: factln_s
Returns ln(n!).
INTEGER(I4B), PARAMETER :: TMAX=100
REAL(SP), DIMENSION(TMAX), SAVE :: a
LOGICAL(LGT), SAVE :: init=.true.
if (init) then
Initialize the table.
a(1:TMAX)=gammln(arth(1.0_sp,1.0_sp,TMAX))
init=.false.
end if
call assert(n >= 0, 'factln_s arg')
if (n < TMAX) then
In range of the table.
factln_s=a(n+1)
else
Out of range of the table.
factln_s=gammln(n+1.0_sp)
end if
END FUNCTION factln_s

FUNCTION factln_v(n)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : gammln
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: n
REAL(SP), DIMENSION(size(n)) :: factln_v
LOGICAL(LGT), DIMENSION(size(n)) :: mask
INTEGER(I4B), PARAMETER :: TMAX=100
REAL(SP), DIMENSION(TMAX), SAVE :: a
LOGICAL(LGT), SAVE :: init=.true.
if (init) then
a(1:TMAX)=gammln(arth(1.0_sp,1.0_sp,TMAX))
init=.false.
end if
call assert(all(n >= 0), 'factln_v arg')
mask = (n >= TMAX)
factln_v=unpack(gammln(pack(n,mask)+1.0_sp),mask,0.0_sp)
where (.not. mask) factln_v=a(n+1)
END FUNCTION factln_v
Another example of the programming convenience of combining a function returning a vector (here, \texttt{arith}) with a special function whose generic name (here, \texttt{gammln}) has an overloaded vector version.

\begin{verbatim}
FUNCTION beta_s(z,w)
USE nrtype
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: z,w
REAL(SP) :: beta_s
Returns the value of the beta function $B(z,w)$.
beta_s=exp(gammln(z)+gammln(w)-gammln(z+w))
END FUNCTION beta_s

FUNCTION beta_v(z,w)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: z,w
REAL(SP), DIMENSION(size(z)) :: beta_v
INTEGER(I4B) :: ndum
ndum=assert_eq(size(z),size(w),'beta_v')
beta_v=exp(gammln(z)+gammln(w)-gammln(z+w))
END FUNCTION beta_v

FUNCTION gammp_s(a,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : gcf,gser
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,x
REAL(SP) :: gammp_s
Returns the incomplete gamma function $P(a,x)$.
call assert( x >= 0.0, a > 0.0, 'gammp_s args')
if (x<a+1.0_sp) then
  Use the series representation.
  gammp_s=gser(a,x)
else
  Use the continued fraction representation and take its complement.
  gammp_s=1.0_sp-gcf(a,x)
end if
END FUNCTION gammp_s

FUNCTION gammp_v(a,x)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : gcf,gser
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
REAL(SP), DIMENSION(size(x)) :: gammp_v
LOGICAL(LGT), DIMENSION(size(x)) :: mask
INTEGER(I4B) :: ndum
ndum=assert_eq(size(a),size(x),'gammp_v')
call assert( all(x >= 0.0), all(a > 0.0), 'gammp_v args')
mask = (x<a+1.0_sp)
gammp_v=merge(gser(a,merge(x,0.0_sp,.not. mask)), &
  1.0_sp-gcf(a,merge(x,0.0_sp,.not. mask))),mask)
END FUNCTION gammp_v
\end{verbatim}
The generic routine assert in nrutil is overloaded with variants for more than one logical assertion, so you can make more than one assertion about argument ranges.

Here we meet the second solution to the problem of getting masked values from an external vector function. (For the first solution, see note to factrl, above.) We call this one “merge with dummy values”: Inappropriate values of the argument \( x \) (as determined by mask) are set to zero before gser, and later gcf, are called, and the supernumerary answers returned are discarded by a final merge. The assumption here is that the dummy value sent to the function (here, zero) is a special value that computes extremely fast, so that the overhead of computing and returning the supernumerary function values is outweighed by the parallelism achieved on the nontrivial components of \( x \). Look at gser_v and gcf_v below to judge whether this assumption is realistic in this case.

```fortran
FUNCTION gammq_s(a,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : gcf,gser
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,x
REAL(SP) :: gammq_s

Returns the incomplete gamma function \( Q(a, x) \equiv 1 - P(a, x) \).
call assert( x >= 0.0, a > 0.0, 'gammq_s args')
if (x<a+1.0_sp) then
   Use the series representation
   gammq_s=1.0_sp-gser(a,x)
   and take its complement.
else
   Use the continued fraction representation.
end if
END FUNCTION gammq_s

FUNCTION gammq_v(a,x)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : gcf,gser
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
REAL(SP), DIMENSION(size(a)) :: gammq_v
LOGICAL(LGT), DIMENSION(size(x)) :: mask
INTEGER(I4B) :: ndum

ndum=assert_eq(size(a),size(x),'gammq_v')
call assert( all(x >= 0.0), all(a > 0.0), 'gammq_v args')
mask = (x<a+1.0_sp)

gammq_v=merge(1.0_sp-gser(a,merge(x,0.0_sp,mask)), &
gcf(a,merge(x,0.0_sp,.not. mask)),mask)
END FUNCTION gammq_v

FUNCTION gser_s(a,x,gln)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,x
REAL(SP), OPTIONAL, INTENT(OUT) :: gln
REAL(SP) :: gser_s
INTEGER(I4B), PARAMETER :: IMAX=100
REAL(SP), PARAMETER :: EPS=epsilon(x)
```
Returns the incomplete gamma function \( P(a, x) \) evaluated by its series representation as \( \text{gamser} \). Also optionally returns \( \ln \Gamma(a) \) as \( \text{gln} \).

\[
\text{INTEGER}(\text{I4B}) :: n
\]
\[
\text{REAL}(\text{SP}) :: ap, del, summ
\]
if \( (x == 0.0) \) then
  \( \text{gser}_s=0.0 \)
RETURN
end if
ap=a
summ=1.0_{\text{sp}}/a
del=summ
do
  \( n=1,\text{ITMAX} \)
  \( \text{ap}=\text{ap}+1.0_{\text{sp}} \)
  \( \text{del}=\text{del} \cdot x / \text{ap} \)
  \( \text{summ}=\text{summ}+\text{del} \)
  if \( (\text{abs} (\text{del}) < \text{abs} (\text{summ}) \cdot \text{EPS}) \) exit
end do
if \( (n > \text{ITMAX}) \) call \text{nrerror}('a too large, ITMAX too small in \text{gser}_s')
if \( (\text{present} (\text{gln})) \) then
  \( \text{gln} = \text{gammln} (a) \)
  \( \text{gser}_s = \text{summ} \cdot \exp (-x + a \cdot \log (x) - \text{gln}) \)
else
  \( \text{gser}_s = \text{summ} \cdot \exp (-x + a \cdot \log (x) - \text{gammln} (a)) \)
end if
END FUNCTION \( \text{gser}_s \)

FUNCTION \( \text{gser}_v(a, x, gln) \)
USE nrtype; USE nrutil, ONLY : assert_eq, nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: a, x
REAL(SP), DIMENSION(:,), OPTIONAL, INTENT(OUT) :: gln
REAL(SP), DIMENSION(size(a)) :: \( \text{gser}_v \)
INTEGER(\text{I4B}), PARAMETER :: \text{ITMAX}=100
REAL(SP), PARAMETER :: \text{EPS} = epsilon(x)
INTEGER(\text{I4B}) :: n
REAL(SP), DIMENSION(size(a)) :: \text{ap}, \text{del}, \text{summ}
LOGICAL(LGT), DIMENSION(size(a)) :: converged, zero
n=assert_eq(size(a), size(x), '\text{gser}_v')
zero=(x == 0.0)
where (.not. zero) \( \text{gser}_v=0.0 \)
ap=a
summ=1.0_{\text{sp}}/a
del=summ
converged=.false.
do
  \( n=1,\text{ITMAX} \)
  where (.not. converged)
    \( \text{ap}=\text{ap}+1.0_{\text{sp}} \)
    \( \text{del}=\text{del} \cdot x / \text{ap} \)
    \( \text{summ}=\text{summ}+\text{del} \)
  ended do
  converged = (abs (del) < abs (summ) \cdot EPS)
end where
if (all(converged)) exit
end do
if \( (n > \text{ITMAX}) \) call \text{nrerror}('a too large, ITMAX too small in \text{gser}_v')
if \( (\text{present} (\text{gln})) \) then
  gln=gammln(a)
  if (size(gln) < size(a)) call &
    nrerror('gser: Not enough space for gln')
  end if
  where (.not. zero) \( \text{gser}_v=\text{summ} \cdot \exp (-x + a \cdot \log (x) - \text{gln}) \)
else
  where (.not. zero) \( \text{gser}_v=\text{summ} \cdot \exp (-x + a \cdot \log (x) - \text{gammln} (a)) \)
REAL(SP), OPTIONAL, INTENT(OUT) :: gln  Normally, an OPTIONAL argument will be INTENT(IN) and be used to provide a less-often-used extra input argument to a function. Here, the OPTIONAL argument is INTENT(OUT), used to provide a useful value that is a byproduct of the main calculation.

Also note that although $x \geq 0$ is required, we omit our usual call assert check for this, because gser is supposed to be called only by gammp or gammq — and these routines supply the argument checking themselves.

do n=1,ITMAX...end do...if (n > ITMAX)... This is typical code in Fortran 90 for a loop with a maximum number of iterations, relying on Fortran 90's guarantee that the index of the do-loop will be available after normal completion of the loop with a predictable value, greater by one than the upper limit of the loop. If the exit statement within the loop is ever taken, the if statement is guaranteed to fail; if the loop goes all the way through ITMAX cycles, the if statement is guaranteed to succeed.

zero=(x == 0.0)...where (zero) gser_v=0.0...converged=zero  This is the code that provides for very low overhead calculation of zero arguments, as is assumed by the merge-with-dummy-values strategy in gammp and gammq. Zero arguments are “pre-converged” and are never the holdouts in the convergence test.

FUNCTION gcf_s(a,x,gln)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,x
REAL(SP), OPTIONAL, INTENT(OUT) :: gln
REAL(SP) :: gcf_s
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: EPS=epsilon(x),FPMIN=tiny(x)/EPS
Returns the incomplete gamma function \( Q(a, x) \) evaluated by its continued fraction representation as gammcf. Also optionally returns \( \ln \Gamma(a) \) as gln.
Parameters: ITMAX is the maximum allowed number of iterations; EPS is the relative accuracy; FPMIN is a number near the smallest representable floating-point number.
INTEGER(I4B) :: i
REAL(SP) :: an,b,c,d,del,h
if (x == 0.0) then
  gcf_s=1.0
  RETURN
end if
b=x+1.0_sp-a
Set up for evaluating continued fraction by modified Lentz's method ([5.2]) with \( b_0 = 0 \).
c=1.0_sp/FPMIN
Iterate to convergence.
d=1.0_sp/b
h=d

   do i=1,ITMAX
      an=-i*(i-a)
      b=b+2.0_sp
      d=an+d*b

      if (abs(d) < FPMIN) d=FPMIN
      c=b+an/c
      if (abs(c) < FPMIN) c=FPMIN
      end do
   if (abs(h) < FPMIN) h=FPMIN
   gcf_s=exp(1.0_sp)*gammln(a)+h

END FUNCTION gcf_s
d=1.0_sp/d
del=d*c
h=h*del
if (abs(del-1.0_sp) <= EPS) exit
end do
if (i > ITMAX) call nrerror('a too large, ITMAX too small in gcf_s')
if (present(gln)) then
  gln=gammln(a)
  gcf_s=exp(-x+a*log(x)-gln)*h Put factors in front.
else
  gcf_s=exp(-x+a*log(x)-gammln(a))*h
end if
END FUNCTION gcf_s

FUNCTION gcf_v(a,x,gln)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
REAL(SP), DIMENSION(:), OPTIONAL, INTENT(OUT) :: gln
REAL(SP), DIMENSION(size(a)) :: gcf_v
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: EPS=epsilon(x),FPMIN=tiny(x)/EPS
INTEGER(I4B) :: i
REAL(SP), DIMENSION(size(a)) :: an,b,c,d,del,h
LOGICAL(LGT), DIMENSION(size(a)) :: converged,zero
i=assert_eq(size(a),size(x),'gcf_v')
zero=(x == 0.0)
where (zero)
gcf_v=1.0
elsewhere
  b=x+1.0_sp-a
  c=1.0_sp/FPMIN
  d=1.0_sp/b
  h=d
end where
converged=zero
do i=1,ITMAX
  where (.not. converged)
    an=-i*(i-a)
    b=b+2.0_sp
    d=an*d+b
    d=merge(FPMIN,d,abs(d)<FPMIN )
    c=b+an/c
    c=merge(FPMIN,c,abs(c)<FPMIN )
    d=1.0_sp/d
    del=d*c
    h=h*del
    converged = (abs(del-1.0_sp)<=EPS)
  end where
  if (all(converged)) exit
end do
if (i > ITMAX) call nrerror('a too large, ITMAX too small in gcf_v')
if (present(gln)) then
  if (size(gln) < size(a)) call &
  nrerror('gser: Not enough space for gln')
gln=gammln(a)
else
end if
where (.not. zero) gcf_v=exp(-x+a*log(x)-gln)*h
where (.not. zero) gcf_v=exp(-x+a*log(x)-gammln(a))*h
end if
END FUNCTION gcf_v
zero=(x == 0.0)...where (zero) gcf_v=1.0...converged=zero  See note on gser. Here, too, we pre-converge the special value of zero.

* * *

FUNCTION erf_s(x)
USE nrtype
USE nr, ONLY : gammp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: erf_s
  Returns the error function erf(x).
  erf_s=gammp(0.5_sp,x**2)
  if (x < 0.0) erf_s=-erf_s
END FUNCTION erf_s

FUNCTION erf_v(x)
USE nrtype
USE nr, ONLY : gammp
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: erf_v
  erf_v=gammp(spread(0.5_sp,1,size(x)),x**2)
  where (x < 0.0) erf_v=-erf_v
END FUNCTION erf_v

erf_v=gammp(spread(0.5_sp,1,size(x)),x**2)  Yes, we do have an over-
loaded vector version of gammp, but it is vectorized on both its arguments.

Thus, in a case where we want to vectorize on only one argument, we
need a spread construction. In many contexts, Fortran 90 automatically makes
scalars conformable with arrays (i.e., it automatically spreads them to the shape of
the array); but the language does not do so when trying to match a generic function
or subroutine call to a specific overloaded name. Perhaps this is wise; it is safer to
prevent “accidental” invocations of vector-specific functions. Or, perhaps it is an
area where the language could be improved.

FUNCTION erfc_s(x)
USE nrtype
USE nr, ONLY : gammp,gammq
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: erfc_s
  Returns the complementary error function erfc(x).
  erfc_s=merge(1.0_sp+gammp(0.5_sp,x**2),gammq(0.5_sp,x**2), x < 0.0)
END FUNCTION erfc_s

erfc_s=merge(1.0_sp+gammp(0.5_sp,x**2),gammq(0.5_sp,x**2), x < 0.0)  An example of our use of merge as an idiom for a conditional expression.
Once you get used to these, you’ll find them just as clear as the multiline
if...then...else alternative.
FUNCTION erfc_v(x)
USE nrtype
USE nr, ONLY : gammp,gammq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: erfc_v
LOGICAL(LGT), DIMENSION(size(x)) :: mask
mask = (x < 0.0)
erfc_v=merge(1.0_sp+gammp(spread(0.5_sp,1,size(x)), &
merge(x,0.0_sp,mask)**2),gammq(spread(0.5_sp,1,size(x)), &
merge(x,0.0_sp,.not. mask)**2),mask)
END FUNCTION erfc_v

erfc_v=merge(1.0_sp,...) Another example of the “merge with dummy values” idiom described on p. 1090. Here positive values of x in the call to gammp, and negative values in the call to gammq, are first set to the dummy value zero. The value zero is a special argument that computes very fast. The unwanted dummy function values are then discarded by the final outer merge.

⋆⋆⋆

FUNCTION erfcc_s(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: erfcc_s
REAL(SP) :: t,z
REAL(SP), DIMENSION(10) :: coef = (/-1.26551223_sp,1.00002368_sp,&
0.37409196_sp,0.09678418_sp,-0.18628806_sp,0.27886807_sp,&
-1.13520398_sp,1.48851587_sp,-0.82215223_sp,0.17087277_sp/)
z=abs(x)
t=1.0_sp/(1.0_sp+0.5_sp*z)
erfcc_s=t*exp(-z*z+poly(t,coef))
if (x < 0.0) erfcc_s=2.0_sp-erfcc_s
END FUNCTION erfcc_s

FUNCTION erfcc_v(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: erfcc_v,t,z
REAL(SP), DIMENSION(10) :: coef = (/-1.26551223_sp,1.00002368_sp,&
0.37409196_sp,0.09678418_sp,-0.18628806_sp,0.27886807_sp,&
-1.13520398_sp,1.48851587_sp,-0.82215223_sp,0.17087277_sp/)
z=abs(x)
t=1.0_sp/(1.0_sp+0.5_sp*z)
erfcc_v=t*exp(-z*z+poly(t,coef))
where (x < 0.0) erfcc_v=2.0_sp-erfcc_v
END FUNCTION erfcc_v
erfcc_y=t*exp(-z*z+poly(t,coef))  The vector code is identical to the scalar, because the nrutil routine poly has overloaded cases for the evaluation of a polynomial at a single value of the independent variable, and at multiple values. One could also overload a version with a matrix of coefficients whose columns could be used for the simultaneous evaluation of different polynomials at different values of independent variable. The point is that as long as there are differences in the shapes of at least one argument, the intended version of poly can be discerned by the compiler.

```
** ** **

FUNCTION expint(n,x)
USE nrtype; USE nrutil, ONLY : arth,assert,nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: expint
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsil(x),BIG=huge(x)*EPS

Evaluates the exponential integral \( E_n(x) \).

Parameters:
MAXIT is the maximum allowed number of iterations; EPS is the desired relative error, not smaller than the machine precision; BIG is a number near the largest representable floating-point number; EULER (in nrtype) is Euler's constant \( \gamma \).

```

```
INTEGER(I4B) :: i,nm1
REAL(SP) :: a,b,c,d,del,fact,h

if (n == 0) then  Special case.
   expint=exp(-x)/x
   RETURN
end if

nm1=n-1
if (x == 0.0) then  Another special case.
   expint=1.0_sp/nm1
else if (x > 1.0) then  Lentz's algorithm (§5.2).
   b=x+n
   c=BIG
   d=1.0_sp/b
   h=d
   do i=1,MAXIT
      a=-i*(nm1+i)
      b=b+2.0_sp
      d=1.0_sp/(a*d+b)  Denominators cannot be zero.
      c=b/a/c
      del=c*d
      h=h*del
      if (abs(del-1.0_sp) <= EPS) exit
   end do
   if (i > MAXIT) call nrerror('expint: continued fraction failed')
   expint=h*exp(-x)
else  Evaluate series.
   if (nm1 /= 0) then  Set first term.
      expint=1.0_sp/nm1
   else
      expint=-log(x)-EULER
   end if
   fact=1.0
   do i=1,MAXIT
      fact=-fact*x/i
      if (i /= nm1) then
         del=-fact/(i-nm1)
```
else \( \psi(n) \) appears here.
\[
\text{del} = \text{fact} \left( -\log(x) - \text{EULER} + \sum \frac{1}{\text{arth}(1, 1, n) - 1} \right)
\]
end if

\[
\text{expint} = \text{expint} + \text{del}
\]

if (abs(del) < abs(expint) * EPS) exit
\end do

if (i > MAXIT) call nrerror('expint: series failed')
\end if
\end function expint

expint does not readily parallelize, and we thus don't provide a vector version. For syntactic convenience you could make a vector version with a do-loop over calls to this scalar version; or, in Fortran 95, you can of course make the function \texttt{ELEMENTAL}.

\* \* \*

FUNCTION ei(x)
USE nrtype; USE nrutil, ONLY : assert, nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: ei
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x), FPMIN=tiny(x)/EPS

Computes the exponential integral \( \text{Ei}(x) \) for \( x > 0 \).

Parameters:
\( \text{MAXIT} \) is the maximum number of iterations allowed; \( \text{EPS} \) is the relative error, or absolute error near the zero of \( \text{Ei} \) at \( x = 0.3725 \); \( \text{FPMIN} \) is a number near the smallest representable floating-point number; \( \text{EULER} \) (in \texttt{nrtype}) is Euler's constant \( \gamma \).

INTEGER(I4B) :: k
REAL(SP) :: fact, prev, sm, term

\text{call assert}(x > 0.0, 'ei arg')

if (x < FPMIN) then
\text{Special case: avoid failure of convergence test because of underflow.}
\text{ei} = \log(x) + \text{EULER}
\text{else if (x} \leq -\log(EPS)) then
\text{Use power series.}
\text{sm} = 0.0
\text{fact} = 1.0
\text{do} \text{k=1,MAXIT}
\text{fact} = \text{fact} \times x / \text{k}
\text{term} = \text{fact} / \text{k}
\text{sm} = \text{sm} + \text{term}
\text{if (term < EPS*sm) exit}
\text{end do}
\text{if (k} > \text{MAXIT) call nrerror('series failed in ei')}
\text{ei} = \text{exp(x)} \times (1.0 + \text{sm}) / \text{x}
\text{else}
\text{Use asymptotic series. Start with second term.}
\text{sm} = 0.0
\text{term} = 1.0
\text{do} \text{k=1,MAXIT}
\text{prev} = \text{term}
\text{term} = \text{term} \times k / x
\text{if (term} < \text{EPS) exit}
\text{if (term < prev) then}
\text{sm} = \text{sm} + \text{term}
\text{else}
\text{sm} = \text{sm} - \text{prev}
\text{exit}
\text{end if}
\text{end do}
\text{if (k} > \text{MAXIT) call nrerror('asymptotic failed in ei')}
\text{ei} = \text{exp(x)} \times (1.0 + \text{sm}) / \text{x}
\text{end if}
\end function ei
ei does not readily parallelize, and we thus don’t provide a vector version.

For syntactic convenience you could make a vector version with a do-loop over calls to this scalar version; or, in Fortran 95, you can of course make the function \texttt{ELEMENTAL}.

\begin{verbatim}
FUNCTION betai_s(a,b,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : betacf,gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b,x
REAL(SP) :: betai_s
Returns the incomplete beta function \( I_x(a,b) \).
REAL(SP) :: bt
call assert(x >= 0.0, x <= 1.0, 'betai_s arg')
if (x == 0.0 .or. x == 1.0) then
  bt=0.0
else
  bt=exp(gammln(a+b)-gammln(a)-gammln(b)&
    +a*log(x)+b*log(1.0_sp-x))
end if
if (x < (a+1.0_sp)/(a+b+2.0_sp)) then
  betai_s=bt*betacf(a,b,x)/a
else
  betai_s=1.0_sp-bt*betacf(b,a,1.0_sp-x)/b
end if
END FUNCTION betai_s

FUNCTION betai_v(a,b,x)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : betacf,gammln
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,x
REAL(SP), DIMENSION(size(a)) :: betai_v
REAL(SP), DIMENSION(size(a)) :: bt
LOGICAL(LGT), DIMENSION(size(a)) :: mask
INTEGER(I4B) :: ndum
ndum=assert_eq(size(a),size(b),size(x),'betai_v')
call assert(all(x >= 0.0), all(x <= 1.0), 'betai_v arg')
where (x == 0.0 .or. x == 1.0)
  bt=0.0
elsewhere
  bt=exp(gammln(a+b)-gammln(a)-gammln(b)&
    +a*log(x)+b*log(1.0_sp-x))
end where
mask=(x < (a+1.0_sp)/(a+b+2.0_sp))
betai_v=bt*betacf(merge(a,b,mask),merge(b,a,mask),merge(x,1.0_sp-x,mask))/merge(a,b,mask)
where (.not. mask) betai_v=1.0_sp-betai_v
END FUNCTION betai_v
\end{verbatim}
Compare the scalar

\[
\text{if (x < (a+1.0_sp)/(a+b+2.0_sp)) then} \\
\quad \beta_i_s = bt*betacf(a,b,x)/a \\
\text{else} \\
\quad \beta_i_s = 1.0_sp-bt*betacf(b,a,1.0_sp-x)/b \\
\text{end if}
\]

with the vector

\[
\text{mask=(x < (a+1.0_sp)/(a+b+2.0_sp))} \\
\beta_i_v = bt*betacf(merge(a,b,mask),merge(b,a,mask),& \\
\quad \text{merge(x,1.0_sp-x,mask)})/merge(a,b,mask) \\
\text{where (.not. mask) \beta_i_v = 1.0_sp-\beta_i_v}
\]

Here merge is used (several times) to evaluate all the required components in a single call to the vectorized betacf, notwithstanding that some components require one pattern of arguments, some a different pattern.

```fortran
FUNCTION betacf_s(a,b,x)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b,x
REAL(SP) :: betacf_s
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x), FPMIN=tiny(x)/EPS

Used by betaI: Evaluates continued fraction for incomplete beta function by modified
Lentz's method (§5.2).

REAL(SP) :: aa,c,d,del,h,qab,qam,qap
INTEGER(I4B) :: m,m2

qab=a+b  
qam=a-1.0_sp

These q's will be used in factors that occur in the coefficients (6.4.6).

First step of Lentz's method.

d=1.0_sp-qab*x/qap
if (abs(d) < FPMIN) d=FPMIN

One step (the even one) of the recurrence.

d1.0_sp=aa*d
if (abs(d) < FPMIN) d=FPMIN

c=1.0_sp+aa/c
if (abs(c) < FPMIN) c=FPMIN

d=1.0_sp/d
h=h*d*c

Next step of the recurrence (the odd one).

d1.0_sp=aa*d
if (abs(d) < FPMIN) d=FPMIN

c=1.0_sp+aa/c
if (abs(c) < FPMIN) c=FPMIN

d=1.0_sp/d
del=d*c
b=h*del
if (abs(del-1.0_sp) <= EPS) exit
Are we done?
end do
if (m > MAXIT)&
call nrerror('a or b too big, or MAXIT too small in betacf_s')

betacf_s=h
END FUNCTION betacf_s
```
**FUNCTION betacf_v(a,b,x)**

```fortran
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,x
REAL(SP), DIMENSION(size(x)) :: betacf_v
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x), FPMIN=tiny(x)/EPS
REAL(SP), DIMENSION(size(x)) :: aa,c,d,del,h,qab,qam,qap
LOGICAL(LGT), DIMENSION(size(x)) :: converged
INTEGER(I4B), PARAMETER :: m
INTEGER(I4B), DIMENSION(size(x)) :: m2
m=assert_eq(size(a),size(b),size(x),'betacf_v')
qab=a+b
qam=a-1.0_sp
qap=a+1.0_sp
h=1.0
where (abs(h) < FPMIN) h=FPMIN
qab=x/qap
d=1.0_sp/qab
converged=.false.
do m=1,MAXIT
  where (.not. converged)
    m2=2*m
    aa=m*(b-m)*x/((qam+m2)*(a+m2))
    d=1.0_sp+aa*d
    d=merge(FPMIN,d, abs(d)<FPMIN )
    c=1.0_sp+aa/c
    c=merge(FPMIN,c, abs(c)<FPMIN )
  end where
  if (all(converged)) exit
  end do
if (m > MAXIT)&
  call nrerror('a or b too big, or MAXIT too small in betacf_v')
betacf_v=h
END FUNCTION betacf_v
```

The scalar version does this with an `if`. Why does it become a `merge` here in the vector version, rather than a `where`? Because we are already inside a “where (.not.converged)” block, and Fortran 90 doesn’t allow nested `where`’s! (Fortran 95 will allow nested `where`’s.)
FUNCTION bessj0_s(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessj0_s

Returns the Bessel function \( J_0(x) \) for any real \( x \).
REAL(SP) :: ax,xx,z
REAL(DP) :: y
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,-0.1098628627e-2_dp,&
  0.2734510407e-4_dp,-0.207370639e-5_dp,0.2093887211e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.1562499995e-1_dp,&
  0.1430488765e-3_dp,-0.6911147651e-5_dp,0.7621095151e-6_dp,&
  -0.934945152e-7_dp/)
REAL(DP), DIMENSION(6) :: r = (/57568490574.0_dp,-13362590354.0_dp,&
  651619640.7_dp,-11214424.18_dp,77392.33017_dp,&
  -184.39052456_dp/)
REAL(DP), DIMENSION(6) :: s = (/57568490411.0_dp,1029532985.0_dp,&
  9494680.718_dp,59272.64853_dp,267.852712_dp,1.0_dp/)

if (abs(x) < 8.0) then
  Direct rational function fit.
  y=x**2
  bessj0_s=poly(y,r)/poly(y,s)
else
  Fitting function (6.5.9).
  ax=abs(x)
  z=8.0_sp/ax
  y=x**2
  xx=ax-0.785398164_sp
  bessj0_s=sqrt(0.636619772_sp/ax)*(cos(xx)*poly(y,p,-mask)-z*sin(xx)*poly(y,q,-mask))
end if
END FUNCTION bessj0_s

FUNCTION bessj0_v(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessj0_v
REAL(SP), DIMENSION(size(x)) :: ax,xx,z
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,-0.1098628627e-2_dp,&
  0.2734510407e-4_dp,-0.207370639e-5_dp,0.2093887211e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.1562499995e-1_dp,&
  0.1430488765e-3_dp,-0.6911147651e-5_dp,0.7621095151e-6_dp,&
  -0.934945152e-7_dp/)
REAL(DP), DIMENSION(6) :: r = (/57568490574.0_dp,-13362590354.0_dp,&
  651619640.7_dp,-11214424.18_dp,77392.33017_dp,&
  -184.39052456_dp/)
REAL(DP), DIMENSION(6) :: s = (/57568490411.0_dp,1029532985.0_dp,&
  9494680.718_dp,59272.64853_dp,267.852712_dp,1.0_dp/)

mask = (abs(x) < 8.0)
where (mask)
  y=x**2
  bessj0_v=poly(y,r,mask)/poly(y,s,mask)
elsewhere
  ax=abs(x)
  z=8.0_sp/ax
  y=x**2
  xx=ax-0.785398164_sp
  bessj0_v=sqrt(0.636619772_sp/ax)*(cos(xx)*poly(y,p,.not. mask)-z*sin(xx)*poly(y,q,.not. mask))
end where
END FUNCTION bessj0_v
where (mask) \ldots bessj0_v \cdot poly(y,r,mask)/poly(y,s,mask)

Here we meet the third solution to the problem of getting masked values from an external vector function. (For the other two solutions, see notes to factrl, p. 1087, and gammp, p. 1090.) Here we simply evade all responsibility and pass the mask into every routine that is supposed to be masked. Let it be somebody else’s problem! That works here because your hardworking authors have overloaded the nrutil routine poly with a masked vector version. More typically, of course, it becomes your problem, and you have to remember to write masked versions of all the vector routines that you call in this way. (We’ll meet examples of this later.)

FUNCTION bessj0_s(x)
USE nrtype; USE nrutil, ONLY : assert, poly
USE nr, ONLY : bessj0
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessj0_s
Returns the Bessel function $J_0(x)$ for positive $x$.
REAL(SP) :: xx,z
REAL(DP) :: y
We'll accumulate polynomials in double precision.
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,-0.1098628627e-2_dp,&
0.2734510407e-4_dp,-0.2073370639e-5_dp,0.2093887211e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.1562499995e-1_dp,&
0.1430488765e-3_dp,-0.6911147651e-5_dp,0.7621095161e-6_dp,&
-0.934945152e-7_dp/)
REAL(DP), DIMENSION(6) :: r = (/2957821389.0_dp,7062834065.0_dp,&
-512359803.6_dp,10879881.29_dp,-86327.92757_dp,&
228.4622733_dp/)
REAL(DP), DIMENSION(6) :: s = (/40076544269.0_dp,47447.26470_dp,226.1030244_dp,1.0_dp/)
call assert(x > 0.0, ’bessj0_s arg’)
if (abs(x) < 8.0) then
Rational function approximation of (6.5.8).
y = x**2
bessj0_s = poly(y,r)/poly(y,s) +
0.636619772_sp*bessj0(x)*log(x)
else
Fitting function (6.5.10).
z = x**2
y = x**2
xx = 0.785398164_sp
bessj0_s = sqrt(0.636619772_sp/x)*(sin(xx)*
poly(y,p)+z*cos(xx)*poly(y,q))
end if
END FUNCTION bessj0_s

FUNCTION bessj0_v(x)
USE nrtype; USE nrutil, ONLY : assert, poly
USE nr, ONLY : bessj0
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessj0_v
REAL(SP), DIMENSION(size(x)) :: xx,z
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,-0.1098628627e-2_dp,&
0.2734510407e-4_dp,-0.2073370639e-5_dp,0.2093887211e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.1562499995e-1_dp,&
0.1430488765e-3_dp,-0.6911147651e-5_dp,0.7621095161e-6_dp,&
-0.934945152e-7_dp/)
REAL(DP), DIMENSION(6) :: r = (/2957821389.0_dp,7062834065.0_dp,&
-512359803.6_dp,10879881.29_dp,-86327.92757_dp,&
228.4622733_dp/)
REAL(DP), DIMENSION(6) :: s = (/40076544269.0_dp,47447.26470_dp,226.1030244_dp,1.0_dp/)
call assert(x > 0.0, ’bessj0_v arg’)
if (abs(x) < 8.0) then
Rational function approximation of (6.5.8).
y = x**2
bessj0_v = poly(y,r)/poly(y,s) +
0.636619772_sp*bessj0(x)*log(x)
else
Fitting function (6.5.10).
z = x**2
y = x**2
xx = 0.785398164_sp
bessj0_v = sqrt(0.636619772_sp/x)*(sin(xx)*
poly(y,p)+z*cos(xx)*poly(y,q))
end if
END FUNCTION bessj0_v

Here we meet the third solution to the problem of getting masked values from an external vector function. (For the other two solutions, see notes to factrl, p. 1087, and gammp, p. 1090.) Here we simply evade all responsibility and pass the mask into every routine that is supposed to be masked. Let it be somebody else’s problem! That works here because your hardworking authors have overloaded the nrutil routine poly with a masked vector version. More typically, of course, it becomes your problem, and you have to remember to write masked versions of all the vector routines that you call in this way. (We’ll meet examples of this later.)

FUNCTION bessj0_s(x)
USE nrtype; USE nrutil, ONLY : assert, poly
USE nr, ONLY : bessj0
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessj0_s
Returns the Bessel function $Y_0(x)$ for positive $x$.
REAL(SP) :: xx,z
REAL(DP) :: y
We'll accumulate polynomials in double precision.
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,-0.1098628627e-2_dp,&
0.2734510407e-4_dp,-0.2073370639e-5_dp,0.2093887211e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.1562499995e-1_dp,&
0.1430488765e-3_dp,-0.6911147651e-5_dp,0.7621095161e-6_dp,&
-0.934945152e-7_dp/)
REAL(DP), DIMENSION(6) :: r = (/2957821389.0_dp,7062834065.0_dp,&
-512359803.6_dp,10879881.29_dp,-86327.92757_dp,&
228.4622733_dp/)
REAL(DP), DIMENSION(6) :: s = (/40076544269.0_dp,47447.26470_dp,226.1030244_dp,1.0_dp/)
call assert(x > 0.0, ’bessj0_s arg’)
if (abs(x) < 8.0) then
Rational function approximation of (6.5.8).
y = x**2
bessj0_s = poly(y,r)/poly(y,s) +
0.636619772_sp*bessj0(x)*log(x)
else
Fitting function (6.5.10).
z = 8.0_sp/x
y = x**2
xx = 0.785398164_sp
bessj0_s = sqrt(0.636619772_sp/x)*(sin(xx)*
poly(y,p)+z*cos(xx)*poly(y,q))
end if
END FUNCTION bessj0_s

FUNCTION bessj0_v(x)
USE nrtype; USE nrutil, ONLY : assert, poly
USE nr, ONLY : bessj0
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessj0_v
REAL(SP), DIMENSION(size(x)) :: xx,z
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,-0.1098628627e-2_dp,&
0.2734510407e-4_dp,-0.2073370639e-5_dp,0.2093887211e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.1562499995e-1_dp,&
0.1430488765e-3_dp,-0.6911147651e-5_dp,0.7621095161e-6_dp,&
-0.934945152e-7_dp/)
REAL(DP), DIMENSION(6) :: r = (/2957821389.0_dp,7062834065.0_dp,&
-512359803.6_dp,10879881.29_dp,-86327.92757_dp,&
228.4622733_dp/)
REAL(DP), DIMENSION(6) :: s = (/40076544269.0_dp,47447.26470_dp,226.1030244_dp,1.0_dp/)
call assert(x > 0.0, ’bessj0_v arg’)
if (abs(x) < 8.0) then
Rational function approximation of (6.5.8).
y = x**2
bessj0_v = poly(y,r)/poly(y,s) +
0.636619772_sp*bessj0(x)*log(x)
else
Fitting function (6.5.10).
z = 8.0_sp/x
y = x**2
xx = 0.785398164_sp
bessj0_v = sqrt(0.636619772_sp/x)*(sin(xx)*
poly(y,p)+z*cos(xx)*poly(y,q))
end if
END FUNCTION bessj0_v
REAL(DP), DIMENSION(6) :: r = (/2957821389.0_dp,7062834065.0_dp,&
-512359803.6_dp,10879881.29_dp,-86327.92757_dp,&
228.4622733_dp/)
REAL(DP), DIMENSION(6) :: s = (/4007664469.0_dp,745249964.8_dp,&
7198466.438_dp,47447.26470_dp,226.1030244_dp,1.0_dp/)
call assert(all(x > 0.0), 'bessy0_v arg')
mask = (abs(x) < 8.0)
where (mask)
y=x**2
bessy0_v=(poly(y,r,mask)/poly(y,s,mask)) + &
0.636619772_sp*bessj0(x)*log(x)
elsewhere
z=8.0_sp/x
y=z**2
xx=x-0.785398164_sp
bessy0_v=sqrt(0.636619772_sp/x)*(&
poly(y,p,.not. mask)+z*cos(xx)*poly(y,q,.not. mask))
end where
END FUNCTION bessy0_v

 FUNCTION bessj1_s(x)
 USE nrtype; USE nrutil, ONLY : poly
 IMPLICIT NONE
 REAL(SP), INTENT(IN) :: x
 REAL(SP) :: bessj1_s
 Returns the Bessel function \( J_1(x) \) for any real \( x \).
 REAL(SP) :: ax,xx,z
 REAL(DP) :: y
 We'll accumulate polynomials in double precision.
 REAL(DP), DIMENSION(6) :: r = (/72362614232.0_dp,&
-7895955235.0_dp,242396853.1_dp,-2972611.439_dp,&
15704.48260_dp,-30.16036606_dp/) 
 REAL(DP), DIMENSION(6) :: s = (/144725228442.0_dp,2300535178.0_dp,&
18583304.74_dp,99447.43394_dp,376.9991397_dp,1.0_dp/) 
 REAL(DP), DIMENSION(5) :: p = (/1.0_dp,0.183105e-2_dp,&
-0.3516396496e-4_dp,0.2457520174e-5_dp,-0.240337019e-6_dp/) 
 REAL(DP), DIMENSION(5) :: q = (/0.04687499995_dp,&
-0.2002690873e-3_dp,0.8449190996e-5_dp,-0.88228987e-6_dp,&
0.10878412e-6_dp/) 
 if (abs(x) < 8.0) then
 Direct rational approximation.
y=x**2
 bessj1_s=x*(poly(y,r)/poly(y,s))
else
 Fitting function (6.5.9).
 ax=abs(x)
y=x**2
 xx=ax-2.356194491_sp
 bessj1_s=sqrt(0.636619772_sp/x)*(&
poly(y,p)-z*sin(xx)*poly(y,q))*sign(1.0_sp,x)
end if
END FUNCTION bessj1_s
FUNCTION bessj1_v(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), DIMENSION(:, ), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessj1_v
REAL(SP), DIMENSION(size(x)) :: ax, xx, z
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(6) :: r = (/72362614232.0_dp, &
-7895059235.0_dp, 242396853.1_dp, -2972611.439_dp, &
15704.48260_dp, -30.16036606_dp/)
REAL(DP), DIMENSION(6) :: s = (/144725228442.0_dp, 2300535178.0_dp, &
18883304.74_dp, 99447.43394_dp, 376.9991397_dp, 1.0_dp/)
REAL(DP), DIMENSION(5) :: p = (/1.0_dp, 0.183105e-2_dp, &
-0.3516396496e-4_dp, 0.2457520174e-5_dp, -0.240337019e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.04687499995_dp, &
-0.2002690873e-5_dp, 0.8449199096e-6_dp, -0.88228987e-6_dp, &
0.105787412e-6_dp/)
mask = (abs(x) < 8.0)
where (mask)
y = x**2
bessj1_v = x * (poly(y, r, mask) / poly(y, s, mask))
elsewhere
ax = abs(x)
z = 8.0_sp / ax
y = x**2
xx = ax - 2.356194491_sp
bessj1_v = sqrt(0.636619772_sp / ax) * (cos(xx) * &
poly(y, p, not. mask) - z * sin(xx) * poly(y, q, not. mask)) * &
sign(1.0_sp, x)
end where
END FUNCTION bessj1_v

FUNCTION bessy1_s(x)
USE nrtype; USE nrutil, ONLY : assert, poly
USE nr, ONLY : bessj1
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessy1_s
REAL(SP) :: xx, z
REAL(DP), DIMENSION(5) :: p = (/1.0_dp, 0.183105e-2_dp, &
-0.3516396496e-4_dp, 0.2457520174e-5_dp, -0.240337019e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.04687499995_dp, &
-0.2002690873e-5_dp, 0.8449199096e-6_dp, -0.88228987e-6_dp, &
0.105787412e-6_dp/)
REAL(DP), DIMENSION(6) :: r = (/-0.4900604943e13_dp, &
0.1275274390e13_dp, -0.5153438139e11_dp, 0.7349264551e9_dp, &
-0.4237922726e7_dp, 0.05119379354e4_dp/)
REAL(DP), DIMENSION(7) :: s = (/0.0969850890e14_dp, &
0.2424419664e12_dp, 0.3733650367e10_dp, 0.2265904002e8_dp, &
0.1020426050e6_dp, 0.3549632885e3_dp, 1.0_dp/)
call assert(x > 0.0, 'bessy1_s arg')
if (abs(x) < 8.0) then
Rational function approximation of (6.5.8).
y = x**2
bessy1_s = (poly(y, r) / poly(y, s)) * &
0.636619772_sp * (bessj1(x) * log(x) - 1.0_sp / x)
else
Fitting function (6.5.10).
endif
END FUNCTION bessy1_s
FUNCTION bessy1_s(x)
  USE nrtype; USE nrutil, ONLY : assert, poly
  USE nr, ONLY : bessj1
  IMPLICIT NONE
  REAL(SP), DIMENSION(:), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: bessy1_s
  REAL(SP), DIMENSION(size(x)) :: xx,z
  LOGICAL(LGT), DIMENSION(size(x)) :: mask
  REAL(DP), DIMENSION(5) :: p = (/1.0_dp,0.183105e-2_dp,&
        -0.3516396496e-4_dp,0.2457520174e-5_dp,-0.240337019e-6_dp/)
  REAL(DP), DIMENSION(5) :: q = (/0.04687499995_dp,&
        -0.2002690873e-3_dp,0.8449199096e-5_dp,-0.88228987e-6_dp,&
        0.105787412e-6_dp/)
  REAL(DP), DIMENSION(6) :: r = (/-0.4900604943e13_dp,&
        0.1275274390e13_dp,-0.5153438139e11_dp,0.7349264551e9_dp,&
        -0.4257927266e7_dp,0.8511937935e6_dp/)
  REAL(DP), DIMENSION(7) :: s = (/0.2499580570e14_dp,&
        0.424419664e12_dp,0.373365636e10_dp,0.2246904002e8_dp,&
        0.1020426050e6_dp,0.3549632885e3_dp,1.0_dp/)
  call assert(all(x > 0.0), 'bessy1_s arg')
  mask = (abs(x) < 8.0)
  where (mask)
    y=x**2
    bessy1_s=x*(poly(y,r,mask)/poly(y,s,mask))+&
          0.636619772_sp*(bessj1(x)*log(x)-1.0_sp/x)
  else where
    z=8.0_sp/x
    y=z**2
    xx=x-2.356194491_sp
    bessy1_s=sqrt(0.636619772_sp/x)*(&
          poly(y,p)*z*cos(xx)+poly(y,q))
  end where
END FUNCTION bessy1_s

FUNCTION bessy_s(n,x)
  USE nrtype; USE nrutil, ONLY : assert
  USE nr, ONLY : bessy0, bessy1
  IMPLICIT NONE
  INTEGER(I4B), INTENT(IN) :: n
  REAL(SP), INTENT(IN) :: x
  REAL(SP) :: bessy_s
  INTEGER(I4B) :: j
  REAL(SP) :: by,bym,byp,tox
  call assert(n >= 2, x > 0.0, 'bessy_s args')
  tox=2.0_sp/x
  by=bessy1(x)
  bym=bessy0(x)
  do j=1,n-1
    bessy_s=by*byp+&
             tox*(bym-byp)*bessy_s
    bym=byp
    by=bym
  end do
  return
END FUNCTION bessy_s
FUNCTION bessy_v(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessy0,bessy1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessy_v
INTEGER(I4B) :: j
REAL(SP), DIMENSION(size(x)) :: by,bym,byp,tox
call assert(n >= 2, all(x > 0.0), 'bessy_v args')
tox=2.0_sp/x
by=bessy1(x)
bym=bessy0(x)
do j=1,n-1
  byp=j*tox*by-bym
  bym=by
  by=byp
end do
bessy_v=by
END FUNCTION bessy_v

Notice that the vector routine is *exactly* the same as the scalar routine, but operates only on vectors, and that nothing in the routine is specific to any level of precision or kind type of real variable. Cases like this make us wish that Fortran 90 provided for “template” types that could automatically take the type and shape of the actual arguments. (Such facilities are available in other, more object-oriented languages such as C++.)

FUNCTION bessj_s(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessj0,bessj1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessj_s
INTEGER(I4B), PARAMETER :: IACC=40,IEXP=maxexponent(x)/2
Returns the Bessel function $J_n(x)$ for any real $x$ and $n \geq 2$. Make the parameter IACC larger to increase accuracy.
INTEGER(I4B) :: j,jsum,m
REAL(SP) :: ax,bj,bjm,bjp,summ,tox
call assert(n >= 2, 'bessj_s args')
ax=abs(x)
if (ax*ax <= 8.0_sp*tiny(x)) then
  bessj_s=0.0
else if (ax > real(n,sp)) then
  tox=2.0_sp/ax
  bje=bessj0(ax)
  bj=bessj1(ax)
doi j=1,n-1
```fortran
bjp = j*tox*bj - bjm
bjm = bj
bj = bjp
end do
bessj_s = bj
else
  tox = 2.0_sp/ax
  m = 2*((n+int(sqrt(real(IACC*n,sp))))/2)
  bessj_s = 0.0
  jsum = 0
  bjp = 0.0
  bj = 1.0
  do j = m, 1, -1
    bjm = j*tox*bj - bjp
    bj = bjm
    if (exponent(bj) > IEXP) then
      Renormalize to prevent overflows.
      bj = scale(bj, -IEXP)
      bjp = scale(bjp, -IEXP)
      bessj_s = scale(bessj_s, -IEXP)
      summ = scale(summ, -IEXP)
    end if
    if (jsum /= 0) summ = summ + bj
      Accumulate the sum.
      jsum = 1 - jsum
      Change 0 to 1 or vice versa.
    end do
    summ = 2.0_sp*summ - bj
    Compute (5.5.16)
  bessj_s = bessj_s / summ
  end if
if (x < 0.0 .and. mod(n,2) == 1) bessj_s = -bessj_s
END FUNCTION bessj_s
```

The `bessj` routine does not conveniently parallelize with Fortran 90’s language constructions, but Bessel functions are of sufficient importance that we feel the need for a parallel version nevertheless. The basic method adopted below is to encapsulate as contained vector functions two separate algorithms, one for the case \( x \leq n \), the other for \( x > n \). Both of these have masks as input arguments; within each routine, however, they immediately revert to the pack-unpack method. The choice to pack in the subsidiary routines, rather than in the main routine, is arbitrary; the main routine is supposed to be a little clearer this way.

```
if (exponent(bj) > IEXP) then... In the Fortran 77 version of this routine, we scaled the variables by \( 10^{-10} \) whenever \( bj \) was bigger than \( 10^{10} \). On a machine with a large exponent range, we could improve efficiency by scaling less often. In order to remain portable, however, we used the conservative value of \( 10^{10} \). An elegant way of handling renormalization is provided by the Fortran 90 intrinsic functions that manipulate real numbers. We test with if (exponent(bj) > IEXP) and then if necessary renormalize with bj = scale(bj, -IEXP) and similarly for the other variables. Our conservative choice is to set IEXP = maxexponent(x)/2. Note that an added benefit of scaling this way is that only the exponent of each variable is modified; no roundoff error is introduced as it can be if we do a floating-point division instead.
```
FUNCTION bessj_v(n,xx)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessj0,bessj1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(1:) :: xx
REAL(SP), DIMENSION(size(xx)) :: bessj_v
INTEGER(I4B), PARAMETER :: IACC=40, IEXP=maxexponent(xx)/2
REAL(SP), DIMENSION(size(xx)) :: ax
LOGICAL(LGT), DIMENSION(size(xx)) :: mask,mask0
REAL(SP), DIMENSION(1:) :: x,bj,bjm,bjp,summ,tox,bessjle
LOGICAL(LGT), DIMENSION(1) :: renorm
INTEGER(I4B) :: j,jsum,m,npak
call assert(n >= 2, 'bessj_v args')
ax=abs(xx)
mask = (ax <= real(n,sp))
mask0 = (ax*ax <= 8.0_sp*tiny(xx))
bessj_v=bessjle_v(n,ax,logical(mask .and. .not.mask0, kind=lgt))
where (mask0) bessj_v=0.0
where (xx < 0.0 .and. mod(n,2) == 1) bessj_v=-bessj_v
CONTAINS
FUNCTION bessjgt_v(n,xx,mask)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(1:) :: xx
LOGICAL(LGT), DIMENSION(size(xx)), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(xx)) :: bessjgt_v
npak=count(mask)
if (npak == 0) RETURN
allocate(x(npak),bj(npak),bjm(npak),bjp(npak),tox(npak))
x=pack(xx,mask)
tox=2.0_sp/x
bjm=bessj0(x)
bj=bessj1(x)
do j=1,n-1
bjp=j*tox*bj-bjm
bjm=bj
bj=bjp
end do
bessjgt_v=unpack(bj,mask,0.0_sp)
deallocate(x,bj,bjm,bjp)
END FUNCTION bessjgt_v
FUNCTION bessjle_v(n,xx,mask)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(1:) :: xx
LOGICAL(LGT), DIMENSION(size(xx)), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(xx)) :: bessjle_v
npak=count(mask)
if (npak == 0) RETURN
allocate(x(npak),bj(npak),bjm(npak),bjp(npak),summ(npak), &
        bessjle(npak),tox(npak),renorm(npak))
x=pack(xx,mask)
tox=2.0_sp/x
m=2*((n+int(sqrt(real(IACC*n,sp))))/2)
bessjle=0.0
jsum=0
summ=0.0
bjp=0.0
bj=1.0
do j=m,1,-1
bjm=j*tox*bj-bjp
bessjle=0.0
jsum=jsum+bj
bjm=jsum/m
bjp=bjp+bj
bj=bjp/m
end do
bessjle=0.0
jsum=0
bjp=bj
bj=bjm
renorm = (exponent(bj)>IEXP)
bj=merge(scale(bj,-IEXP),bj,renorm)
bjp=merge(scale(bjp,-IEXP),bjp,renorm)
bessjle=merge(scale(bessjle,-IEXP),bessjle,renorm)
summ=merge(scale(summ,-IEXP),summ,renorm)
if (jsum /= 0) summ=summ+bj
jsum=1-jsum
if (j == n) bessjle=bjp
end do
summ=2.0_sp*summ-bj
bessjle=bessjle/summ
bessjle_v=unpack(bessjle,mask,0.0_sp)
deallocate(x,bj,bjm,bjp,summ,bessjle,tox,renorm)
END FUNCTION bessjle_v
END FUNCTION bessj_v

The vector bessj_v is set once (with a mask) and then merged with itself, along with the vector result of the bessjgt_v call. Thus are the two evaluation methods combined. (A third case, where an argument is zero, is then handled by an immediately following where.)

* * *

FUNCTION bessi0_s(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessi0_s
Returns the modified Bessel function $I_0(x)$ for any real $x$.
REAL(SP) :: ax
REAL(DP), DIMENSION(7) :: p = (/1.0_dp,3.5156229_dp,&
3.0899424_dp,1.2067492_dp,0.2659732_dp,0.360768e-1_dp,&
0.45813e-2_dp/)
Accumulate polynomials in double precision.
REAL(DP), DIMENSION(9) :: q = (/0.39894228_dp,0.1328592e-1_dp,&
0.225319e-2_dp,-0.157565e-2_dp,0.916281e-2_dp,&
-0.2057706e-1_dp,0.2635537e-1_dp,-0.1647633e-1_dp,&
0.392377e-2_dp/)
ax=abs(x)
if (ax < 3.75) then
    Polynomial fit.
    bessi0_s=poly(real((x/3.75_sp)**2,dp),p)
else
    bessi0_s=(exp(ax)/sqrt(ax))*poly(real(3.75_sp/ax,dp),q)
end if
END FUNCTION bessi0_s

FUNCTION bessi0_v(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessi0_v
REAL(SP), DIMENSION(size(x)) :: ax
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(7) :: p = (/1.0_dp,3.5156229_dp,&
3.0899424_dp,1.2067492_dp,0.2659732_dp,0.360768e-1_dp,&
REAL(DP), DIMENSION(9) :: q = (/0.39894228_dp,0.1328592e-1_dp,&
0.225319e-2_dp,-0.157565e-2_dp,0.916281e-2_dp,&
-0.2057706e-1_dp,0.2635537e-1_dp,-0.1647633e-1_dp,&
0.392377e-2_dp/)

ax=abs(x)
mask = (ax < 3.75)
where (mask)
   bessi0_v=poly(real((x/3.75_sp)**2,dp),p,mask)
elsewhere
   y=3.75_sp/ax
   bessi0_v=(exp(ax)/sqrt(ax))*poly(real(y,dp),q,.not. mask)
end where

END FUNCTION bessi0_v

FUNCTION bessk0_s(x)
USE nrtype; USE nrutil, ONLY : assert,poly
USE nr, ONLY : bessi0
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessk0_s

Returns the modified Bessel function $K_0(x)$ for positive real $x$.

REAL(DP) :: y
Accumulate polynomials in double precision.
REAL(DP), DIMENSION(7) :: p = (/-0.57721566_dp,0.42278420_dp,&
0.23069756_dp,0.3488590e-1_dp,0.262698e-2_dp,0.10750e-3_dp,&
0.74e-5_dp/)
REAL(DP), DIMENSION(7) :: q = (/1.25331414_dp,-0.7832358e-1_dp,&
0.2189568e-1_dp,-0.1062446e-1_dp,0.587872e-2_dp,&
-0.251540e-2_dp,0.53208e-3_dp/)
call assert(x > 0.0, 'bessk0_s arg')
if (x <= 2.0) then
   Polynomial fit.
   y=x*x/4.0_sp
   bessk0_s=(-log(x/2.0_sp)*bessi0(x))+poly(y,p)
else
   y=(2.0_sp/x)
   bessk0_s=(exp(-x)/sqrt(x))*poly(y,q)
end if
END FUNCTION bessk0_s

FUNCTION bessk0_v(x)
USE nrtype; USE nrutil, ONLY : assert,poly
USE nr, ONLY : bessi0
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessk0_v
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask

REAL(DP), DIMENSION(7) :: p = (/-0.57721566_dp,0.42278420_dp,&
0.23069756_dp,0.3488590e-1_dp,0.262698e-2_dp,0.10750e-3_dp,&
0.74e-5_dp/)
REAL(DP), DIMENSION(7) :: q = (/1.25331414_dp,-0.7832358e-1_dp,&
0.2189568e-1_dp,-0.1062446e-1_dp,0.587872e-2_dp,&
-0.251540e-2_dp,0.53208e-3_dp/)
call assert(all(x > 0.0), 'bessk0_v arg')
mask = (x <= 2.0)
where (mask)
   y=x*x/4.0_sp
   bessk0_v=(-log(x/2.0_sp)*bessi0(x))+poly(y,p,mask)
elsewhere
   y=(2.0_sp/x)
   bessk0_v=(exp(-x)/sqrt(x))\*poly(y, q, .not. mask)
end where
END FUNCTION bessk0_v

FUNCTION bessi1_s(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessi1_s

Returns the modified Bessel function \( I_1(x) \) for any real \( x \).

REAL(SP) :: ax
REAL(DP), DIMENSION(7) :: p = (/0.5_dp, 0.87890594_dp, &
                      0.51498869_dp, 0.15084934_dp, 0.2658733e-1_dp, &
                      0.301532e-2_dp, 0.32411e-3_dp/)
  Accumulate polynomials in double precision.
REAL(DP), DIMENSION(9) :: q = (/0.39894228_dp, -0.3988024e-1_dp, &
                      -0.362018e-2_dp, 0.163801e-2_dp, -0.1031555e-1_dp, &
                      0.2282967e-1_dp, -0.2895312e-1_dp, 0.1787654e-1_dp, &
                      -0.420059e-2_dp/)

ax=abs(x)
if (ax < 3.75) then  Polynomial fit.
   bessi1_s=ax*poly(real((x/3.75_sp)**2, dp), p)
else
   bessi1_s=(exp(ax)/sqrt(ax))*poly(real(3.75_sp/ax, dp), q)
end if
if (x < 0.0) bessi1_s=-bessi1_s
END FUNCTION bessi1_s

FUNCTION bessi1_v(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessi1_v
REAL(SP), DIMENSION(size(x)) :: ax
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(7) :: p = (/0.5_dp, 0.87890594_dp, &
                      0.51498869_dp, 0.15084934_dp, 0.2658733e-1_dp, &
                      0.301532e-2_dp, 0.32411e-3_dp/)
REAL(DP), DIMENSION(9) :: q = (/0.39894228_dp, -0.3988024e-1_dp, &
                      -0.362018e-2_dp, 0.163801e-2_dp, -0.1031555e-1_dp, &
                      0.2282967e-1_dp, -0.2895312e-1_dp, 0.1787654e-1_dp, &
                      -0.420059e-2_dp/)

ax=abs(x)
mask = (ax < 3.75)
where (mask)
   bessi1_v=ax*poly(real((x/3.75_sp)**2, dp), p, mask)
elsewhere
   y=3.75_sp/ax
   bessi1_v=(exp(ax)/sqrt(ax))*poly(real(y, dp), q, .not. mask)
end where
where (x < 0.0) bessi1_v=-bessi1_v
END FUNCTION bessi1_v
FUNCTION bessk1_s(x)
USE nrtype; USE nrutil, ONLY : assert, poly
USE nr, ONLY : bessi1
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessk1_s

Returns the modified Bessel function \( K_1(x) \) for positive real \( x \).

REAL(DP) :: y
Accumulate polynomials in double precision.
REAL(DP), DIMENSION(7) :: p = (/1.0_dp,0.15443144_dp,&
               -0.67278579_dp,-0.18156897_dp,-0.1919402e-1_dp,&
               -0.110404e-2_dp,-0.4686e-4_dp/)
REAL(DP), DIMENSION(7) :: q = (/1.25331414_dp,0.23498619_dp,&
               -0.3655620e-1_dp,0.1504268e-1_dp,-0.780353e-2_dp,&
               0.325614e-2_dp,-0.68245e-3_dp/)
call assert(x > 0.0, 'bessk1_s arg')

if (x <= 2.0) then
    Polynomial fit.
    y=x**4.0_sp
    bessk1_s=(log(x/2.0_sp)*bessi1(x))+(1.0_sp/x)*poly(y,p)
else
    y=2.0_sp/x
    bessk1_s=(exp(-x)/sqrt(x))*poly(y,q)
end if
END FUNCTION bessk1_s

FUNCTION bessk1_v(x)
USE nrtype; USE nrutil, ONLY : assert, poly
USE nr, ONLY : bessi1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessk1_v
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(7) :: p = (/1.0_dp,0.15443144_dp,&
               -0.67278579_dp,-0.18156897_dp,-0.1919402e-1_dp,&
               -0.110404e-2_dp,-0.4686e-4_dp/)
REAL(DP), DIMENSION(7) :: q = (/1.25331414_dp,0.23498619_dp,&
               -0.3655620e-1_dp,0.1504268e-1_dp,-0.780353e-2_dp,&
               0.325614e-2_dp,-0.68245e-3_dp/)
call assert(all(x > 0.0), 'bessk1_v arg')

mask = (x <= 2.0)
where (mask)
    y=x**4.0_sp
    bessk1_v=(log(x/2.0_sp)*bessi1(x))+(1.0_sp/x)*poly(y,p,mask)
elsewhere
    y=2.0_sp/x
    bessk1_v=(exp(-x)/sqrt(x))*poly(y,q,.not. mask)
end where
END FUNCTION bessk1_v
FUNCTION bessk_s(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessk0,bessk1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessk_s

Returns the modified Bessel function \( K_n(x) \) for positive \( x \) and \( n \geq 2 \).

INTEGER(I4B) :: j
REAL(SP) :: bk,bkm,bkp,tox
call assert(n >= 2, x > 0.0, 'bessk_s args')
tox=2.0_sp/x
bkm=bessk0(x) \quad \text{Upward recurrence for all } x \ldots
bk=bessk1(x)
do j=1,n-1 
  bk=bkm+j*tox*bk 
bkm=bk 
bkp=bkm+j*tox*bk
end do
bessk_s=bk
END FUNCTION bessk_s

FUNCTION bessk_v(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessk0,bessk1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessk_v
INTEGER(I4B) :: j
REAL(SP), DIMENSION(size(x)) :: bk,bkm,bkp,tox
call assert(n >= 2, all(x > 0.0), 'bessk_v args')
tox=2.0_sp/x
bkm=bessk0(x)
bk=bessk1(x)
do j=1,n-1 
  bk=bkm+j*tox*bk 
bkm=bk 
bkp=bkm+j*tox*bk
end do
bessk_v=bk
END FUNCTION bessk_v

The scalar and vector versions of \texttt{bessk} are identical, and have no precision-specific constants, another example of where we would like to define a generic "template" function if the language had this facility.

\*
\*
\*
FUNCTION bessi_s(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessi0
IMPLICIT NONE
INTEGER(4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessi_s
INTEGER(4B), PARAMETER :: IACC=40, IEXP=maxexponent(x)/2

Returns the modified Bessel function \( I_n(x) \) for any real \( x \) and \( n \geq 2 \). Make the parameter IACC larger to increase accuracy.

INTEGER(4B) :: j,m
REAL(SP) :: bi,bim,bip,tox

call assert(n \geq 2, 'bessi_s args')
bessi_s=0.0

if (x*x \leq 8.0_sp*tiny(x)) RETURN Underflow limit.
tox=2.0_sp/abs(x)
bip=0.0
bi=1.0

m=2*(n+int(sqrt(real(IACC*n,sp))))

DO j=m,1,-1
  bim=bip+j*tox*bi
  The downward recurrence.
  bi=bim
  bip=bi
  if (exponent(bi) \> IEXP) then Renormalize to prevent overflows.
    bessi_s=scale(bessi_s,-IEXP)
    bi=scale(bi,-IEXP)
    bip=scale(bip,-IEXP)
  end if
  if (j == n) bessi_s=bip
  end do

bessi_s=bessi_s*bessi0(x)/bi Normalize with bessi0.

if (x < 0.0 .and. mod(n,2) == 1) bessi_s=-bessi_s

END FUNCTION bessi_s

FUNCTION bessi_v(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessi0
IMPLICIT NONE
INTEGER(4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessi_v
INTEGER(4B), PARAMETER :: IACC=40, IEXP=maxexponent(x)/2

INTEGER(4B) :: j,m
REAL(SP), DIMENSION(size(x)) :: bi,bim,bip,tox
LOGICAL(LGT), DIMENSION(size(x)) :: mask

call assert(n \geq 2, 'bessi_v args')
bessi_v=0.0

mask = (x \leq 8.0_sp*tiny(x))
tox=2.0_sp/merge(2.0_sp,abs(x),mask)
bip=0.0
bi=1.0_sp

m=2*(n+int(sqrt(real(IACC*n,sp))))

DO j=m,1,-1
  bim=bip+j*tox*bi
  bi=bim
  if (exponent(bi) \> IEXP) then See discussion of scaling for bessj on
    bessi_v=scale(bessi_v,-IEXP)
  end if
  where (exponent(bi) \> IEXP)
  bessi_v=scale(bessi_v,-IEXP)

END FUNCTION bessi_v
bi=scale(bi,-IEXP)
bip=scale(bip,-IEXP)
end where
if (j == n) bessi_v=bip
end do
bessi_v=bessi_v*bessi0(x)/bi
where (mask) bessi_v=0.0_sp
where (x < 0.0 .and. mod(n,2) == 1) bessi_v=-bessi_v
END FUNCTION bessi_v

mask = (x == 0.0)
tox=2.0_sp/merge(2.0_sp,abs(x),mask)

For the special case \( x = 0 \), the value of the returned function should be zero; however, the evaluation of \( \text{tox} \) will give a divide check. We substitute an innocuous value for the zero cases, then fix up their answers at the end.

⋆⋆⋆

SUBROUTINE bessjy_s(x,xnu,rj,ry,rjp,ryp)
USE nrtype; USE nrutil, ONLY : assert,nrerror
USE nr, ONLY : beschb
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,xnu
REAL(SP), INTENT(OUT) :: rj,ry,rjp,ryp
INTEGER(I4B), PARAMETER :: MAXIT=10000
REAL(DP), PARAMETER :: XMIN=2.0_dp,EPS=1.0e-10_dp,FPMIN=1.0e-30_dp

Returns the Bessel functions \( rj = J_\nu \), \( ry = Y_\nu \) and their derivatives \( rjp = J'_\nu \), \( ryp = Y'_\nu \), for positive \( x \) and for \( xnu = \nu \geq 0 \). The relative accuracy is within one or two significant digits of \( \text{EPS} \), except near a zero of one of the functions, where \( \text{EPS} \) controls its absolute accuracy. \( \text{FPMIN} \) is a number close to the machine’s smallest floating-point number. All internal arithmetic is in double precision. To convert the entire routine to double precision, change the \( \text{SP} \) declaration above and decrease \( \text{EPS} \) to \( 10^{-16} \). Also convert the subroutine \( \text{beschb} \).

INTEGER(I4B) :: i,isign,l,nl
REAL(DP) :: a,b,c,d,del,del1,e,f,fact,fact2,fact3,ff,gam,gam1,gam2,&
gammi,gampl,h,p,pimu,pimu2,q,r,rjl,rjl1,rjmu,rjpl,rjtemp,&
rjl1,rjmu,rjmu2,ry1,rymu,rymup,rytemp,sum,sum1,w,x2,xi,xi2,xmu,xmu2
COMPLEX(DPC) :: aa,bb,cc,dd,dl,pq

call assert(x > 0.0, xnu >= 0.0, 'bessjy args')
nl=merge(int(xnu+0.5_dp), max(0,int(xnu-x+1.5_dp)), x < XMIN)
nl is the number of downward recurrences of the \( J \)'s and upward recurrences of \( Y \)'s. \( xmu \) lies between \( -1/2 \) and \( 1/2 \) for \( x < \text{XMIN} \), while it is chosen so that \( x \) is greater than the turning point for \( x \geq \text{XMIN} \).

\[ xmu=xnu-nl \]
\[ xmu2=xmu*xmu \]
\[ xi=1.0_dp/x \]
\[ xi2=2.0_dp*xi \]
\[ w=xi2/PI_D \]
\[ isign=1 \]
\[ h=xnu*xi \]
\[ if (h < \text{FPMIN}) h=\text{FPMIN} \]
\[ b=xi2*xmu \]
\[ d=0.0 \]
\[ c=h \]
do i=1,MAXIT
\[ b=b+xi2 \]
\[ d=d-b \]
\[ if (abs(d) < \text{FPMIN}) d=\text{FPMIN} \]
\[ c=b-1.0_dp/c \]
if (abs(c) < FPMIN) c=FPMIN
    d=1.0_dp/d
    del=c*d
    h=del*h
if (d < 0.0) isign=-isign
if (abs(del-1.0_dp) < EPS) exit
end do
if (i > MAXIT) call nrerror('x too large in bessjy; try asymptotic expansion')
    rjl=isign*FPMIN
    Initialize $J_\nu$ and $J'_\nu$ for downward recurrence.
    rjpl=h*rjl
    Store values for later rescaling.
    rjl1=rjl
    rjpl1=rjpl
    fact=xmu*xi
    do l=nl,1,-1
        rjtemp=fact*rjl+rjpl
        fact=fact-xi
        rjpl=fact*rjtemp-rjl
        rjl=rjtemp
    end do
    if (rjl == 0.0) rjl=EPS
    f=rjpl/rjl
Now have unnormalized $J_\mu$ and $J'_\mu$.
if (x < XMIN) then
    Use series.
    x2=0.5_dp*x
    pimu=PI_D*xmu
    if (abs(pimu) < EPS) then
        fact=1.0
    else
        fact=pimu/sin(pimu)
    end if
    d=-log(x2)
    e=xmu*d
    if (abs(e) < EPS) then
        fact2=1.0
    else
        fact2=sinh(e)/e
    end if
    call beschb(xmu,gam1,gam2,gampl,gammi)
    Chebyshev evaluation of $\Gamma_1$ and $\Gamma_2$.
    ff=2.0_dp/PI_D*fact*(gam1*cosh(e)+gam2*fact2*d)
    e=exp(e)
    p=e/(gampl*PI_D)
    q=1.0_dp/(e*PI_D*gammi)
    if (abs(pimu2) < EPS) then
        fact3=1.0
    else
        fact3=sin(pimu2)/pimu2
    end if
    r=PI_D*pimu2*fact3*fact3
    c=1.0
    d=-x2*x2
    sum=ff*p*r
    sum1=p
    do i=1,MAXIT
        ff=(i*ff+p+q)/(i*i-xmu2)
        c=c*d/i
        p=p/(i-xmu)
        q=q/(i+xmu)
        del=c*(ff+p*r)
        sum=sum+del
        del1=c*p-i*del
        sum1=sum1+del1
        if (abs(del) < (1.0_dp+abs(sum))*EPS) exit
    end do
if (i > MAXIT) call nrerror('bessy series failed to converge')
rymu = -sum
ry1 = sum1 * xi2
rymup = xmu * xi * rymu - ry1
rjmu = sqrt(w / (rymup - f * rymu))

else

  a = 0.25_dp - xmu2
  pq = cmplx(-0.5_dp * xi, 1.0_dp, kind=dpc)
  aa = cmplx(0.0_dp, xi * a, kind=dpc)
  bb = cmplx(2.0_dp * x, 2.0_dp, kind=dpc)
  cc = bb + aa / pq
  dd = 1.0_dp / bb
  pq = cc / dd * pq
  do i = 2, MAXIT
    a = a + 2 * (i - 1)
    bb = bb + cmplx(0.0_dp, 2.0_dp, kind=dpc)
    dd = a * dd + bb
    if (absc(dd) < FPMIN) dd = FPMIN
    cc = bb / cc
    if (absc(cc) < FPMIN) cc = FPMIN
    dd = 1.0_dp / dd
    dl = cc / dd
    if (absc(dl - 1.0_dp) < EPS) exit
  end do
  if (i > MAXIT) call nrerror('cf2 failed in bessjy')
  p = real(pq)
  q = aimag(pq)
  rjmu = sqrt(w / ((p - f) * gam + q))
  rjmu = sign(rjmu, rjl)
  rymu = rjmu * gam
  rymup = rymu * (p + q / gam)
  ry1 = xmu * xi * rymu - rymup
end if

f = sqrt((w / ((p - f) * gam + q))
  rjmu = sqrt(w / ((p - f) * gam + q))
  rjmu = sign(rjmu, rjl)
  rj = rjl * fact
  rjp = rjp1 * fact
  do i = 1, nl
    rytemp = (xmu + i) * xi2 * ry1 - rymu
    rymu = ry1
    ry1 = rytemp
  end do
  ry = rymu
  ryp = xmu * xi * rymu - ry1

CONTAINS

FUNCTION absc(z)
IMPLICIT NONE
COMPLEX(DPC), INTENT(IN) :: z
REAL(DP) :: absc
absc = abs(real(z)) + abs(aimag(z))
END FUNCTION absc
END SUBROUTINE bessjy_s

Yes there is a vector version bessjy_v. Its general scheme is to have a bunch of contained functions for various cases, and then combine their outputs (somewhat like bessj_v, above, but much more complicated). A listing runs to about four printed pages, and we judge it to be of not much interest, so we will not include it here. (It is included on the machine-readable media.)

*  *  *
SUBROUTINE beschb_s(x,gam1,gam2,gampl,gammi)
USE nrtype
USE nr, ONLY : chebev
IMPLICIT NONE
REAL(DP), INTENT(IN) :: x
REAL(DP), INTENT(OUT) :: gam1,gam2,gampl,gammi
INTEGER(IA4), PARAMETER :: NUSE1=5,NUSE2=5
Evaluates $\Gamma_1$ and $\Gamma_2$ by Chebyshev expansion for $|x| \leq 1/2$. Also returns $1/\Gamma(1+x)$ and $1/\Gamma(1-x)$. If converting to double precision, set NUSE1 = 7, NUSE2 = 8.
REAL(SP) :: xx
REAL(SP), DIMENSION(7) :: c1=(-1.142022680371168_sp,&
 6.5165112670737e-3_sp,3.087090173086e-4_sp,-3.4706269649e-6_sp,&
 6.9437664e-9_sp,3.67795e-11_sp,-1.356e-13_sp/)
REAL(SP), DIMENSION(8) :: c2=(/1.843740587300905_sp,&
 -7.6852840847867e-2_sp,1.2719271366546e-3_sp,&
 -4.9717367042e-6_sp,-3.1261198e-8_sp,2.423096e-10_sp,&
 -1.702e-13_sp,-1.49e-15_sp/)
xx=8.0_dp*x*x-1.0_dp      ! Multiply x by 2 to make range be -1 to 1, and then apply
gam1=chebev(-1.0_sp,1.0_sp,c1(1:NUSE1),xx)  ! transformation for evaluating even Cheby-
gam2=chebev(-1.0_sp,1.0_sp,c2(1:NUSE2),xx)  ! shev series.
gampl=gam2-x*gam1
gammi=gam2+x*gam1
END SUBROUTINE beschb_s

SUBROUTINE beschb_v(x,gam1,gam2,gampl,gammi)
USE nrtype
USE nr, ONLY : chebev
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: x
REAL(DP), DIMENSION(:), INTENT(OUT) :: gam1,gam2,gampl,gammi
INTEGER(I4B), PARAMETER :: NUSE1=5,NUSE2=5
REAL(SP), DIMENSION(size(x)) :: xx
REAL(SP), DIMENSION(7) :: c1=(-1.142022680371168_sp,&
 6.5165112670737e-3_sp,3.087090173086e-4_sp,-3.4706269649e-6_sp,&
 6.9437664e-9_sp,3.67795e-11_sp,-1.356e-13_sp/)
REAL(SP), DIMENSION(8) :: c2=(/1.843740587300905_sp,&
 -7.6852840847867e-2_sp,1.2719271366546e-3_sp,&
 -4.9717367042e-6_sp,-3.1261198e-8_sp,2.423096e-10_sp,&
 -1.702e-13_sp,-1.49e-15_sp/)
xx=8.0_dp*x*x-1.0_dp      ! Multiply x by 2 to make range be -1 to 1, and then apply
gam1=chebev(-1.0_sp,1.0_sp,c1(1:NUSE1),xx)
gam2=chebev(-1.0_sp,1.0_sp,c2(1:NUSE2),xx)
gampl=gam2-x*gam1
gammi=gam2+x*gam1
END SUBROUTINE beschb_v

SUBROUTINE bessik(x,xnu,ri,rk,rip,rkp)
USE nrtype; USE nrutil, ONLY : assert,nrerror
USE nr, ONLY : beschb
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,xnu
REAL(SP), INTENT(OUT) :: ri,rk,rip,rkp
INTEGER(IA4), PARAMETER :: MAXIT=10000
REAL(SP), PARAMETER :: XMIN=2.0
REAL(DP), PARAMETER :: EPS=1.0e-10_dp,FPMIN=1.0e-30_dp

Returns the modified Bessel functions $r_i = I_i(x)$ and $K_i = K_i(x)$, and their derivatives $r_i' = I_i'(x)$ and $K_i' = K_i'(x)$, for positive $x$ and for $xnu = \nu \geq 0$. The relative accuracy is within one or
two significant digits of EPS. FPMIN is a number close to the machine’s smallest floating-point number. All internal arithmetic is in double precision. To convert the entire routine to double precision, change the REAL declaration above and decrease EPS to $10^{-16}$. Also convert the subroutine bescb.

REAL(DP) :: a,al,b,c,d,del,del1,delh,dels,e,f,fact,fact2,ff,
gam1,gam2,gammi,gampl,h,pimu,q,q1,q2,qnew,&
ril,rill,rimu,ripl,ritemp,rlk1,rlkmu,rlkmp,rltemp,&
s,sum,sum1,x2,x1,x12,xmu,xmu2

INTEGER(4B) :: i,l,nl

call assert(x > 0.0, xnu >= 0.0, ‘bessik args’)

nl=int(xnu+0.5_dp)

xmu=xmu-nl

xmu2=xmu*xmu

xi=1.0_dp/x

x12=2.0_dp*x

h=xmu*x

if (h < FPMIN) h=FPMIN

b=x12*xmu

d=0.0

c=h

do i=1,MAXIT

b=b+xi2

d=1.0_dp/(b+d)

c=b+1.0_dp/c

del=c*d

h=del*h

if (abs(del-1.0_dp) < EPS) exit

do l=nl,1,-1

ritemp=fact*ril+ripl

fact=fact-xi

ripl=fact*ritemp+ril

ril=ritemp

end do

f=ripl/ril

if (x < XMIN) then

Use series.

x2=0.5_dp*x

pmu=PI_D*xmu

if (abs(pimu) < EPS) then

fact=1.0

else

fact=pimu/sin(pimu)

end if

d=-log(x2)

e=xmu*d

if (abs(e) < EPS) then

fact2=1.0

else

fact2=sinh(e)/e

end if

c=1.0

d=x2*x2

end if

call bescb(xmu,gam1,gam2,gampl,gammi)

ff=fact*(gam1*cosh(e)+gam2*fact2*e)

sum=ff

e=exp(e)

p=0.5_dp*e/gampl

q=0.5_dp/(e*gammi)

c=1.0

d=x2*x2

Chapter B6. Special Functions

Evaluate CF1 by modified Lentz’s method (§5.2).

Denominators cannot be zero here, so no need for special precautions.

Now have unnormalized $I_\mu$ and $I'_\mu$.

Use series.

Chebyshev evaluation of $\Gamma_1$ and $\Gamma_2$. $f_0$. $p_0$. $q_0$. 

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. ... or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).
sum1=p
do i=1,MAXIT
  ff=(i*ff+p+q)/(i*i-xmu2)
c=c*d/i
  p=p/(i-xmu)
  q=q/(i+xmu)
del=c*ff
  sum=sum+del
del1=c*(p-i*ff)
sum1=sum1+del1
  if (abs(del) < abs(sum)*EPS) exit
end do
if (i > MAXIT) call nrerror('bessk series failed to converge')
rkmu=sum
rk1=sum1*x12
else
  Evaluate CF2 by Steed's algorithm (§5.2), which is OK because there can be no
  zero denominators.
b=2.0_dp*(1.0_dp+x)
d=1.0_dp/b
delh=d
h=delh
q1=0.0
q2=1.0
a1=0.25_dp-xmu2
c=a1
q=q/c
da=a1
s=1.0_dp*q+delh
do i=2,MAXIT
  a=a-2*(i-1)
c=-a*c/i
  qnew=(q1-b*q2)/a
  q1=q2
  q2=qnew
  b=b+2.0_dp
d=1.0_dp/(b*a*d)
delh=(b*d-1.0_dp)*delh
  h=h+delh
  dels=q*delh
  s=s+dels
  if (abs(dels/s) < EPS) exit
end do
if (i > MAXIT) call nrerror('bessik: failure to converge in cf2')
h=a1*h
rkmu=sqrt(PI_D/(2.0_dp*x))*exp(-x)/s
rk1=rkmu*(xmu+x+0.5_dp-h)*xi
end if
rktemp=(xmu+i)*xi2*rk1+rkmu
rkmu=rk1
rk1=rktemp
do i=1,nl
  Upward recurrence of K ν.
  rktemp=(xmu+i)*xi2*rk1+rkmu
  rkmu=rk1
  rk1=rktemp
end do
rk=rkmu
rkp=xnu*xi*rkmu-rk1
END SUBROUTINE bessik
bessik does not readily parallelize, and we thus don’t provide a vector version. Since airy, immediately following, requires bessik, we don’t have a vector version of it, either.

* * *

SUBROUTINE airy(x, ai, bi, aip, bip)
USE nrtype
USE nr, ONLY : bessik, bessjy
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: ai, bi, aip, bip

Returns Airy functions $\text{Ai}(x)$, $\text{Bi}(x)$, and their derivatives $\text{Ai}'(x)$, $\text{Bi}'(x)$.

REAL(SP) :: absx, ri, rip, rj, rjp, rk, rkp, rootx, ry, ryp, z
REAL(SP), PARAMETER :: THIRD=1.0_sp/3.0_sp, TWOTHIR=2.0_sp/3.0_sp, &
ONOVRT=0.5773502691896258_sp

absx=abs(x)
rootx=sqrt(absx)
z=TWOTHIR*absx*rootx
if (x > 0.0) then
  call bessik(z, THIRD, ri, rk, rip, rkp)
  ai=rootx*ONOVRT*rk/PI
  bi=rootx*(rk/PI+2.0_sp*ONOVRT*ri)
else if (x < 0.0) then
  call bessjy(z, THIRD, rj, ry, rjp, ryp)
  ai=0.5_sp*rootx*(rj-ONOVRT*ry)
  bi=-0.5_sp*rootx*(ry+ONOVRT*rj)
else
  Case $x = 0$.
  ai=0.3550280538878172_sp
  bi=ai/ONOVRT
  aip=-0.2588194037928068_sp
  bip=-aip/ONOVRT
end if
END SUBROUTINE airy

* * *

SUBROUTINE sphbes_s(n, x, sj, sy, sjp, syp)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessjy
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: sj, sy, sjp, syp

Returns spherical Bessel functions $j_n(x)$, $y_n(x)$, and their derivatives $j'_n(x)$, $y'_n(x)$ for integer $n \geq 0$ and $x > 0$.

REAL(SP), PARAMETER :: RTPIO2=1.253314137135500_sp
REAL(SP) :: order, rj, rjp, ry, ryp

call assert(n >= 0, x > 0.0, 'sphbes_s args')
order=n+0.5_sp
call bessjy(x, order, rj, rjp, ry, ryp)
factor=RTPIO2/sqrt(x)
sj=factor*rj
sy=factor*ry
**Note that sphbes_v uses (through overloading) bessjy_v. The listing of that routine was omitted above, but it is on the machine-readable media.**
end if
end if
END FUNCTION plgndr_s

FUNCTION plgndr_v(l,m,x)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
INTEGER(16), INTENT(IN) :: l,m
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: plgndr_v
INTEGER(16) :: ll
REAL(SP), DIMENSION(size(x)) :: pll,pmm,pmmp1,somx2
call assert(m >= 0, m <= l, all(abs(x) <= 1.0), 'plgndr_v args')
pmm=1.0
if (m > 0) then
  somx2=sqrt((1.0_sp-x)*(1.0_sp+x))
  pmm=product(arth(1.0_sp,2.0_sp,m))*somx2**m
  if (mod(m,2) == 1) pmm=-pmm
end if
if (l == m) then
  plgndr_v=pmm
else
  pmmp1=x*(2*m+1)*pmm
  if (l == m+1) then
    plgndr_v=pmmp1
  else
    do ll=m+2,l
      pll=(x*(2*ll-1)*pmmp1-(ll+m-1)*pmm)/(ll-m)
      pmm=pmmp1
      pmmp1=pll
    end do
    plgndr_v=pll
  end if
end if
END FUNCTION plgndr_v

All those if’s (not where’s) may strike you as odd in a vector routine,
but it is vectorized only on $x$, the dependent variable, not on the scalar
indices $l$ and $m$. Much harder to write a routine that is parallel for a
vector of arbitrary triplets $(l,m,x)$. Try it!

*  *  *

SUBROUTINE frenel(x,s,c)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: s,c
INTEGER(16), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x),FMIN=tiny(x),BIG=huge(x)*EPS,&
 XMIN=1.5
Computes the Fresnel integrals $S(x)$ and $C(x)$ for all real $x$.
Parameters: MAXIT is the maximum number of iterations allowed; EPS is the relative error;
FMIN is a number near the smallest representable floating-point number; BIG is a number
near the machine overflow limit; $XMIN$ is the dividing line between using the series and continued fraction.

```fortran
INTEGER(I4B) :: k, n
REAL(SP) :: a, ax, fact, pix2, sign, sum, sumc, sums, term, test
COMPLEX(SPC) :: b, cc, d, h, del, cs
LOGICAL(LGT) :: odd

ax=abs(x)
if (ax < sqrt(FPMIN)) then
  s=0.0
  c=ax
else if (ax <= XMIN) then
  Evaluate both series simultaneously.
  sum=0.0
  sums=0.0
  sumc=ax
  sign=1.0
  fact=PIO2*ax*ax
  odd=.true.
  term=ax
  n=3
  do k=1,MAXIT
    term=term*fact/k
    sum=sum+sign*term/n
    test=abs(sum)*EPS
    if (odd) then
      sign=-sign
      sums=sum
      sum=sumc
    else
      sumc=sum
      sum=sums
    end if
    if (term < test) exit
    odd=.not. odd
    n=n+2
  end do
  if (k > MAXIT) call nrerror('frenel: series failed')
  s=sums
  c=sumc
else
  Evaluate continued fraction by modified Lentz's method (§5.2).
  pix2=PI*ax*ax
  b=cmplx(1.0_sp,-pix2,kind=spc)
  cc=BIG
  d=1.0_sp/b
  h=d
  n=-1
  do k=2,MAXIT
    n=n+2
    a=-n*(n+1)
    b=b+4.0_sp
    d=1.0_sp/(a*d+b)
    cc=b+a/cc
    del=cc*d
    h=h*del
    if (absc(del-1.0_sp) <= EPS) exit
  end do
  if (k > MAXIT) call nrerror('cf failed in frenel')
  h=star(b)*cc
  cs=cmplx(0.5_sp,0.5_sp,kind=spc)*h
  c=real(cs)
  s=aimag(cs)
end if
if (x < 0.0) then
  Use antisymmetry.
  c=-c
end if
```

Special case: avoid failure of convergence test because of underflow.
s=-s
end if

CONTAINS

FUNCTION absc(z)
IMPLICIT NONE
COMPLEX(SPC), INTENT(IN) :: z
REAL(SP) :: absc
absc=abs(real(z))+abs(aimag(z))
END FUNCTION absc
END SUBROUTINE frenel

It's a good idea always to include the kind= parameter when you use the cmplx intrinsic. The reason is that, perhaps counterintuitively, the result of cmplx is not determined by the kind of its arguments, but is rather the “default complex kind.” Since that default may not be what you think it is (or what spc is defined to be), the desired kind should be specified explicitly.

And why not specify a kind= parameter here, where it is also optionally allowed? Our answer is that the real intrinsic actually merges two different usages. When its argument is complex, it is the counterpart of aimag and returns a value whose kind is determined by the kind of its argument. In fact aimag doesn’t even allow an optional kind parameter, so we never put one in the corresponding use of real. The other usage of real is for “casting,” that is, converting one real type to another (e.g., double precision to single precision, or vice versa). Here we always include a kind parameter, since otherwise the result is the default real kind, with the same dangers mentioned in the previous paragraph.

⋆⋆⋆

SUBROUTINE cisi(x,ci,si)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: ci,si
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x),FPMIN=4.0_sp*tiny(x),&
BIG=huge(x)*EPS,TMIN=2.0

Computes the cosine and sine integrals \( Ci(x) \) and \( Si(x) \). \( Ci(0) \) is returned as a large negative number and no error message is generated. For \( x < 0 \) the routine returns \( Ci(-x) \) and you must supply the \(-\pi i\) yourself.

Parameters: MAXIT is the maximum number of iterations allowed; EPS is the relative error, or absolute error near a zero of \( Ci(x) \); FPMIN is a number near the smallest representable floating-point number; BIG is a number near the machine overflow limit; TMIN is the dividing line between using the series and continued fraction; EULER = \( \gamma \) (in nrtype).

INTEGER(I4B) :: i,k
REAL(SP) :: a,err,fact,sign,sum,sumc,sums,t,term
COMPLEX(SPC) :: h,b,c,d,del
LOGICAL(LGT) :: odd

if (t == 0.0) then
   \textbf{Special case.}
   si=0.0
   ci=-BIG
   RETURN
end if
if (t > TMIN) then
   b=cmplx(1.0_sp,-pix2,kind=spc)

   Evaluate continued fraction by modified Lentz’s method (§5.2).
c = BIG

d = 1.0_sp/b

h = d

do i = 2, MAXIT
    a = -(i-1)**2
    b = b + 2.0_sp
    d = 1.0_sp/(a*d+b)
    c = b*a/c
    del = c - d
    h = h*del
    if (absc(del-1.0_sp) <= EPS) exit
end do

if (i > MAXIT) call nrerror('continued fraction failed in cisi')

h = cmplx(cos(t), -sin(t), kind=spc)*h

si = P122*aimag(h)

else

    if (t < sqrt(FPMIN)) then
        Special case: avoid failure of convergence test
        because of underflow.
    else
        sum = 0.0
        sumc = 0.0
        sign = 1.0
        fact = 1.0
        odd = .true.
        do k = 1, MAXIT
            fact = fact*t/k
            term = fact/k
            sum = sum + sign*term
            err = term/abs(sum)
            if (odd) then
                sign = -sign
                sums = sum
                sum = sumc
            else
                sumc = sum
                sum = sums
            end if
            if (err < EPS) exit
            odd = .not. odd
        end do
        if (k > MAXIT) call nrerror('MAXIT exceeded in cisi')
    end if

    ci = sumc + log(t) + EULER

    if (x < 0.0) si = -si

CONTAINS

FUNCTION absc(z)
IMPLICIT NONE
COMPLEX(SPC), INTENT(IN) :: z
REAL(SP) :: absc
absc = abs(real(z)) + abs(aimag(z))
END FUNCTION absc
END SUBROUTINE cisi

*  *  *
FUNCTION dawson_s(x)
USE nrtype; USE nrutil, ONLY : arth, geop
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: dawson_s

Returns Dawson's integral \( F(x) = e^{-x^2} \int_0^x e^{t^2} dt \) for any real \( x \).

INTEGER(I4B), PARAMETER :: NMAX=6
REAL(SP), PARAMETER :: H=0.4_sp, A1=2.0_sp/3.0_sp, A2=0.4_sp,&
A3=2.0_sp/7.0_sp
INTEGER(I4B) :: i, n0
REAL(SP) :: ec, x2, xp, xx
REAL(SP), DIMENSION(NMAX) :: d1, d2, e1

if (c(1) == 0.0) c(1:NMAX)=exp((-arth(1,2,NMAX)*H)**2)

Initialize c on first call.

if (abs(x) < 0.2_sp) then
  Use series expansion.
  x2=x**2
  dawson_s=x*(1.0_sp-A1*x2*(1.0_sp-A2*x2*(1.0_sp-A3*x2))
else
  Use sampling theorem representation.
  xx=abs(x)
n0=2*nint(0.5_sp*xx/H)
xp=xx-real(n0,sp)*H
e1=geop(ec, ec**2, NMAX)

  dawson_s=0.5641895835477563_sp*sign(exp(-xp**2),x)*&
             Constant is \( \frac{1}{\sqrt{\pi}} \).
  sum(c*(e1/d1+1.0_sp/(d2*e1)))
end if

END FUNCTION dawson_s

REAL(SP), DIMENSION(NMAX), SAVE :: c=/(0.0_sp,i=1,NMAX)/
This is one way to give initial values to an array. Actually, we're somewhat
nervous about using the "implied do-loop" form of the array constructor,
as above, because our parallel compilers might not always be smart enough to execute
the constructor in parallel. In this case, with NMAX=6, the damage potential is quite
minimal. An alternative way to initialize the array would be with a data statement,
"DATA c/NMAX*0.0_sp/"; however, this is not considered good Fortran 90 style, and
there is no reason to think that it would be faster.

FUNCTION dawson_v(x)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: dawson_v
INTEGER(I4B), PARAMETER :: NMAX=6

real(n0+1,2,NMAX) ... arth(n0-1,-2,NMAX) ... geop(ec, ec**2, NMAX) These are not
just notationally convenient for generating the sequences \((n_0 + 1, n_0 + 3, n_0 + 5, \ldots),
(n_0 - 1, n_0 - 3, n_0 - 5, \ldots),\) and \((ec, ec^3, ec^5, \ldots)\). They also may allow parallelization
with parallel versions of arth and geop, such as those in nrutil.

FUNCTION dawson_v(x)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: dawson_v
INTEGER(I4B), PARAMETER :: NMAX=6
REAL(SP), PARAMETER :: H=0.4_sp, A1=2.0_sp/3.0_sp, A2=0.4_sp, A3=2.0_sp/7.0_sp
INTEGER(I4B) :: i,n
REAL(SP), DIMENSION(size(x)) :: x2
REAL(SP), DIMENSION(NMAX), SAVE :: c=(/ (0.0_sp, i=1,NMAX) /)
LOGICAL(LGT), DIMENSION(size(x)) :: mask
if (c(1) == 0.0) c(1:NMAX)=exp(-((arth(1,2,NMAX)*H)**2))
mask = (abs(x) >= 0.2_sp)
dawson_v=dawsonseries_v(x,mask)
where (.not. mask)
  x2=x**2
  dawson_v=x*(1.0_sp-A1*x2*(1.0_sp-A2*x2*(1.0_sp-A3*x2)))
end where
CONTAINS
FUNCTION dawsonseries_v(xin,mask)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xin
LOGICAL(LGT), DIMENSION(size(xin)), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(xin)) :: dawsonseries_v
INTEGER(I4B), DIMENSION(:), ALLOCATABLE :: n0
REAL(SP), DIMENSION(:), ALLOCATABLE :: d1,d2,e1,e2,sm,xp,xx,x
n=count(mask)
if (n == 0) RETURN
allocate(n0(n),d1(n),d2(n),e1(n),e2(n),sm(n),xp(n),xx(n),x(n))
x=pack(xin,mask)
xx=abs(x)
n0=2*nint(0.5_sp*xx/H)
xp=xx-real(n0,sp)*H
e1=exp(2.0_sp*xp*H)
e2=e1**2
d1=n0+1.0_sp
d2=d1-2.0_sp
sm=0.0
do i=1,NMAX
  sm=sm+c(i)*(e1/d1+1.0_sp/(d2*e1))
  d1=d1+2.0_sp
  d2=d2-2.0_sp
  e1=e2*e1
end do
sm=0.5641895835477563_sp*sign(exp(-xp**2),x)*sm
dawsonseries_v=unpack(sm,mask,0.0_sp)
deallocate(n0,d1,d2,e1,e2,sm,xp,xx)
END FUNCTION dawsonseries_v
END FUNCTION dawson_v

FUNCTION rf_s(x,y,z)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y,z
REAL(SP) :: rf_s
REAL(SP), PARAMETER :: ERRTOL=0.08_sp,TINY=1.5e-38_sp,BIG=3.0e37_sp,&
THIRD=1.0_sp/3.0_sp,

dawson_v=dawsonseries_v(x,mask)

Pass-the-buck method for getting masked values, see note to bessj0_v above, p. 1102. Within the contained dawsonseries, we use the pack-unpack method. Note that, unlike in dawson_s, the sums are done by do-loops, because the parallelization is already over the components of the vector argument.
C1=1.0_sp/24.0_sp,C2=0.1_sp,C3=3.0_sp/44.0_sp,C4=1.0_sp/14.0_sp
Computes Carlson’s elliptic integral of the first kind, \( R_F(x, y, z) \). \( x, y, \) and \( z \) must be nonnegative, and at most one can be zero. TINY must be at least 5 times the machine underflow limit, \( \text{BIG} \) at most one-fifth the machine overflow limit.
REAL(SP) :: alamb, ave, delx, dely, delz, e2, e3, sqrtx, sqrty, sqrtz, xt, yt, zt
call assert(min(x,y,z) >= 0.0, min(x+y,x+z,y+z) >= TINY, &
max(x,y,z) <= BIG, 'rf_s args')
xt=x
yt=y
zt=z
do
  sqrtx=sqrt(xt)
sqrtz=sqrt(zt)
  alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
xt=0.25_sp*(xt+alamb)
yt=0.25_sp*(yt+alamb)
zt=0.25_sp*(zt+alamb)
ave=THIRD*(xt+yt+zt)
delx=(ave-xt)/ave
dely=(ave-yt)/ave
delz=(ave-zt)/ave
  if (max(abs(delx),abs(dely),abs(delz)) <= ERRTOL) exit
end do
e2=delx*dely-delz**2
e3=delx*dely*delz
rf_s=(1.0_sp+(C1*e2-C2-C3*e3)*e2+C4*e3)/sqrt(ave)
END FUNCTION rf_s

FUNCTION rf_v(x,y,z)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z
REAL(SP), DIMENSION(size(x)) :: rf_v
REAL(SP), PARAMETER :: ERRTOL=0.08_sp,TINY=1.5e-38_sp,BIG=3.0e37_sp,&
THIRD=1.0_sp/3.0_sp,&
C1=1.0_sp/24.0_sp,C2=0.1_sp,C3=3.0_sp/44.0_sp,C4=1.0_sp/14.0_sp
REAL(SP), DIMENSION(size(x)) :: alamb, ave, delx, dely, delz, e2, e3,
  sqrtx, sqrty, sqrtz, xt, yt, zt
LOGICAL(LGT), DIMENSION(size(x)) :: converged
INTEGER(I4B) :: ndum
ndum=assert_eq(size(x),size(y),size(z),'rf_v')
call assert(all(min(x,y,z) >= 0.0), all(min(x+y,x+z,y+z) >= TINY), &
all(max(x,y,z) <= BIG, 'rf_v args')
xt=x
yt=y
zt=z
converged=.false.
do
  where (.not. converged)
    sqrtx=sqrt(xt)
sqrtz=sqrt(zt)
    alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
xt=0.25_sp*(xt+alamb)
yt=0.25_sp*(yt+alamb)
zt=0.25_sp*(zt+alamb)
ave=THIRD*(xt+yt+zt)
delx=(ave-xt)/ave
dely=(ave-yt)/ave
delz=(ave-zt)/ave
  converged = (max(abs(delx),abs(dely),abs(delz)) <= ERRTOL)
end do

FUNCTION rf_v(x,y,z)
FUNCTION rd_s(x,y,z)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y,z
REAL(SP) :: rd_s
REAL(SP), PARAMETER :: ERRTOL=0.05_sp,TINY=1.0e-25_sp,BIG=4.5e21_sp,&
C1=3.0_sp/14.0_sp,C2=1.0_sp/6.0_sp,C3=9.0_sp/22.0_sp,&
C4=3.0_sp/26.0_sp,C5=0.25_sp*C3,C6=1.5_sp*C4

Computes Carlson's elliptic integral of the second kind, \( R_D(x, y, z) \). \( x \) and \( y \) must be nonnegative, and at most one can be zero. \( z \) must be positive. \( TINY \) must be at least twice the negative 2/3 power of the machine overflow limit. \( BIG \) must be at most \( 0.1 \times \) ERRTOL times the negative 2/3 power of the machine underflow limit.

REAL(SP) :: alamb,ave,delx,dely,delz,ea,eb,ec,ed,&
ee,fac,sqrtx,sqty,sqrtz,sum,xt,yt,zt
call assert(min(x,y) >= 0.0, min(x+y,z) >= TINY, max(x,y,z) <= BIG, &
'rd_s args')

xt=x
ty=y
zt=z
sum=0.0
fac=1.0
do
  sqrtx=sqrt(xt)
sqty=sqrt(yt)
sqrtz=sqrt(zt)
alamb=sqrtx*(sqty+sqrtz)+sqty*sqrtz
  sum=sum+fac/(sqrtz*(zt+alamb))
  fac=0.25_sp*fac
  xt=0.25_sp*(xt+alamb)
yt=0.25_sp*(yt+alamb)
  zt=0.25_sp*(zt+alamb)
  ave=0.2_sp*(xt+yt+3.0_sp*zt)
delx=(ave-xt)/ave
dely=(ave-yt)/ave
delz=(ave-zt)/ave
  if (max(abs(delx),abs(dely),abs(delz)) <= ERRTOL) exit
  ea=delx*dely
eb=delz*delz
  ec=ea+eb
  ee=ed+ec
  rd_s=3.0_sp*sum+fac*(1.0_sp+ed*(-C1+5*ed-C5*delz*ee)&
  +delz*(C2+delz*(-C3+ec+delz*C4*ea)))/(ave*sqrt(ave))
end do
END FUNCTION rd_s
FUNCTION rd_v(x,y,z)
USE nrtype; USE nrutil, ONLY : assert, assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y,z
REAL(SP), DIMENSION(size(x)) :: rd_v
REAL(SP), PARAMETER :: ERRTOL=0.05_sp, TINY=0.5e-16_sp, BIG=450.0e21_sp,
C1=3.0_sp/26.0_sp, C2=1.0_sp/6.0_sp, C3=9.0_sp/22.0_sp, C4=3.0_sp/26.0_sp,
C5=0.5_sp*C3, C6=1.5_sp*C4
REAL(SP), DIMENSION(size(x)) :: alamb, ave, delx, dely, delz, ea, eb, ec, ed, ee, fac, sqrtx, sqrty, sqrtz, sum, xt, yt, zt
LOGICAL(LGT), DIMENSION(size(x)) :: converged
INTEGER(I4B) :: ndum
ndum=assert_eq(size(x),size(y),size(z),'rd_v')
call assert(all(min(x,y) >= 0.0), all(min(x+y,z) >= TINY), &
all(max(x,y,z) <= BIG), 'rd_v args')
xt=x
yt=y
zt=z
sum=0.0
fac=1.0
converged=.false.
do
where (.not. converged)
  sqrtx=sqrt(xt)
sqrty=sqrt(yt)
sqrtz=sqrt(zt)
alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
sum=sum+fac/(sqrtz*(zt+alamb))
fac=0.25_sp*fac
xt=0.25_sp*(xt+alamb)
yt=0.25_sp*(yt+alamb)
zt=0.25_sp*(zt+alamb)
ave=0.2_sp*(xt+y+3.0_sp*zt)
delx=(ave-xt)/ave
dely=(ave-yt)/ave
delz=(ave-zt)/ave
converged = (all(max(abs(delx),abs(dely),abs(delz)) <= ERRTOL))
end where
if (all(converged)) exit
end do
ea=delx+dely
eb=delz+delt
fc=ea-eb
ed=ea-6.0_sp*eb
ee=ed+fc
rd_v=3.0_sp*sum+fac*(1.0_sp+ed*(-C1+C5*ed+C6*delz*ee)&
  +delz*(C2+ee+delz*(-C3*ec+delz*C4*ea)))/(ave*sqrt(ave))
END FUNCTION rd_v

FUNCTION rj_s(x,y,z,p)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : rc, rf
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y,z,p
REAL(SP) :: rj_s
REAL(SP), PARAMETER :: ERRTOL=0.05_sp, TINY=0.5e-16_sp, BIG=450.0e21_sp,
C1=3.0_sp/26.0_sp, C2=1.0_sp/6.0_sp, C3=9.0_sp/22.0_sp, C4=3.0_sp/26.0_sp,
C5=0.5_sp*C3, C6=1.5_sp*C4
Computes Carlson's elliptic integral of the third kind, \( R_J(x,y,z,p) \). x, y, and z must be nonnegative, and at most one can be zero. p must be nonzero. If p < 0, the Cauchy
principal value is returned. TINY must be at least twice the cube root of the machine underflow limit, BIG at most one-fifth the cube root of the machine overflow limit.

REAL(SP) :: a, alamb, alpha, ave, b, bet, delx, dely, delz, ea, eb, ec, ee, fac, pt, rho, sqrtx, sqrtty, sqrtz, &
sm, tau, xt, yt, zt
call assert(min(x,y,z) >= 0.0, min(x+y+z, y+z, abs(p)) >= TINY, &
max(x,y,z, abs(p)) <= BIG, 'rj_s args')

sm=0.0
fac=1.0
if (p > 0.0) then
xt=x
yt=y
zt=z
pt=p
else
xt=min(x,y,z)
yt=max(x,y,z)
z=0.5*(x+y+z-xt-zt)
a=1.0_sp/(yt-p)
b=a*(zt-yt)*(yt-xt)
pt=yt+b
rho=xt*zt/yt
tau=p*pt/yt
end if
do
sqrtx=sqrt(xt)
sqrty=sqrt(yt)
sqrtz=sqrt(zt)
alamb=sqrt(xt)*(sqrty+sqrtz)+sqrty*sqrtz
alpha=(pt*(sqrtx+sqrty+sqrtz)+sqrtx*sqrty*sqrtz)**2
bet=pt*(pt+alamb)**2
sm=sm+fac*rc(alpha, bet)
fac=0.25_sp*fac
xt=0.25_sp*(xt+alamb)
yt=0.25_sp*(yt+alamb)
z=0.25_sp*(z+alamb)
pt=0.25_sp*(pt+alamb)
ave=(0.2_sp*(xt+yt+zt+pt)+pt)
delix=(ave-xt)/ave
dely=(ave-yt)/ave
delz=(ave-zt)/ave
delp=(ave-pt)/ave
if (max(abs(delx), abs(dely), abs(delz), abs(delp)) <= ERRTOL) exit
end do
ea=delx*(dely+delz)+dely*delz
eb=delx*dely*delz
ec=delp**2
ed=ea-0.0_sp*ec
ee=eb-0.0_sp*delx
rj_s=3.0_sp*sm*fac*1.0_sp*ed*(-C1+C5*ed-C6*ee)+eb*(C7+delp*(-C8 &
+delp*C4))+(C2-delp*C3)-C2*delx)+pt/ave**2
if (p <= 0.0) rj_s=a*(b*rj_s+3.0_sp*(rc(rho,tau)-rf(xt,yt,zt)))
end do
end if
FUNCTION rj_v(x,y,z,p)
USE nrtype; USE nrutil, ONLY : assert, assert_eq
USE nr, ONLY : rc, rf
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: x, y, z, p
REAL(SP), DIMENSION(size(x)) :: rj_v
REAL(SP), PARAMETER :: ERRTOL=0.05_sp, TINY=2.5e-13_sp, BIG=9.0e11_sp,&
C1=3.0_sp/14.0_sp, C2=1.0_sp/3.0_sp, C3=3.0_sp/22.0_sp,&
C4=3.0_sp/26.0_sp,C5=0.75_sp*C3,C6=1.5_sp*C4,C7=0.5_sp*C2,&
C8=C3+C3
REAL(SP), DIMENSION(size(x)) :: a,alamb, alpha,ave,b,bet,delp,delx,&
dely,delz,ea,eb,ec,ed,ee,fac,pt,rho,sqrtx,sqrty,sqrtz,&
sm,tau,xt,yt,zt
LOGICAL(LGT), DIMENSION(size(x)) :: mask
INTEGER(I4B) :: ndum
ndum=assert_eq(size(x),size(y),size(z),size(p),'rj_v')
call assert(all(min(x,y,z) >= 0.0), all(min(x+y,x+z,y+z,abs(p)) >= TINY), &
all(max(x,y,z,abs(p)) <= BIG), 'rj_v args')
sm=0.0
fac=1.0
where (p > 0.0)
x=x
y=y
z=z
pt=p
elsewhere
xt=min(x,y,z)
yt=x+y+z-xt-zt
zt=max(x,y,z)
a=1.0_sp/(yt-pt)
b=a*(zt-yt)*(yt-xt)
pt=yt+b
rho=xt*zt/yt
tau=p*pt/yt
end where
mask=.false.
do
where (.not. mask)
sqrtx=sqrt(xt)
sqrty=sqrt(yt)
sqrtz=sqrt(zt)
alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
alpha=(pt*(sqrtx+sqrty+sqrtz)+sqrtx*sqrty*sqrtz)**2
bet=pt*(pt+alamb)**2
sm=sm+fac*rc(alpha,bet)
fac=0.25_sp*fac
xt=0.25_sp*(xt+alamb)
yt=0.25_sp*(yt+alamb)
zt=0.25_sp*(zt+alamb)
pt=0.25_sp*(pt+alamb)
ave=0.2_sp*(xt+yt+zt+pt+pt)
delx=(ave-xt)/ave
dely=(ave-yt)/ave
delz=(ave-zt)/ave
delp=(ave-pt)/ave
if (max(abs(delx),abs(dely),abs(delz),abs(delp)) <= ERRTOL) exit
end where
if (all(mask)) exit
end do
ea=delx*(delx+delx)+delx+delx
ec=delx+delx
ee=eb+2.0_sp*delp*(ea-ec)
rj_v=3.0_sp*sm*fac*(1.0_sp+ed+(-C1*C5+ed-C6+ee)*eb+(C7+delp*(-C8&
+delp*C4))+delp*(C2+delp*C3)-C2+delp)*/ (ave*sqrt(ave))
mask = (p <= 0.0)
where (mask) rj_v=aa*(b-rj_v+&
unpack(3.0_sp*(rc(pack(rho,mask),pack(tau,mask))-&
rf(pack(xt,mask),pack(yt,mask),pack(zt,mask))),mask,0.0_sp))
END FUNCTION rj_v
If you're willing to put up with fairly unreadable code, you can use the pack-unpack trick (for getting a masked subset of components out of a vector function) right in-line, as here. Of course the "outer level" that is seen by the enclosing where construction has to contain only objects that have the same shape as the mask that goes with the where. Because it is so hard to read, we don't like to do this very often. An alternative would be to use CONTAINS to incorporate short, masked "wrapper functions" for the functions used in this way.

FUNCTION rc_s(x,y)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y
REAL(SP), PARAMETER :: ERRTOL=0.04_sp,TINY=1.69e-38_sp,&
SQRTNY=1.3e-19_sp,BIG=3.0e37_sp,TNBG=TINY*BIG,&
COMP1=2.236_sp/SQRTNY,COMP2=TNBG/TNBG/25.0_sp,&
THIRD=1.0_sp/3.0_sp,&
C1=0.3_sp,C2=1.0_sp/7.0_sp,C3=0.375_sp,C4=9.0_sp/22.0_sp
Computes Carlson's degenerate elliptic integral, \( RC(x,y) \). \( x \) must be nonnegative and \( y \) must be nonzero. If \( y \) < 0, the Cauchy principal value is returned. TINY must be at least 5 times the machine underflow limit, BIG at most one-fifth the machine maximum overflow limit.
REAL(SP) :: alamb,ave,s,w,xt,yt
call assert( (/x >= 0.0,y /= 0.0,x+abs(y) >= TINY,x+abs(y) <= BIG, &
y >= -COMP1 .or. x <= 0.0 .or. x >= COMP2/),'rc_s')
if (y > 0.0) then
   xt=x
   yt=y
   w=1.0
else
   xt=x-y
   yt=-y
   w=sqrt(x)/sqrt(xt)
end if
do
   alamb=2.0_sp*sqrt(xt)*sqrt(yt)+yt
   xt=0.25_sp*(xt+alamb)
   yt=0.25_sp*(yt+alamb)
   ave=THIRD*(xt+yt+yt)
   s=(yt-ave)/ave
   if (abs(s) <= ERRTOL) exit
end do
rc_s=w*(1.0_sp+s*s*(C1+s*(C2+s*(C3+s*C4))))/sqrt(ave)
END FUNCTION rc_s

FUNCTION rc_v(x,y)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y
REAL(SP), DIMENSION(size(x)) :: rc_v
REAL(SP), PARAMETER :: ERRTOL=0.04_sp,TINY=1.69e-38_sp,&
SQRTNY=1.3e-19_sp,BIG=3.0e37_sp,TNBG=TINY*BIG,&
COMP1=2.236_sp/SQRTNY,COMP2=TNBG/TNBG/25.0_sp,&
THIRD=1.0_sp/3.0_sp,&
C1=0.3_sp,C2=1.0_sp/7.0_sp,C3=0.375_sp,C4=9.0_sp/22.0_sp
REAL(SP), DIMENSION(size(x)) :: alamb,ave,s,w,xt,yt
LOGICAL(LGT), DIMENSION(size(x)) :: converged
INTEGER(14B) :: ndum
ndum=assert_eq(size(x),size(y),'rc_v')
call assert( (/all(x .ge. 0.0),all(y .ne. 0.0),all(x+abs(y) .ge. TINY), &
all(x+abs(y) .le. BIG),all(y .ge. -COMP1 .or. x .le. 0.0 &
.or. x .ge. COMP2) ),'rc_v')
where (y > 0.0)
x=xt
y=yt
w=1.0
elsewhere
xt=x-y
yt=-y
w=sqrt(x)/sqrt(xt)
end where
converged=.false.
do
where (.not. converged)
alamb=2.0_sp*sqrt(xt)*sqrt(yt)+yt
xt=0.25_sp*(xt+alamb)
yt=0.25_sp*(yt+alamb)
av=THIRD*(xt+yt+yt)
s=(yt-ave)/ave
converged = (abs(s) <= ERRTOL)
end where
if (all(converged)) exit
end do
rc_v=w*(1.0_sp+s*s*(C1+s*(C2+s*(C3+s*C4))))/sqrt(ave)
END FUNCTION rc_v

FUNCTION ellf_s(phi,ak)
USE nrtype
USE nr, ONLY : rf
IMPLICIT NONE
REAL(SP), INTENT(IN) :: phi,ak
REAL(SP) :: ellf_s
Legendre elliptic integral of the 1st kind \( F(\phi, k) \), evaluated using Carlson's function \( R_F \).
The argument ranges are \( 0 \leq \phi \leq \pi/2, 0 \leq k \sin \phi \leq 1 \).
REAL(SP) :: s
s=sin(phi)
ellf_s=s*rf(cos(phi)**2,(1.0_sp-s*ak)*(1.0_sp+s*ak),1.0_sp)
END FUNCTION ellf_s

FUNCTION ellf_v(phi,ak)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : rf
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: phi,ak
REAL(SP), DIMENSION(size(phi)) :: ellf_v
REAL(SP), DIMENSION(size(phi)) :: s
INTEGER(I4B) :: ndum
ndum=assert_eq(size(phi),size(ak),'ellf_v')
s=sin(phi)
ellf_v=s*rf(cos(phi)**2,(1.0_sp-s*ak)*(1.0_sp+s*ak), &
spread(1.0_sp,1,size(phi)))
END FUNCTION ellf_v
FUNCTION elle_s(phi,ak)
USE nrtype
USE nr, ONLY : rd,rf
IMPLICIT NONE
REAL(SP), INTENT(IN) :: phi,ak
REAL(SP) :: elle_s

Legendre elliptic integral of the 2nd kind $E(\phi,k)$, evaluated using Carlson’s functions $R_D$ and $R_F$. The argument ranges are $0 \leq \phi \leq \pi/2$, $0 \leq k \sin \phi \leq 1$.

REAL(SP) :: cc,q,s
s=sin(phi)
c=cos(phi)**2
q=(1.0_sp-s*ak)*(1.0_sp+s*ak)
elle_s=s*(rd(cc,q,1.0_sp)-(s*ak)**2)*rd(cc,q,1.0_sp)/3.0_sp)
END FUNCTION elle_s

FUNCTION elle_v(phi,ak)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : rd,rf
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: phi,ak
REAL(SP), DIMENSION(size(phi)) :: elle_v
REAL(SP), DIMENSION(size(phi)) :: cc,q,s
INTEGER(I4B) :: ndum
ndum=assert_eq(size(phi),size(ak),'elle_v')

s=sin(phi)
c=cos(phi)**2
q=(1.0_sp-s*ak)*(1.0_sp+s*ak)
elle_v=s*(rf(cc,q,spread(1.0_sp,1,size(phi)))-(s*ak)**2)*rd(cc,q,spread(1.0_sp,1,size(phi)))/3.0_sp)
END FUNCTION elle_v

FUNCTION ellpi_s(phi,en,ak)
USE nrtype
USE nr, ONLY : rf,rj
IMPLICIT NONE
REAL(SP), INTENT(IN) :: phi,en,ak
REAL(SP) :: ellpi_s

Legendre elliptic integral of the 3rd kind $\Pi(\phi,n,k)$, evaluated using Carlson’s functions $R_J$ and $R_F$. (Note that the sign convention on $n$ is opposite that of Abramowitz and Stegun.)

The ranges of $\phi$ and $k$ are $0 \leq \phi \leq \pi/2$, $0 \leq k \sin \phi \leq 1$.

REAL(SP) :: cc,enss,q,s
s=sin(phi)
enss=en*s*s
c=cos(phi)**2
q=(1.0_sp-s*ak)*(1.0_sp+s*ak)
ellpi_s=s*(rj(cc,q,spread(1.0_sp,1,size(phi)))-enss*rj(cc,q,1.0_sp,1.0_sp+enss))/3.0_sp)
END FUNCTION ellpi_s

f90
rd(cc,q,spread(1.0_sp,1,size(phi)))) See note to erf_v, p. 1094 above.

FUNCTION ellpi_s(phi,en,ak)
USE nrtype
USE nr, ONLY : rf,rj
IMPLICIT NONE
REAL(SP), INTENT(IN) :: phi,en,ak
REAL(SP) :: ellpi_s

Legendre elliptic integral of the 3rd kind $\Pi(\phi,n,k)$, evaluated using Carlson’s functions $R_J$ and $R_F$. (Note that the sign convention on $n$ is opposite that of Abramowitz and Stegun.)

The ranges of $\phi$ and $k$ are $0 \leq \phi \leq \pi/2$, $0 \leq k \sin \phi \leq 1$.

REAL(SP) :: cc,enss,q,s
s=sin(phi)
enss=en*s*s
c=cos(phi)**2
q=(1.0_sp-s*ak)*(1.0_sp+s*ak)
ellpi_s=s*(rj(cc,q,spread(1.0_sp,1,size(phi)))-enss*rj(cc,q,1.0_sp,1.0_sp+enss))/3.0_sp)
END FUNCTION ellpi_s
FUNCTION ellpi_v(phi, en, ak)
USE nr
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: phi, en, ak
REAL(SP), DIMENSION(size(phi)) :: ellpi_v
REAL(SP), DIMENSION(size(phi)) :: cc, enss, q, s
s=sin(phi)
enss=en*s*s
cc=cos(phi)**2
q=(1.0_sp-s*ak)*(1.0_sp+s*ak)
ellpi_v=s*(rf(cc,q,spread(1.0_sp,1,size(phi)))-enss*&
 &rj(cc,q,spread(1.0_sp,1,size(phi)),1.0_sp+enss)/3.0_sp)
END FUNCTION ellpi_v

SUBROUTINE sncndn(uu, emmc, sn, cn, dn)
USE nr
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: uu, emmc
REAL(SP), INTENT(OUT) :: sn, cn, dn
Returns the Jacobian elliptic functions sn(u, k), cn(u, k), and dn(u, k). Here uu = u,
while emmc = k^2.
REAL(SP), PARAMETER :: CA=0.0003_sp The accuracy is the square of CA.
INTEGER(I4B), PARAMETER :: MAXIT=13
INTEGER(I4B) :: i, ii, l
REAL(SP) :: a, b, c, d, emc, u
REAL(SP), DIMENSION(MAXIT) :: em, en
LOGICAL(LGT) :: bo
emc=emmc
u=uu
if (emc /= 0.0) then
  bo=(emc < 0.0)
  if (bo) then
    d=1.0_sp-emc
    emc=-emc/d
    d=sqrt(d)
    u=d*u
  end if
  a=1.0
  dn=1.0
  do i=1, MAXIT
    l=i
    em(l)=a
    emc=sqrt(emc)
    en(l)=emc
c=0.5_sp*(a+emc)
    if (abs(a-emc) <= CA*a) exit
    emc=a*emc
    a=c
  end do
  if (i > MAXIT) call nrerror('sncndn: convergence failed')
u=c*u
sn=sin(u)
cn=cos(u)
if (sn /= 0.0) then
  a=cn/sn
  c=a*cn
do ii=1, 1, -1
  b=em(ii)
\begin{verbatim}

a=c*a
c=dn*c
dn=(en(ii)+a)/(b+a)
a=c/b
end do
a=1.0_sp/sqrt(c**2+1.0_sp)
sn=sign(a,sn)
cn=c*sn
end if
if (bo) then
  a=dn
dn=cn
cn=a
sn=sn/d
end if
else
  cn=1.0_sp/cosh(u)
dn=cn
sn=tanh(u)
end if

END SUBROUTINE sncndn

* * *

MODULE hypgeo_info
USE nrtype
COMPLEX(SPC) :: hypgeo_aa,hypgeo_bb,hypgeo_cc,hypgeo_dz,hypgeo_z0
END MODULE hypgeo_info

FUNCTION hypgeo(a,b,c,z)
USE nrtype
USE hypgeo_info
USE nr, ONLY : bsstep,hypdrv,hypser,odeint
IMPLICIT NONE
COMPLEX(SPC), INTENT(IN) :: a,b,c,z
COMPLEX(SPC) :: hypgeo
REAL(SP), PARAMETER :: EPS=1.0e-6_sp

Complex hypergeometric function \( \, _2F_1 \) for complex \( \, a, b, c \), and \( \, z \), by direct integration of the hypergeometric equation in the complex plane. The branch cut is taken to lie along the real axis, \( \, \text{Re} \, z > 1 \).
Parameter: \( \, \text{EPS} \) is an accuracy parameter.

COMPLEX(SPC), DIMENSION(2) :: y
REAL(SP), DIMENSION(4) :: ry
if (real(z)**2+aimag(z)**2 <= 0.25) then
  Use series...
call hypser(a,b,c,z,hypgeo,y(2))
RETURN
else if (real(z) < 0.0) then
  ...or pick a starting point for the path integration.
  hypgeo_z0=complx(-0.5_sp,0.0_sp,kind=spc)
else if (real(z) <= 1.0) then
  hypgeo_z0=complx(0.5_sp,0.0_sp,kind=spc)
else
  hypgeo_z0=complx(0.0_sp,sign(0.5_sp,aimag(z)),kind=spc)
end if
hypgeo_aa=a
hypgeo_bb=b
hypgeo_cc=c
hypgeo_dz=z-hypgeo_z0
call hypser(hypgeo_aa,hypgeo_bb,hypgeo_cc,hypgeo_z0,y(1),y(2))
Get starting function and derivative.
ry(1:4:2)=real(y)
\end{verbatim}
ry(2:4:2)=aimag(y)
call odeint(ry,0.0_sp,1.0_sp,EPS,0.1_sp,0.0001_sp,hypdrv,bstep)
    The arguments to odeint are the vector of independent variables, the starting and ending 
    values of the dependent variable, the accuracy parameter, an initial guess for steps size, a 
    minimum stepsize, and the names of the derivative routine and the (here Bulirsch-Stoer) 
    stepping routine.
y=cmplx(ry(1:4:2),ry(2:4:2),kind=spc)
hypgeo=y(1)
END FUNCTION hypgeo

SUBROUTINE hypser(a,b,c,z,series,deriv)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
COMPLEX(SPC), INTENT(IN) :: a,b,c,z
COMPLEX(SPC), INTENT(OUT) :: series,deriv
    Returns the hypergeometric series \( \sum_{n=1}^{\infty} \frac{(a)_n(b)_n}{(c)_n} \) and its derivative, iterating to machine accuracy.
    For \( c \geq 1 \), convergence is quite rapid.
INTEGER(I4B) :: n
INTEGER(I4B), PARAMETER :: MAXIT=1000
COMPLEX(SPC) :: aa,bb,cc,fac,temp
deriv=cmplx(0.0_sp,0.0_sp,kind=spc)
fac=cmplx(1.0_sp,0.0_sp,kind=spc)
temp=fac
aa=a
bb=b
cc=c
do n=1,MAXIT
    fac=((aa*bb)/cc)*fac
    deriv=deriv+fac
    series=temp+fac
    if (series == temp) RETURN
    temp=series
    aa=aa+1.0
    bb=bb+1.0
    cc=cc+1.0
end do
    call nrerror('hypser: convergence failure')
END SUBROUTINE hypser

SUBROUTINE hypdrv(s,ry,rdys)
USE nrtype
USE hypgeo_info
IMPLICIT NONE
REAL(SP), Intent(IN) :: s
REAL(SP), DIMENSION(:,), Intent(IN) :: ry
REAL(SP), Dimension(2), Intent(OUT) :: rdyds
    Derivative subroutine for the hypergeometric equation; see text equation (5.14.4).
COMPLEX(SPC), Dimension(2) :: y,rdys
COMPLEX(SPC) :: z
y=cmplx(ry(1:4:2),ry(2:4:2),kind=spc)
z=hypgeo_z0+s*hypgeo_dz
dyds(1)=y(2)*hypgeo_dz
dyds(2)=((hypgeo_aa*hypgeo_bb)*y(1)-(hypgeo_cc-&
    ((hypgeo_aa+hypgeo_bb)+1.0_sp)*z)*y(2))*hypgeo_dz/(z*(1.0_sp-z))
rdys(1:4:2)=real(dyds)
rdys(2:4:2)=aimag(dyds)
END SUBROUTINE hypdrv
Notice that the real array (of length 4) $r_y$ is immediately mapped into a complex array of length 2, and that the process is reversed at the end of the routine with $r_d y_d s$. In Fortran 77 no such mapping is necessary: the calling program sends real arguments, and the Fortran 77 hypdrv simply interprets what is sent as complex. Fortran 90’s stronger typing does not encourage (and, practically, does not allow) this convenience; but it is a small price to pay for the vastly increased error-checking capabilities of a strongly typed language.
Chapter B7. Random Numbers

One might think that good random number generators, including those in Volume 1, should last forever. The world of computing changes very rapidly, however:

- When Volume 1 was published, it was unusual, except on the fastest supercomputers, to “exhaust” a 32-bit random number generator, that is, to call for all $2^{32}$ sequential random values in its periodic sequence. Now, this is feasible, and not uncommon, on fast desktop workstations. A useful generator today must have a minimum of 64 bits of state space, and generally somewhat more.
- Before Fortran 90, the Fortran language had no standardized calling sequence for random numbers. Now, although there is still no standard algorithm defined by the language (rightly, we think), there is at least a standard calling sequence, exemplified in the intrinsics `random_number` and `random_seed`.
- The rise of parallel computing places new algorithmic demands on random generators. The classic algorithms, which compute each random value from the previous one, evidently need generalization to a parallel environment.
- New algorithms and techniques have been discovered, in some cases significantly faster than their predecessors.

These are the reasons that we have decided to implement, in Fortran 90, different uniform random number generators from those in Volume 1’s Fortran 77 implementations. We hasten to add that there is nothing wrong with any of the generators in Volume 1. That volume’s `ran0` and `ran1` routines are, to our knowledge, completely adequate as 32-bit generators; `ran2` has a 64-bit state space, and our previous offer of $1000 for any demonstrated failure in the algorithm has never yet been claimed (see [1]).

Before we launch into the discussion of parallelizable generators with Fortran 90 calling conventions, we want to attend to the continuing needs of longtime “x=ran(idum)” users with purely serial machines. If you are a satisfied user of Volume 1’s `ran0`, `ran1`, or `ran2` Fortran 77 versions, you are in this group. The following routine, `ran`, preserves those routines’ calling conventions, is considerably faster than `ran2`, and does not suffer from the old `ran0` or `ran1`’s 32-bit period exhaustion limitation. It is completely portable to all Fortran 90 environments. We recommend `ran` as the plug-compatible replacement for the old `ran0`, `ran1`, and `ran2`, and we happily offer exactly the same $1000 reward terms as were (and are still) offered on the old `ran2`. 

1141
FUNCTION ran(idum)
IMPLICIT NONE
INTEGER, PARAMETER :: K4B=selected_int_kind(9)
INTEGER(K4B), INTENT(INOUT) :: idum
REAL :: ran

"Minimal" random number generator of Park and Miller combined with a Marsaglia shift sequence. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This fully portable, scalar generator has the "traditional" (not Fortran 90) calling sequence with a random deviate as the returned function value: call with idum a negative integer to initialize; thereafter, do not alter idum except to reinitialize. The period of this generator is about $3.1 \times 10^{18}$.

INTEGER(K4B), PARAMETER :: IA=16807, IM=2147483647, IQ=127773, IR=2836
REAL, SAVE :: am
INTEGER(K4B), SAVE :: ix=-1, iy=-1, k

if (idum <= 0 .or. iy < 0) then
  Initialize.
  am=nearest(1.0,-1.0)/IM
  iy=ior(ieor(888889999, abs(idum)),1)
  ix=ieor(777755555, abs(idum))
  idum=abs(idum)+1
end if

ix=ieor(ix,ishft(ix,13))
ix=ieor(ix,ishft(ix,-17))
ix=ieor(ix,ishft(ix,5))
k=iy/IQ
iy=IA*(iy-k*IQ)-IR*k
if (iy < 0) iy=iy+IM
ran=am*ior(iand(IM,ieor(ix,iy)),1)

END FUNCTION ran

This is a good place to discuss a new bit of algorithmics that has crept into ran, above, and even more strongly affects all of our new random number generators, below. Consider:

ix=ieor(ix,ishft(ix,13))
ix=ieor(ix,ishft(ix,-17))
ix=ieor(ix,ishft(ix,5))

These lines update a 32-bit integer ix, which cycles pseudo-randomly through a full period of $2^{32} - 1$ values (excluding zero) before repeating. Generators of this type have been extensively explored by Marsaglia (see [2]), who has kindly communicated some additional results to us in advance of publication. For convenience, we will refer to generators of this sort as “Marsaglia shift registers.”

Useful properties of Marsaglia shift registers are (i) they are very fast on most machines, since they use only fast logical operations, and (ii) the bit-mixing that they induce is quite different in character from that induced by arithmetic operations such as are used in linear congruential generators (see Volume 1) or lagged Fibonacci generators (see below). Thus, the combination of a Marsaglia shift register with another, algorithmically quite different generator is a powerful way to suppress any residual correlations or other weaknesses in the other generator. Indeed, Marsaglia finds (and we concur) that the above generator (with constants 13, -17, 5, as shown) is by itself about as good as any 32-bit random generator.

Here is a very brief outline of the theory behind these generators: Consider the 32 bits of the integer as components in a vector of length 32, in a linear space where addition and multiplication are done modulo 2. Noting that exclusive-or (ieor) is the same as addition, each of the three lines in the updating can be written as the action of a $32 \times 32$ matrix on a vector, where the matrix is all zeros except for ones on
the diagonal, and on exactly one super- or subdiagonal (corresponding to positive or negative second arguments in `ishft`). Denote this matrix as $S_k$, where $k$ is the shift argument. Then, one full step of updating (three lines of code, above) corresponds to multiplication by the matrix $T = S_{k_3}S_{k_2}S_{k_1}$.

One next needs to find triples of integers $(k_1, k_2, k_3)$, for example $(13, -17, 5)$, that give the full $M = 2^{32} - 1$ period. Necessary and sufficient conditions are that $T^M = I$ (the identity matrix), and that $T^N \neq I$ for these five values of $N$: $N = 3 \times 5 \times 17 \times 257$, $N = 3 \times 5 \times 17 \times 65537$, $N = 3 \times 5 \times 257 \times 65537$, $N = 3 \times 17 \times 257 \times 65537$, $N = 5 \times 17 \times 257 \times 65537$. (Note that each of the five prime factors of $M$ is omitted one at a time to get the five values of $N$.) The required large powers of $T$ are readily computed by successive squarings, requiring only on the order of $32^3 \log M$ operations. With this machinery, one can find full-period triples $(k_1, k_2, k_3)$ by exhaustive search, at reasonable cost.

Not all such triples are equally good as generators of random integers, however. Marsaglia subjects candidate values to a battery of tests for randomness, and we have ourselves applied various tests. This stage of winnowing is as much art as science, because all 32-bit generators can be made to exhibit signs of failure due to period exhaustion (if for no other reason). “Good” triples, in order of our preference, are $(13, -17, 5)$, $(5, -13, 6)$, $(5, -9, 7)$, $(13, -17, 15)$, $(16, -7, 11)$. When a full-period triple is good, its reverse is also full-period, and also generally good. A good quadruple due to Marsaglia (generalizing the above in the obvious way) is $(-4, 8, -1, 5)$. We would not recommend relying on any single Marsaglia shift generator (nor on any other simple generator) by itself. Two or more generators, of quite different types, should be combined [1].

Let us now discuss explicitly the needs of parallel random number generators. The general scheme, from the user’s perspective, is that of Fortran 90’s intrinsic `random_number`: A statement like `call ran1(harvest)` (where `ran1` will be one of our portable replacements for the compiler-dependent `random_number`) should fill the real array `harvest` with pseudo-random real values in the range $(0, 1)$. Of course, we want the underlying machinery to be completely parallel, that is, no do-loops of order $N = \text{size}(\text{harvest})$.

A first design decision is whether to replicate the state-space across the parallel dimension $N$, i.e., whether to reserve storage for essentially $N$ scalar generators. Although there are various schemes that avoid doing this (e.g., mapping a single, smaller, state space into $N$ different output values on each call), we think that it is a memory cost well worth paying in return for achieving a less exotic (and thus better tested) algorithm. However, this choice dictates that we must keep the state space per component quite small. We have settled on five or fewer 32-bit words of state space per component as a reasonable limit. Some otherwise interesting and well tested methods (such as Knuth’s subtractive generator, implemented in Volume 1 as `ran3`) are ruled out by this constraint.

A second design decision is how to initialize the parallel state space, so that different parallel components produce different sequences, and so that there is an acceptable degree of randomness across the parallel dimension, as well as between successive calls of the generator. Each component starts its life with one and only one unique identifier, its component index $n$ in the range $1 \ldots N$. One is
tempted simply to hash the values \( n \) into the corresponding components of initial state space. “Random” hashing is a bad idea, however, because different \( n \)'s will produce identical 32-bit hash results by chance when \( N \) is no larger than \( \sim 2^{16} \). We therefore prefer to use a kind of reversible pseudo-encryption (similar to the routine psdes in Volume 1 and below) which guarantees causally that different \( n \)'s produce different state space initializations.

The machinery for allocating, deallocating, and initializing the state space, including provision of a user interface for getting or putting the contents of the state space (as in the intrinsic \texttt{random seed}) is fairly complicated. Rather than duplicate it in each different random generator that we provide, we have consolidated it in a single module, \texttt{ran state}, whose contents we will now discuss. Such a discussion is necessarily technical, if not arcane; on first reading, you may wish to skip ahead to the actual new routines \texttt{ran0}, \texttt{ran1}, and \texttt{ran2}. If you do so, you will need to know only that \texttt{ran state} provides each vector random routine with five 32-bit vectors of state information, denoted \texttt{iran}, \texttt{jran}, \texttt{kran}, \texttt{mran}, and \texttt{nran}. (The overloaded scalar generators have five corresponding 32-bit scalars, denoted \texttt{iran0}, etc.)

\begin{verbatim}
MODULE ran_state
This module supports the random number routines \texttt{ran0}, \texttt{ran1}, \texttt{ran2}, and \texttt{ran3}. It provides each generator with five integers (for vector versions, five vectors of integers), for use as internal state space. The first three integers (\texttt{iran}, \texttt{jran}, \texttt{kran}) are maintained as nonnegative values, while the last two (\texttt{mran}, \texttt{nran}) have 32-bit nonzero values. Also provided by this module is support for initializing or reinitializing the state space to a desired standard sequence number, hashing the initial values to random values, and allocating and deallocating the internal workspace.

USE nrtype
IMPLICIT NONE
INTEGER, PARAMETER :: K4B=selected_int_kind(9)  
Independent of the usual integer kind \texttt{I4B}, we need a kind value for (ideally) 32-bit integers.
INTEGER(K4B), PARAMETER :: hg=huge(1_K4B), hgm=-hg, hgng=hgm-1
INTEGER(K4B), SAVE :: lenran=0, seq=0
INTEGER(K4B), SAVE :: iran0,jran0,kran0,nran0,mran0,rans
INTEGER(K4B), DIMENSION(:,:), POINTER, SAVE :: ranseeds
INTEGER(K4B), DIMENSION(:,), POINTER, SAVE :: iran,jran,kran, &
mran,nran,ranv
REAL(SP), SAVE :: amm
INTERFACE ran_hash  
Scalar and vector versions of the hashing procedure.
MODULE PROCEDURE ran_hash_s, ran_hash_v
END INTERFACE
CONTAINS

(We here intersperse discussion with the listing of the module.) The module defines \texttt{K4B} as an integer \texttt{KIND} that is intended to be 32 bits. If your machine doesn’t have 32-bit integers (hard to believe!) this will be caught later, and an error message generated. The definition of the parameters \texttt{hg}, \texttt{hgm}, and \texttt{hgng} makes an assumption about 32-bit integers that goes beyond the strict Fortran 90 integer model, that the magnitude of the most negative representable integer is greater by one than that of the most positive representable integer. This is a property of the two’s complement arithmetic that is used on virtually all modern machines (see, e.g., [3]).

The global variables \texttt{rans} (for scalar) and \texttt{ranv} (for vector) are used by all of our routines to store the \texttt{integer} value associated with the most recently returned call. You can access these (with a “\texttt{USE ran_state}” statement) if you want integer, rather than real, random deviates.
\end{verbatim}
The first routine, `ran_init`, is called by routines later in the chapter to initialize their state space. It is not intended to be called from a user's program.

```fortran
SUBROUTINE ran_init(length)
  USE nrtype; USE nrutil, ONLY : arth, nrerror, reallocate
  IMPLICIT NONE
  INTEGER(K4B), INTENT(IN) :: length
  Initialize or reinitialize the random generator state space to vectors of size `length`. The saved variable `seq` is hashed (via calls to the module routine `ran_hash`) to create unique starting seeds, different for each vector component.

  INTEGER(K4B) :: new, j, hgt
  if (length < lenran) RETURN
  Simply return if enough space is already allocated.

  The following lines check that kind value K4B is in fact a 32-bit integer with the usual properties that we expect it to have (under negation and wrap-around addition). If all of these tests are satisfied, then the routines that use this module are portable, even though they go beyond Fortran 90's integer model.

  if (hg /= 2147483647) call nrerror('ran_init: arith assump 1 fails')
  if (hgt+1 /= hgng) call nrerror('ran_init: arith assump 2 fails')
  if (not(hgng) >= 0) call nrerror('ran_init: arith assump 3 fails')
  if (not(hgng) < 0) call nrerror('ran_init: arith assump 4 fails')
  if (not(hgng) >= 0) call nrerror('ran_init: arith assump 5 fails')
  if (not(-1_k4b) < 0) call nrerror('ran_init: arith assump 6 fails')
  if (not(0_k4b) >= 0) call nrerror('ran_init: arith assump 7 fails')
  if (not(1_k4b) >= 0) call nrerror('ran_init: arith assump 8 fails')
  if (lenran > 0) then
    Reallocate space, or...
    ranseeds => reallocate(ranseeds, length, 5)
    ranv => reallocate(ranv, length-1)
    new = lenran+1
    allocate space.
    allocate(ranseeds(length,5))
    allocate(ranv(length-1))
    new = 1
    Index of first location not yet initialized.
    amm = nearest(1.0_sp, -1.0_sp)/hgng
    Use of nearest is to ensure that returned random deviates are strictly less than 1.0.
    if (amm*hgng >= 1.0 .or. amm*hgng <= 0.0) &
      call nrerror('ran_init: arith assump 10 fails')
  end if

  Set starting values, unique by seq and vector component.
  ranseeds(new;1)=seq
  ranseeds(new;2:5)=spread(arth(new,1,size(ranseeds(new;1))),2,4)
  do j=1,4
    Hash them.
    call ran_hash(ranseeds(new;j), ranseeds(new;j+1))
  end do
  where (ranseeds(new;1:3) < 0) &
    Enforce nonnegativity.
  ranseeds(new;1:3) = not(ranseeds(new;1:3))
  where (ranseeds(new;4:5) == 0) ranseeds(new;4:5)=1
  Enforce nonzero.

  if (new == 1) then
    Set scalar seeds.
    i ran0 = ranseeds(1,1)
    j ran0 = ranseeds(1,2)
    k ran0 = ranseeds(1,3)
    m ran0 = ranseeds(1,4)
    n ran0 = ranseeds(1,5)
    ranv = n ran0
  end if

  if (length > 1) then
    Point to vector seeds.
    i ran = ranseeds(2:1)
    j ran = ranseeds(2:2)
    k ran = ranseeds(2:3)
    m ran = ranseeds(2:4)
    n ran = ranseeds(2:5)
    ranv = n ran
  end if
```

Bit of dirty laundry here! We are testing whether the most positive integer \( h_g \) wraps around to the most negative integer \( h_gng \) when 1 is added to it. We can’t just write \( h_g+1 \), since some compilers will evaluate this at compile time and return an overflow error message. If your compiler sees through the charade of the temporary variable \( h_gt \), you’ll have to find another way to trick it.

Logically, \( a m_m \) should be a parameter; but the nearest intrinsic is trouble-prone in the initialization expression for a parameter (named constant), so we compute this at run time. We then check that \( a m_m \), when multiplied by the largest possible negative integer, does not equal or exceed unity. (Our random deviates are guaranteed never to equal zero or unity exactly.)

You might wonder why \( a m_m \) is negative, and why we multiply it by negative integers to get positive random deviates. The answer, which will become manifest in the random generators given below, is that we want to use the fast \( \text{not} \) operation on integers to convert them to nonzero values of all one sign. This is possible if the conversion is to negative values, since \( \text{not}(i) \) is negative for all nonnegative \( i \). If the conversion were to positive values, we would have problems both with zero (its sign bit is already positive) and \( h_gng \) (since \( \text{not}(h_gng) \) is generally zero).

The initial state information is stored in \( r a n_s e e d s \), a two-dimensional array whose column (second) index ranges from 1 to 5 over the state variables. \( r a n_s e e d s(1,:) \) is reserved for scalar random generators, while \( r a n_s e e d s(2,:,:) \) is for vector-parallel generators. The \( r a n_s e e d s \) array is made available to vector generators through the pointers \( i_r a n, j_r a n, k_r a n, m_r a n, n_r a n \). The corresponding scalar values, \( i_r a n_0, ..., n_r a n_0 \) are simply global variables, not pointers, because the overhead of addressing a scalar through a pointer is often too great. (We will have to copy these scalar values back into \( r a n_s e e d s \) when it, rarely, needs to be addressed as an array.)

The above routine is supplied as a user interface for deallocating all the state space storage.
SUBROUTINE ran_seed(sequence, size, put, get)
IMPLICIT NONE
INTEGER, OPTIONAL, INTENT(IN) :: sequence
INTEGER, OPTIONAL, INTENT(OUT) :: size
INTEGER, DIMENSION(:), OPTIONAL, INTENT(IN) :: put
INTEGER, DIMENSION(:), OPTIONAL, INTENT(OUT) :: get
User interface for seeding the random number routines. Syntax is exactly like Fortran 90's random_seed routine, with one additional argument keyword: sequence, set to any integer value, causes an immediate new initialization, seeded by that integer.
if (present(size)) then
  size=5*lenran
else if (present(put)) then
  if (lenran == 0) RETURN
  ranseeds=reshape(put,shape(ranseeds))
  where (ranseeds(:,1:3) < 0) ranseeds(:,1:3)=not(ranseeds(:,1:3))
  where (ranseeds(:,4:5) == 0) ranseeds(:,4:5)=1
  iran0=ranseeds(1,1)
  jran0=ranseeds(1,2)
  kran0=ranseeds(1,3)
  mran0=ranseeds(1,4)
  nran0=ranseeds(1,5)
else if (present(get)) then
  if (lenran == 0) RETURN
  ranseeds(1,1:5)=(/ iran0,jran0,kran0,mran0,nran0 /)
  get=reshape(ranseeds,shape(get))
else if (present(sequence)) then
  call ran_deallocate
  seq=sequence
end if
END SUBROUTINE ran_seed

Fortran 90’s convention is that random state space is a one-dimensional array, so we map to this on both the get and put keywords.

iran0=...jran0=...kran0=...
ranseeds(1,1:5)=(/ iran0,jran0,kran0,mran0,nran0 /)

It’s much more convenient to set a vector from a bunch of scalars then the other way around.

SUBROUTINE ran_hash_s(il,ir)
IMPLICIT NONE
INTEGER(K4B), INTENT(INOUT) :: il,ir
DES-like hashing of two 32-bit integers, using shifts, xor’s, and adds to make the internal nonlinear function.
INTEGER(K4B) :: is,j
do j=1,4
  is=ir
  ir=ieor(is,ishft(is,5))#1422217823
  ir=ieor(is,ishft(is,-16))#1842055030
  is=ieor(is,ishft(is,9))#80567781
  il=is
end do
END SUBROUTINE ran_hash_s
SUBROUTINE ran_hash_v(il,ir)
IMPLICIT NONE
INTEGER(K4B), DIMENSION(:), INTENT(INOUT) :: il,ir
    Vector version of ran_hash_s.
INTEGER(K4B), DIMENSION(size(il)) :: is
INTEGER(K4B) :: j
do j=1,4
    is=ir
    ir=ieor(ir,ishft(ir,5))+1422217823
    ir=ieor(ir,ishft(ir,-16))+1842055030
    ir=ieor(ir,ishft(ir,9))+80567781
    ir=ieor(il,ir)
    il=is
end do
END SUBROUTINE ran_hash_v
END MODULE ran_state

The lines

\[
\begin{align*}
    ir &= \text{ieor}(ir, \text{ishft}(ir,5)) + 1422217823 \\
    ir &= \text{ieor}(ir, \text{ishft}(ir,-16)) + 1842055030 \\
    ir &= \text{ieor}(ir, \text{ishft}(ir,9)) + 80567781
\end{align*}
\]

are not a Marsaglia shift sequence, though they resemble one. Instead, they implement a fast, nonlinear function on \( ir \) that we use as the “S-box” in a DES-like hashing algorithm. (See Volume 1, §7.5.) The triplet \((5, -16, 9)\) is not chosen to give a full period Marsaglia sequence — it doesn’t. Instead it is chosen as being particularly good at separating in Hamming distance (i.e., number of nonidentical bits) two initially close values of \( ir \) (e.g., differing by only one bit). The large integer constants are chosen by a similar criterion. Note that the wrap-around of addition without generating an overflow error condition, which was tested in \texttt{ran_init}, is relied upon here.

* * *

SUBROUTINE ran0_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init,iran0,jran0,kran0,nran0,rans
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
Lagged Fibonacci generator combined with a Marsaglia shift sequence. Returns as \texttt{harvest} a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This generator has the same calling and initialization conventions as Fortran 90’s \texttt{random} routine. Use \texttt{ran_seed} to initialize or reinitialize to a particular sequence. The period of this generator is about \( 2^{31} \times 10^{28} \), and it fully vectorizes. Validity of the integer model assumed by this generator is tested at initialization.

if (lenran < 1) call ran_init(1)
    if (rans < 0) rans=rans+2147483579_k4b
    iran0=jran0
    jran0=kran0
    kran0=rans
    nran0=ieor(nran0,ishft(nran0,13))
    nran0=ieor(nran0,ishft(nran0,-17))
    nran0=ieor(nran0,ishft(nran0,5))
    rans=ieor(rans,rans)
    harvest=amm*merge(rans,not(rans), rans<0 )
END SUBROUTINE ran0_s

Initialization routine in \texttt{ran_state}. Update Fibonacci generator, which has period \( p^2 + p + 1, p = 2^{31} - 69 \).

Update Marsaglia shift sequence with period \( 2^{32} - 1 \).

Combine the generators.

Make the result positive definite (note that \( \text{amm} \) is negative).
This is the simplest, and fastest, of the generators provided. It combines a subtractive Fibonacci generator (Number 6 in ref. [1], and one of the generators in Marsaglia and Zaman’s mzran) with a Marsaglia shift sequence. On typical machines it is only 20% or so faster than ran1, however; so we recommend the latter preferentially. While we know of no weakness in ran0, we are not offering a prize for finding a weakness. ran0 does have the feature, useful if you have a machine with nonstandard arithmetic, that it does not go beyond Fortran 90’s assumed integer model.

Note that ran0_s and ran0_v are overloaded by the module nr onto the single name ran0 (and similarly for the routines below).

SUBROUTINE ran1_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init,iran0,jran0,kran0,nran0,mran0,rans
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest

Lagged Fibonacci generator combined with two Marsaglia shift sequences. On output, returns as harvest a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This generator has the same calling and initialization conventions as Fortran 90’s random_number routine. Use ran_seed to initialize or reinitialize to a particular sequence.

The period of this generator is about \( 8.5 \times 10^{37} \), and it fully vectorizes. Validity of the integer model assumed by this generator is tested at initialization.

Initialization routine in ran_state.
Update Fibonacci generator, which has period \( p^2 + p + 1, p = 2^{31} - 69 \).
Update Marsaglia shift sequence.
Once only per cycle, advance sequence by 1, shortening its period to \( 2^{32} - 2 \).
Update Marsaglia shift sequence with period \( 2^{32} - 1 \).
rans=ieor(nran0,rans)+mran0

Combine the generators. The above statement has wrap-around addition.

harvest=amm*merge(rans,not(rans),rans<0 )

Make the result positive definite (note that amm is negative).

SUBROUTINE ran1_v(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init, &
  iran,jran,kran,nran,mran,ranv
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(OUT) :: harvest
INTEGER(K4B) :: n
n=size(harvest)
if (lenran < n+1) call ran_init(n+1)
ranv(1:n)=iran(1:n)-kran(1:n)
where (ranv(1:n) < 0) ranv(1:n)=ranv(1:n)+2147483579_k4b
iran(1:n)=jran(1:n)
jran(1:n)=kran(1:n)
kran(1:n)=ranv(1:n)
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),13))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),-17))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),5))
where (nran(1:n) == 1) nran(1:n)=270369_k4b
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),5))
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),-13))
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),6))
ranv(1:n)=ieor(nran(1:n),ranv(1:n))+mran(1:n)
harvest=amm*merge(ranv(1:n),not(ranv(1:n)), ranv(1:n)<0 )
END SUBROUTINE ran1_v

The routine ran1 combines three fast generators: the two used in ran0, plus an additional (different) Marsaglia shift sequence. The last generator is combined via an addition that can wrap-around.

We think that, within the limits of its floating-point precision, ran1 provides perfect random numbers. We will pay $1000 to the first reader who convinces us otherwise (by exhibiting a statistical test that ran1 fails in a nontrivial way, excluding the ordinary limitations of a floating-point representation).

* * *

SUBROUTINE ran2_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init, &
  iran0,jran0,kran0,nran0,mran0,ran0
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest

Lagged Fibonacci generator combined with a Marsaglia shift sequence and a linear congruential generator. Returns as harvest a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This generator has the same calling and initialization conventions as Fortran 90's random number routine. Use ran_seed to initialize or reinitialize to a particular sequence. The period of this generator is about $8.5 \times 10^{37}$, and it fully vectorizes. Validity of the integer model assumed by this generator is tested at initialization.

if (lenran < 1) call ran_init(1)
ran0=iran0-kran0
if (rans < 0) rans=rans+2147483579_k4b
iran0=jran0
jran0=kran0
kran0=rans

Initialization routine in ran_state. Update Fibonacci generator, which has period $p^2 + p + 1$, $p = 2^{31} - 69$. 

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. Further reproduction of any part or the whole without permission from the publishers is prohibited.
ran2, for use by readers whose caution is extreme, also combines three
generators. The difference from ran1 is that each generator is based on a completely
different method from the other two. The third generator, in this case, is a linear
congruential generator, modulo $2^{32}$. This generator relies extensively on wrap-
around addition (which is automatically tested at initialization). On machines with
fast arithmetic, ran2 is on the order of only 20% slower than ran1. We offer a
$1000 bounty on ran2, with the same terms as for ran1, above.
SUBROUTINE expdev_v(harvest)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
REAL(SP), DIMENSION(size(harvest)) :: dum
call ran1(dum)
harvest=-log(dum)
END SUBROUTINE expdev_v

The only noteworthy thing about this line is its simplicity: Once all the machinery is in place, the random number generators are self-initializing (to the sequence defined by \( seq = 0 \)), and (via overloading) usable with both scalar and vector arguments.

SUBROUTINE gasdev_s(harvest)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
Returns in harvest a normally distributed deviate with zero mean and unit variance, using ran1 as the source of uniform deviates.
REAL(SP) :: rsq,v1,v2
REAL(SP), SAVE :: g
LOGICAL, SAVE :: gaus_stored=.false.
if (gaus_stored) then
    harvest=g
    gaus_stored=.false.
else
    call ran1(v1)
call ran1(v2)
v1=2.0_sp*v1-1.0_sp
v2=2.0_sp*v2-1.0_sp
rsq=v1**2+v2**2
if (rsq > 0.0 .and. rsq < 1.0) exit
do
    rsq=sqrt(-2.0_sp*log(rsq)/rsq)
end do
harvest=v1*rsq
g=v2*rsq
gaus_stored=.true.
end if
END SUBROUTINE gasdev_s

SUBROUTINE gasdev_v(harvest)
USE nrtype; USE nrutil, ONLY : array_copy
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
REAL(SP), DIMENSION(size(harvest)) :: rsq,v1,v2
REAL(SP), ALLOCATABLE, DIMENSION(:), SAVE :: g
INTEGER(I4B) :: n,ng,nn,m
INTEGER(I4B), SAVE :: last_allocated=0
LOGICAL, SAVE :: gaus_stored=.false.
LOGICAL, DIMENSION(size(harvest)) :: mask
n=size(harvest)
if (n /= last_allocated) then
if (last_allocated /= 0) deallocate(g)
allocate(g(n))
last_allocated=n
gaus_stored=.false.
end if
if (gaus_stored) then
harvest=g
gaus_stored=.false.
else
ng=1
do
if (ng > n) exit
call ran1(v1(ng:n))
call ran1(v2(ng:n))
v1(ng:n)=2.0_sp*v1(ng:n)-1.0_sp
v2(ng:n)=2.0_sp*v2(ng:n)-1.0_sp
rsq(ng:n)=v1(ng:n)**2+v2(ng:n)**2
mask(ng:n)=(rsq(ng:n)>0.0 .and. rsq(ng:n)<1.0)
call array_copy(pack(v1(ng:n),mask(ng:n)),v1(ng:),nn,m)
v2(ng:ng+nn-1)=pack(v2(ng:n),mask(ng:n))
rsq(ng:ng+nn-1)=pack(rsq(ng:n),mask(ng:n))
ng=ng+nn
end do
rsq=sqrt(-2.0_sp*log(rsq)/rsq)
harvest=v1*rsq
g=v2*rsq
gaus_stored=.true.
end if
END SUBROUTINE gasdev_v

if (n /= last_allocated) ... We make the assumption that, in most cases, the size of harvest will not change between successive calls. Therefore, if it does change, we don’t try to save the previously generated deviates that, half the time, will be around. If your use has rapidly varying sizes (or, even worse, calls alternating between two different sizes), you should remedy this inefficiency in the obvious way.

call array_copy(pack(v1(ng:n),mask(ng:n)),v1(ng:),nn,m) This is a variant of the pack-unpack method (see note to factrl, p. 1087). Different here is that we don’t care which random deviates end up in which component. Thus, we can simply keep packing successful returns into v1 and v2 until they are full.

Note also the use of array_copy, since we don’t know in advance the length of the array returned by pack.

FUNCTION gamdev(ia)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : ran1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ia
REAL(SP) :: gamdev
Returns a deviate distributed as a gamma distribution of integer order ia, i.e., a waiting time to the ia-th event in a Poisson process of unit mean, using ran1 as the source of uniform deviates.
REAL(SP) :: a,a,e,h,a,x,y,v(2),arr(5)
call assert(ia > 0, ‘gamdev arg’)
if (ia < 6) then
  Use direct method, adding waiting times.
call ran1(arr(1:ia))
x=-log(product(arr(1:ia)))
else
do
   call ran1(v)
   v(2)=2.0_sp*v(2)-1.0_sp
   These three lines generate the tangent of a
   random angle, i.e., are equivalent to
   y=v(2)/v(1)
   am=ia-1
   s=sqrt(2.0_sp*am+1.0_sp)
   x=y*s
We decide whether to reject x:
   if (x <= 0.0) cycle
   Reject in region of zero probability.
   e=(1.0_sp+y**2)*exp(am*log(x/am)-s*y)
   Ratio of probability function to
   call ran1(h)
   comparison function.
   if (h <= e) exit
   Reject on basis of a second uniform deviate.
end do
end if
gamdev=x
END FUNCTION gamdev

Why take the log of the product instead of
the sum of the logs? Because log is assumed to be slower than multiply.

We don’t have vector versions of the less commonly used deviate generators, gamdev, poidev, and bnldev.

FUNCTION poidev(xm)
USE nrtype
USE nr, ONLY : gammln, ran1
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xm
REAL(SP) :: poidev

Returns as a floating-point number an integer value that is a random deviate drawn from a
Poisson distribution of mean \( \mu \), using \texttt{ran1} as a source of uniform random deviates.

REAL(SP) :: em, harvest, t, y
REAL(SP), SAVE :: alxm, g, oldm=-1.0_sp, sq
oldm is a flag for whether \( \mu \) has changed since last call.
if (xm < 12.0) then
   Use direct method.
   if (xm /= oldm) then
      oldm=xm
      g=exp(-xm)
      end if
      em=-1
      t=1.0
      do
         em=em+1.0_sp
         call ran1(harvest)
         t=t*harvest
         if (t <= g) exit
         end do
      else
         if (xm /= oldm) then
            oldm=xm
            sq=sqrt(2.0_sp*xm)
            alxm=log(xm)
            g=xm*alxm-gammln(xm+1.0_sp)
         end if
doi
end if

do
    call ran1(harvest)
y(tan(PI*harvest)) = is a deviate from a Lorentzian comparison function.
    em=sq*y+xm
    if (em >= 0.0) exit
end do
em=int(em)

The trick for integer-valued distributions.
t=0.9_sp*(1.0_sp+y**2)*exp(em*alxm-gammln(em+1.0_sp)-g)
The ratio of the desired distribution to the comparison function; we accept or reject
by comparing it to another uniform deviate. The factor 0.9 is chosen so that t never
exceeds 1.
call ran1(harvest)
if (harvest <= t) exit
end do

end if

poidev=em
END FUNCTION poidev

FUNCTION bnldev(pp,n)
USE nrtype
USE nr, ONLY : gammln, ran1
IMPLICIT NONE
REAL(SP), INTENT(IN) :: pp
INTEGER(I4B), INTENT(IN) :: n
REAL(SP) :: bnldev

Returns as a floating-point number an integer value that is a random deviate drawn from a
binomial distribution of n trials each of probability pp, using ran1 as a source of uniform
random deviates.

INTEGER(I4B) :: j
INTEGER(I4B), SAVE :: nold=-1
REAL(SP) :: am,em,g,h,p,sq,t,y,arr(24)
REAL(SP), SAVE :: pc,plog,pclog,en,oldg,pold=-1.0

Arguments from previous calls.
p=merge(pp,1.0_sp-pp, pp <= 0.5_sp )
The binomial distribution is invariant under changing pp to 1.-pp, if we also change the
answer to n minus itself; we'll remember to do this below.
am=n*p

if (n < 25) then
    call ran1(arr(1:n))
    bnldev=count(arr(1:n)<p)
else if (am < 1.0) then
    g=exp(-am)
t=1.0
    do j=0,n
        call ran1(h)
t=t*h
        if (t < g) exit
    end do
    bnldev=merge(j,n, j <= n)
else
    if (n /= nold) then
        If fewer than one event is expected out of 25 or more trials, then the distribution is quite
        accurately Poisson. Use direct Poisson method.
        g=exp(-am)
        t=1.0
        do j=0,n
            call ran1(h)
t=t*h
            if (t < g) exit
        end do
        bnldev=merge(j,n, j <= n)
    end if
    else
        if (p /= pold) then
            Use the rejection method.
            pc=1.0_sp-p
            plog=log(pc)
pold=p
        end if
        end if

END FUNCTION bnldev
The following code should by now seem familiar: a rejection method with a Lorentzian comparison function.

\[
\begin{align*}
\text{sq} &= \sqrt{2.0 \cdot \text{am} \cdot \text{pc}} \\
\text{do} & \\
\text{call ran1(h)} \\
y &= \tan(\pi \cdot h) \\
\text{em} &= \text{sq} \cdot y + \text{am} \\
\text{if (em < 0.0 .or. em >= \text{en} + 1.0)} \text{ cycle Reject.} \\
\text{em} &= \text{int(em)} \\
t &= 1.2 \cdot \text{sq} \cdot (1.0 \cdot \text{y}^2) \cdot \exp(\text{oldg} - \text{gammln}(\text{em} + 1.0) - \text{gammln}((\text{en} - \text{em}) + 1.0) + \text{em} \cdot \text{plog} + (\text{en} - \text{em}) \cdot \text{pclog}) \\
\text{call ran1(h)} \\
\text{if (h <= t)} \text{ exit Reject. This happens about 1.5 times per deviation, on average.} \\
\text{end do} \\
\text{bnldev} &= \text{em} \\
\text{end if} \\
\text{if (p /= pp) bnldev} &= n - \text{bnldev} \\
\text{end if} \\
\text{END FUNCTION bnldev} \\
\end{align*}
\]

The routines \textit{psdes} and \textit{psdes\_safe} both perform exactly the same hashing as was done by the Fortran 77 routine \textit{psdes}. The difference is that \textit{psdes} makes assumptions about arithmetic that go beyond the strict Fortran 90 model, while \textit{psdes\_safe} makes no such assumptions. The disadvantage of \textit{psdes\_safe} is that it is significantly slower, performing most of its arithmetic in double-precision reals that are then converted to integers with Fortran 90's modulo intrinsic.

In fact the nonsafe version, \textit{psdes}, works fine on almost all machines and compilers that we have tried. There is a reason for this: Our assumed integer model is the same as the C language unsigned \texttt{int}, and virtually all modern computers and compilers have a lot of C hidden inside. If \textit{psdes} and \textit{psdes\_safe} produce identical output on your system for any hundred or so different input values, you can be quite confident about using the faster version exclusively.

At the other end of things, note that in the very unlikely case that your system fails on the \textit{ran\_hash} routine in the \textit{ran\_state} module (you will have learned this from error messages generated by \textit{ran\_init}), you can substitute \textit{psdes\_safe} for \textit{ran\_hash}: They are plug-compatible.

\begin{verbatim}
SUBROUTINE psdes_s(lword,rword)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4

"Pseudo-DES" hashing of the 64-bit word (lword,irword). Both 32-bit arguments are returned hashed on all bits. Note that this version of the routine assumes properties of integer arithmetic that go beyond the Fortran 90 model, though they are compatible with unsigned integers in C.

INTEGER(I4B), DIMENSION(4), SAVE :: C1,C2
DATA C1 /Z'BAA96887',Z'1E17D32C',Z'03BCDC3C',Z'0F33D1B2'/
DATA C2 /Z'4B0F3B58',Z'E874F0C3',Z'6955C5A6',Z'55A7CA46'/

INTEGER(I4B) :: i,ia,ib,iswap,itmph,itmpl

\textbf{do i=1,NITER}
Perform \texttt{niter} iterations of DES logic, using a simpler
\textbf{iswap=word} (noncryptographic) nonlinear function instead of DES's.
\textbf{ia=ieor(rword,C1(i))} The bit-rich constants C1 and (below) C2 guarantee lots
\textbf{itmpl=iand(ia,65535)} of nonlinear mixing.
\textbf{itmpb=iand(ishift(ia,-16),65535)}
\textbf{end do}
\end{verbatim}
SUBROUTINE psdes_s(lword,rword)
USE nrtype, USE nrutil, ONLY : assert_eq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(4), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
INTEGER(I4B), DIMENSION(4), SAVE :: C1,C2
DATA C1 /'BAA96887',/'1E17D32C',/'03BCDC3C',/'0F33D1B2'/
DATA C2 /'4B0F3B58',/'E874F0C3',/'6955C5A6',/'55A7CA46'/
INTEGER(I4B), DIMENSION(size(lword)) :: ia,ib,iswap,itmph,itmpl
INTEGER(I4B) :: i
i=assert_eq(size(lword),size(rword),'psdes_s')
do i=1,NITER
iswap=rword
ia=ieor(rword,C1(i))
itmpl=iand(ia,65535)
itmph=iand(ishft(ia,-16),65535)
ib=itmpl**2+not(itmph**2)
ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
rword=ieor(lword,ieor(C2(i),ia)+itmpl*itmph)
lword=iswap
end do
END SUBROUTINE psdes_s

SUBROUTINE psdes_v(lword,rword)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
INTEGER(I4B), DIMENSION(4), SAVE :: C1,C2
DATA C1 /'BAA96887',/'1E17D32C',/'03BCDC3C',/'0F33D1B2'/
DATA C2 /'4B0F3B58',/'E874F0C3',/'6955C5A6',/'55A7CA46'/
INTEGER(I4B), DIMENSION(size(lword)) :: ia,ib,iswap,itmph,itmpl
INTEGER(I4B) :: i
i=assert_eq(size(lword),size(rword),'psdes_v')
do i=1,NITER
iswap=rword
ia=ieor(rword,C1(i))
alo=real(iand(ia,65535),dp)
ahi=real(iand(ishft(ia,-16),65535),dp)
ib=modint(alo*alo+real(not(modint(ahi*ahi)),dp))
ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
rword=ieor(lword,modint(real(ieor(C2(i),ia),dp)+alo*ahi))
lword=iswap
end do
END SUBROUTINE psdes_v

SUBROUTINE psdes_safe_s(lword,rword)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
"Pseudo-DES" hashing of the 64-bit word (lword,irword). Both 32-bit arguments are
returned hashed on all bits. This is a slower version of the routine that makes no assumptions
outside of the Fortran 90 integer model
INTEGER(I4B), DIMENSION(4), SAVE :: C1,C2
DATA C1 /'BAA96887',/'1E17D32C',/'03BCDC3C',/'0F33D1B2'/
DATA C2 /'4B0F3B58',/'E874F0C3',/'6955C5A6',/'55A7CA46'/
REAL(DP) :: alo,ahi
do i=1,NITER
iswap=rword
ia=ieor(rword,C1(i))
alo=real(iand(ia,65535),dp)
ahi=real(iand(ishft(ia,-16),65535),dp)
ib=modint(alo*alo+real(not(modint(ahi*ahi)),dp))
ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
rword=ieor(lword,modint(real(ieor(C2(i),ia),dp)+alo*ahi))
lword=iswap
end do
CONTAINS
FUNCTION modint(x)
REAL(DP), INTENT(IN) :: x
INTEGER(I4B) :: modint
REAL(DP) :: a
REAL(DP), PARAMETER :: big=huge(modint), base=big+big+2.0_dp
a=modulo(x,base)
if (a > big) a = a - base
modint = int(a, kind=4b)
END FUNCTION modint
END SUBROUTINE psdes_safe_s

SUBROUTINE psdes_safe_v(lword, rword)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
INTEGER(4b), DIMENSION(:,), INTENT(INOUT) :: lword, rword
INTEGER(4b), PARAMETER :: NITER = 4
INTEGER(4b), SAVE :: C1(4), C2(4)
DATA C1 /'BA96887',/'E17D32C',/'03BDC3C',/'0F33D1B2'/
DATA C2 /'4B0F3B58',/'E874F0C3',/'6955C5A6',/'55A7CA46'/
INTEGER(4b), DIMENSION(size(lword)) :: ia, ib, iswap
REAL(dp), DIMENSION(size(lword)) :: alo, ahi
INTEGER(4b) :: i
i = assert_eq(size(lword), size(rword), 'psdes_safe_v')
do i = 1, NITER
  iswap = rword
  ia = ieor(rword, C1(i))
  alo = real(iand(ia, 65535), dp)
  ahi = real(iand(ishft(ia, -16), 65535), dp)
  ib = modint(alo * alo + real(not(modint(ahi * ahi)), dp))
  ia = ior(ishft(ib, 16), iand(ishft(ib, -16), 65535))
  rword = ieor(lword, modint(real(ieor(C2(i), ia), dp) + alo * ahi))
  lword = iswap
end do
CONTAINS
  FUNCTION modint(x)
  REAL(dp), DIMENSION(:,), INTENT(IN) :: x
  INTEGER(4b), DIMENSION(size(x)) :: modint
  REAL(dp), PARAMETER :: big = huge(modint), base = big + big + 2.0_dp
  a = modulo(x, base)
  where (a > big) a = a - base
  modint = int(a, kind=4b)
END FUNCTION modint
END SUBROUTINE psdes_safe_v

FUNCTION modint(x)
  This embedded routine takes a double-precision real argument, and returns it as an integer mod $2^{32}$ (correctly wrapping it to negative to take into account that Fortran 90 has no unsigned integers).

* * *

SUBROUTINE ran3_s(harvest)
USE nrtype
USE ran_state, ONLY: k4b, am, lenran, ran_init, ran_hash, mr0, nran0, rans
IMPLICIT NONE
REAL(dp), INTENT(OUT) :: harvest
Random number generation by DES-like hashing of two 32-bit words, using the algorithm ran_hash. Returns as harvest a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values).
INTEGER(k4b) :: temp
if (lenran < 1) call ran_init(1)
nran0 = ieor(mran0, ishf(mran0, 13))
nran0 = ieor(mran0, ishf(mran0, -17))
nran0 = ieor(mran0, ishf(mran0, 5))
if (nran0 == 1) nran0 = 270369_k4b
Initialize.
Two Marsaglia shift sequences are maintained as input to the hashing. The period of the combined generator is about $1.8 \times 10^{19}$. 
As given, ran3 uses the ran_hash function in the module ran_state as its DES surrogate. That function is sufficiently fast to make ran3 only about a factor of 2 slower than our baseline recommended generator ran1. The slower routine psdes and (even slower) psdes_safe are plug-compatible with ran_hash, and could be substituted for it in this routine.

    *    *    *

FUNCTION irbit1(iseed)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: iseed
Irbit1 returns as an integer a random bit, based on the 18 low-significance bits in iseed (which is modified for the next call).
if (btest(iseed,17) .neqv. btest(iseed,4) .neqv. btest(iseed,1) &
    .neqv. btest(iseed,0)) then
    iseed=ibset(ishft(iseed,1),0)          Leftshift the seed and put a 1 in its bit 1.
    irbit1=1
else
    iseed=ishft(iseed,1)                    But if the XOR calculation gave a 0,
    irbit1=0                                then put that in bit 1 instead.
end if
END FUNCTION irbit1
FUNCTION irbit2(iseed)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: iseed
INTEGER(I4B) :: irbit2
Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which is modified for the next call).
INTEGER(I4B), PARAMETER :: IB1=1,IB2=2,IB5=16,MASK=IB1+IB2+IB5
if (btest(iseed,17)) then
  Change all masked bits, shift, and put 1 into bit 1.
  iseed=ibset(ishft(ieor(iseed,MASK),1),0)
  irbit2=1
else
  Shift and put 0 into bit 1.
  iseed=ibclr(ishft(iseed,1),0)
  irbit2=0
end if
END FUNCTION irbit2

SUBROUTINE sobseq(x,init)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: x
INTEGER(I4B), OPTIONAL, INTENT(IN) :: init
INTEGER(I4B), PARAMETER :: MAXBIT=30,MAXDIM=6
When the optional integer init is present, internally initializes a set of MAXBIT direction numbers for each of MAXDIM different Sobol' sequences. Otherwise returns as the vector x of length N the next values from N of these sequences. (N must not be changed between initializations.)
REAL(SP), SAVE :: fac
INTEGER(I4B) :: i,im,ipp,j,k,l
INTEGER(I4B), DIMENSION(:,:), ALLOCATABLE:: iu
INTEGER(I4B), SAVE :: in
INTEGER(I4B), DIMENSION(MAXDIM), SAVE :: ip,ix,mdeg
INTEGER(I4B), DIMENSION(MAXDIM*MAXBIT), SAVE :: iv
DATA ip /0,1,1,2,1,4/, mdeg /1,2,3,3,4,4/, ix /6*0/
DATA iv /6*1,3,1,1,5,7,3,3,5,15,11,5,15,13,9,156*0/
if (present(init)) then
  Initialize, don't return a vector.
  ix=0
  in=0
  if (iv(1) /= 1) RETURN
  fac=1.0_sp/2.0_sp**MAXBIT
  allocate(iu(MAXDIM,MAXBIT))
  iv=reshape(iv,shape(iu))
  To allow both 1D and 2D addressing.
  do k=1,MAXDIM
    do j=1,mdeg(k)
      Stored values require only normalization.
      iu(k,j)=iu(k,j)*2**(MAXBIT-j)
    end do
    do j=mdeg(k)+1,MAXBIT
      Use the recurrence to get other values.
      ipp=ip(k)
      i=iu(k,j-mdeg(k))
      i=ieor(i,1/2**mdeg(k))
      do l=mdeg(k)-1,1,-1
        if (btest(ipp,0)) i=ieor(i,1,iu(k,j-l))
        ipp=ipp/2
      end do
      iu(k,j)=1
    end do
  end do
  iv=reshape(iv,shape(iv))
deallocate(iu)
else  
    Calculate the next vector in the sequence.
    im=in
    do j=1,MAXBIT
        if (.not. btest(im,0)) exit
        im=im/2
    end do
    if (j > MAXBIT) call nrerror('MAXBIT too small in sobseq')
    j=min(size(x),MAXDIM)
    ix(1:j)=ieor(ix(1:j),iv(1+im:j+im))
    XOR the appropriate direction number into each component of the vector and convert
    to a floating number.
    x(1:j)=ix(1:j)*fac
    im=in+1  
    Increment the counter.
END SUBROUTINE sobseq

if (present(init)) then ... allocate(iu(...)) ... iu=reshape(...)

Wanting to avoid the deprecated EQUIVALENCE statement, we must
reshape iv into a two-dimensional array, then un-reshape it after we
are done. This is done only once, at initialization time, so there is no serious
inefficiency introduced.

* * *

SUBROUTINE vegas(region,func,init,ncall,itmx,nprn,tgral,sd,chi2a)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: region
INTEGER(I4B), INTENT(IN) :: init,ncall,itmx,nprn
REAL(SP), INTENT(OUT) :: tgral,sd,chi2a
INTERFACE
FUNCTION func(pt,wgt)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: pt
REAL(SP), INTENT(IN) :: wgt
REAL(SP) :: func
END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: ALPH=1.5_sp,TINY=1.0e-30_sp
INTEGER(I4B), PARAMETER :: MXDIM=10,NDMX=50
Performs Monte Carlo integration of a user-supplied d-dimensional function func
over a rectangular volume specified by region, a vector of length 2d consisting of d “lower left”
coordinates of the region followed by d “upper right” coordinates. The integration consists of
itmx iterations, each with approximately ncall calls to the function. After each iteration
the grid is refined; more than 5 or 10 iterations are rarely useful. The input flag init
signals whether this call is a new start, or a subsequent call for additional iterations (see
comments below). The input flag nprn (normally 0) controls the amount of diagnostic
output. Returned answers are tgral (the best estimate of the integral), sd (its standard
deviation), and chi2a (χ² per degree of freedom, an indicator of whether consistent results
are being obtained). See text for further details.
INTEGER(I4B), SAVE :: i,it,j,k,mds,nd,ndim,ndo,ng,npg
Best make everything static,
INTEGER(I4B), DIMENSION(MXDIM), SAVE :: ia,kg
allowing restarts.
REAL(SP), SAVE :: calls,dv2g,dxg,f,f2,f2b,fb,rc,ti,tsi,wgt,xjac,xn,xnd,zo,harvest
REAL(SP), DIMENSION(NDMX,MXDIM), SAVE :: d,di,xi
REAL(SP), DIMENSION(MXDIM), SAVE :: dt,dx,x
REAL(SP), DIMENSION(NDMX), SAVE :: r,xin
REAL(DP), SAVE :: schi,si,swgt
ndim = size(region)/2

if (init <= 0) then
    mds = 1
    ndo = 1
    xi(1,:) = 1.0
end if

if (init <= 1) then
    si = 0.0
    splt = 0.0
    schi = 0.0
end if

if (init <= 2) then
    Enter here to inherit the previous grid and its
    answers.
    nd = NDMX
    ng = 1
    if (mds != 0) then
        ng = (ncall/2.0_sp+0.25_sp)**(1.0_sp/ndim)
        if ((2*ng-NDMX) >= 0) then
            mds = -1
            npg = ng/NDMX+1
            nd = ng/npg
            ng = npg*nd
        end if
    end if
    k = ng**ndim
    npg = max(ncall/k, 2)
    calls = real(npg, sp)*real(k, sp)
    dxg = 1.0_sp/ng
    dv2g = (calls*dxg**ndim)**2/npg/npg/(npg-1.0_sp)
    xnd = nd
    dx = dxg*xnd
    dx(1:ndim) = region(1+ndim:2*ndim) - region(1:ndim)
    xjac = 1.0_sp/calls*product(dx(1:ndim))
    if (nd /= ndo) then
        Do binning if necessary.
        r(1:max(nd, ndo)) = 1.0
        do j = 1, ndim
            call rebin(ndo/xnd, nd, r, xin, xi(:, j))
        end do
        nd = nd
    end if
    if (nprn >= 0) write(*, 200) ndim, calls, it, itmx, nprn, &
        ALPH, mds, nd, (j, region(j), j, region(j+ndim), j=1, ndim)
end if

do it = 1, itmx
    Main iteration loop. Can enter here (init >= 3) to do an additional itmx iterations
    with all other parameters unchanged.
    ti = 0.0
    tsi = 0.0
    kg(,:) = 1
    d(1:nd, :) = 0.0
    di(1:nd, :) = 0.0
    iterate: do
        fb = 0.0
        f2b = 0.0
        do k = 1, npg
            wgt = xjac
            do j = 1, ndim
                call rani(harvest)
                xn = (kg(j)-harvest)*dxg+1.0_sp
                ia(j) = max(min(int(xn), NDMX), 1)
                if (ia(j) > 1) then
                    xo = xi(ia(j), j)-xi(ia(j)-1, j)
                    rc = xi(ia(j)-1, j)*(xn-ia(j))*xo
                else
                    xo = xi(ia(j), j)
                    rc = (xn-ia(j))*xo
end if
x(j)=region(j)+rc*dx(j)
ugt=ugt*xo*xnd
end do
f=ugt*func(x(1:ndim),ugt)
f2=f+f
f2b=f2b+f2
do j=1,ndim
di(ia(j),j)=di(ia(j),j)+f
if (mds >= 0) d(ia(j),j)=d(ia(j),j)+f2
end do
def=f2b
f2b=sqrt(f2b*npg)
if (mds < 0) then
Use stratified sampling.
do j=1,ndim
d(ia(j),j)=d(ia(j),j)+f2b
end do
end if
end do
end do
df2b=sqrt(df2b/npg)
df2b=(df2b-fb)*(df2b+fb)
if (df2b <= 0.0) df2b=TINY
ti=ti+fb
tsi=tsi+df2b
if (mds < 0) then
Use stratified sampling.
do j=1,ndim
d(ia(j),j)=d(ia(j),j)+df2b
end do
end if
do k=ndim,1,-1
kg(k)=mod(kg(k),ng)+1
if (kg(k) /= 1) cycle iterate
end do
exit iterate
end do
Compute final results for this iteration.
tsi=tsi*dv2g
wgt=1.0_sp/tsi
si=si+real(wgt,dp)*real(ti,dp)
schi=schi+real(wgt,dp)**2
swgt=swgt+real(wgt,dp)
tgral=si/swgt
chid=max((schi-si*tgral)/(it-0.99_dp),0.0_dp)
sd=sqrt(1.0_sp/swgt)
tsi=sqrt(tsi)
if (nprn >= 0) then
write(*,201) it,ti,tsi,tgral,sd,chid
if (nprn /= 0) then
do j=1,ndim
write(*,202) j,(xi(i,j),di(i,j),&
i=1+nprn/2,nd,nprn)
end do
end if
end if
do j=1,ndim
refine the grid. Consult references to understand the subtlety of this procedure. The refinement is damped, to avoid rapid, destabilizing changes, and also compressed in range by the exponent ALPH.
xo=d(1,j)
xn=d(2,j)
d(1,j)=(xo+xn)/2.0_sp
dt(j)=d(1,j)
do i=2,nd-1
rc=xo+xn
xn=xn
d(i+1,j)=(rc+xn)/3.0_sp
dt(j)=dt(j)+d(i,j)
end do
d(nd,j)=(xo+xn)/2.0_sp
dt(j)=dt(j)+d(nd,j)
end do
where (d(1:nd,:) < TINY) d(1:nd,:)=TINY
do j=1,ndim
r(1:nd)=((1.0_sp-d(1:nd,j)/dt(j))/(log(dt(j))-log(d(1:nd,j))))**alpha
rc=sum(r(1:nd))
call rebin(rc/xnd,nd,r,xin,x(:,j))
end do
end do

200 format(/' input parameters for vegas: ndim=',i3,' ncall=',f8.0&
/28x,' it=',i5,' itmx=',i5&
/28x,' pprn=',i3,' alpha=',f5.2,' mds=',i3,' nd=',i4&
/30x,' xl(',i2,')= ',g11.4,' xu(',i2,')= ',g11.4)
201 format(/' iteration no.',i3,': ','integral =',g14.7,' +/- ',g9.2,&
/' all iterations: integral =',g14.7,' +/- ',g9.2,&
' chi**2/it''n =',g9.2)
202 format(/' data for axis ',i2/' X delta i ',&
' x delta i ',&
'(lx,f7.5,1x,gi1.4,x,f7.5,1x,gi1.4,x,f7.5,1x,gi1.4))
CONTAINS

SUBROUTINE rebin(rc,nd,r,xin,xi)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: rc
INTEGER(I4B), INTENT(IN) :: nd
REAL(SP), DIMENSION(:), INTENT(IN) :: r
REAL(SP), DIMENSION(:), INTENT(INOUT) :: xi

Utility routine used by vegas, to rebin a vector of densities xi into new bins defined by a vector r.

INTEGER(I4B) :: i,k
REAL(SP) :: dr,xn,xo
k=0
xo=0.0
dr=0.0

do i=1,nd-1
  do
    if (rc <= dr) exit
    k=k+1
    dr=dr+r(k)
  end do
  if (k > 1) xo=xi(k-1)
  xn=xi(k)
  dr=dr-rc
  xi(i)=(xn-xo)*dr/r(k)
end do
xi(1:nd-1)=xn
xi(nd)=1.0
END SUBROUTINE rebin

END SUBROUTINE vegas

* * *

RECURSIVE SUBROUTINE miser(func,regn,ndim,npts,dith,ave,var)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP) :: func
REAL(SP), DIMENSION(:), INTENT(IN) :: x
END FUNCTION func
END INTERFACE
REAL(SP), DIMENSION(:), INTENT(IN) :: regn
INTEGER(I4B), INTENT(IN) :: ndim,npts

Monte Carlo samples a user-supplied ndim-dimensional function func in a rectangular volume specified by region, a 2×ndim vector consisting of ndim "lower-left" coordinates of the region followed by ndim "upper-right" coordinates. The function is sampled a total of npts times, at locations determined by the method of recursive stratified sampling. The mean value of the function in the region is returned as ave; an estimate of the statistical uncertainty of ave (square of standard deviation) is returned as var. The input parameter dith should normally be set to zero, but can be set to (e.g.) 0.1 if func's active region falls on the boundary of a power-of-2 subdivision of region.

Parameters: PFAC is the fraction of remaining function evaluations used at each stage to explore the variance of func. At least MNPT function evaluations are performed in any terminal subregion; a subregion is further bisected only if at least MNBS function evaluations are available.

```
REAL(SP), INTENT(IN) :: dith
REAL(SP), INTENT(OUT) :: ave, var
REAL(SP), PARAMETER :: PFAC=0.1_sp, TINY=1.0e-30_sp, BIG=1.0e30_sp
INTEGER(I4B), PARAMETER :: MNPT=15, MNBS=60
```

Monte Carlo samples a user-supplied ndim-dimensional function func in a rectangular volume specified by region, a 2×ndim vector consisting of ndim "lower-left" coordinates of the region followed by ndim "upper-right" coordinates. The function is sampled a total of npts times, at locations determined by the method of recursive stratified sampling. The mean value of the function in the region is returned as ave; an estimate of the statistical uncertainty of ave (square of standard deviation) is returned as var. The input parameter dith should normally be set to zero, but can be set to (e.g.) 0.1 if func's active region falls on the boundary of a power-of-2 subdivision of region.

Parameters: PFAC is the fraction of remaining function evaluations used at each stage to explore the variance of func. At least MNPT function evaluations are performed in any terminal subregion; a subregion is further bisected only if at least MNBS function evaluations are available.

```
REAL(SP), DIMENSION(:,), ALLOCATABLE :: regn_temp
INTEGER(I4B) :: j, jb, n, ndim, npre, nptl, nptr
INTEGER(I4B), SAVE :: iran=0
REAL(SP) :: avel, varl, fracl, fval, rgl, rgn, rgr, &
  s, sigl, sigbr, sigl, sigl, sm, sm2, snum, sumr
REAL(SP), DIMENSION(:,), ALLOCATABLE :: fmaxl, fmaxr, fminl, fminr, pt, rmid
ndum=assert_eq(size(regn),2*ndim,'miser')
allocate(pt(ndim))
if (npts < MNBS) then
  Too few points to bisect; do straight Monte Carlo.
  sm=0.0
  sm2=0.0
  do n=1,npts
    call ranpt(pt,regn)
    fval=func(pt)
    sm=sm+fval
    sm2=sm2+fval**2
  end do
  ave=sm/npts
  var=max(TINY,(sm2-sm**2/npts)/npts**2)
else
  Do the preliminary (uniform) sampling.
  npre=max(int(npts*PFAC),MNPT)
  allocate(rmid(ndim),fmaxl(ndim),fmaxr(ndim),fminl(ndim),fminr(ndim))
  fminl(:)=BIG
  Initialize the left and right bounds for each dimension.
  fmaxl(:)=BIG
  fminr(:)=BIG
  fmaxr(:)=BIG
  do j=1,ndim
    iran=mod(iran*2661+36979,175000)
    s=sign(dith,real(iran-87500,sp))
    rmid(j)=(0.5_sp+s)*regn(j)+(0.5_sp-s)*regn(ndim+j)
  end do
  do n=1,npre
    Loop over the points in the sample.
    call ranpt(pt,regn)
    fval=func(pt)
    where (pt <= rmid)
      Find the left and right bounds for each dimension.
      fminl=min(fminl,fval)
      fmaxl=max(fmaxl,fval)
    elsewhere
      fminr=min(fminr,fval)
      fmaxr=max(fmaxr,fval)
    end where
    snum=0.0
    Choose which dimension jb to bisect.
    jb=0
    siglb=1.0
    sigbr=1.0
    do j=1,ndim
      if (fmaxl(j) > fminl(j) .and. fmaxr(j) > fminr(j)) then
```
sigl = max(TINY, (fmaxl(j) - fminl(j))**(2.0_sp/3.0_sp))
sigr = max(TINY, (fmaxr(j) - fminr(j))**(2.0_sp/3.0_sp))
sumr = sigl + sigr  \quad \text{Equation (7.8.24); see text.}
if (sumr <= sumb) then
  sumb = sumr
  jb = j
  siglb = sigl
  sigrb = sigr
end if
end if
deallocate(fminr, fminl, fmaxr, fmaxl)
if (jb == 0) jb = 1 + (ndim + iran) / 175000  \quad \text{MNPT may be too small.}
rgl = regn(jb)
rgm = rmid(jb)
rgr = regn(ndim + jb)
frac = abs((rgm - rgl) / (rgr - rgl))
nptl = (MNPT * npts - npre - 2 * MNPT) * frac * siglb / &
       (frac * siglb + (1.0_sp - frac) * sigrb)  \quad \text{Equation (7.8.23).}
nptr = npts - npre - nptl
allocate(regn_temp(2 * ndim))
regn_temp(jb) = regn(jb)  \quad \text{Set region to left.}
call miser(func, regn_temp, ndim, nptl, dith, avel, varl)
  Dispatch recursive call; will return back here eventually.
regn_temp(jb) = rmid(jb)
regn_temp(ndim + jb) = regn(ndim + jb)  \quad \text{Set region to right.}
call miser(func, regn_temp, ndim, nptr, dith, ave, var)
  Dispatch recursive call; will return back here eventually.
ave = frac * avel + (1.0 - frac) * ave  \quad \text{Combine left and right regions by equation (7.8.11) (1st line).}
var = frac * frac * varl + (1.0 - frac) * (1.0 - frac) * var
   \quad \text{(7.8.11) (1st line).}
deallocate(regn_temp)
end if
deallocate(pt)
CONTAINS
SUBROUTINE ranpt(pt, region)
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(OUT) :: pt
REAL(SP), DIMENSION(:,), INTENT(IN) :: region
Returns a uniformly random point pt in a rectangular region of dimension d. Used by
miser; calls ran1 for uniform deviates.
INTEGER(I4B) :: n
call ran1(pt)
n = size(pt)
pt(1:n) = region(1:n) + (region(n+1:2*n) - region(1:n)) * pt(1:n)
END SUBROUTINE ranpt
END SUBROUTINE miser

The Fortran 90 version of this routine is much more straightforward than
the Fortran 77 version, because Fortran 90 allows recursion. (In fact,
this routine is modeled on the C version of miser, which was recursive
from the start.)

CITED REFERENCES AND FURTHER READING:
Chapter B8. Sorting

Caution! If you are expecting to sort efficiently on a parallel machine, whether its parallelism is small-scale or massive, you almost certainly want to use library routines that are specific to your hardware.

We include in this chapter translations into Fortran 90 of the general purpose serial sorting routines that are in Volume 1, augmented by several new routines that give pedagogical demonstrations of how parallel sorts can be achieved with Fortran 90 parallel constructions and intrinsics. However, we intend the above word “pedagogical” to be taken seriously: these new, supposedly parallel, routines are not likely to be competitive with machine-specific library routines. Neither do they compete successfully on serial machines with the all-serial routines provided (namely sort, sort2, sort3, indexx, and select).

⋆⋆⋆

SUBROUTINE sort_pick(arr)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
Sorts an array arr into ascending numerical order, by straight insertion. arr is replaced on output by its sorted rearrangement.
INTEGER(I4B) :: i,j,n
REAL(SP) :: a
n=size(arr)
do j=2,n
   Pick out each element in turn.
a=arr(j)
do i=j-1,1,-1
      Look for the place to insert it.
      if (arr(i) <= a) exit
      arr(i+1)=arr(i)
   end do
   Insert it.
   arr(i+1)=a
end do
END SUBROUTINE sort_pick

Not only is sort_pick (renamed from Volume 1’s piksort) not parallelizable, but also, even worse, it is an \( N^2 \) routine. It is meant to be invoked only for the most trivial sorting jobs, say, \( N < 20 \).

⋆⋆⋆
SUBROUTINE sort_shell(arr)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
Sort an array arr into ascending numerical order by Shell’s method (diminishing increment sort). arr is replaced on output by its sorted rearrangement.
INTEGER(I4B) :: i,j,inc,n
REAL(SP) :: v
n=size(arr)
inc=1
do
  inc=3*inc+1
  if (inc > n) exit
end do
do
determine the starting increment.
  inc=inc/3
  do i=inc+1,n
    outer loop of straight insertion.
    v=arr(i)
    j=i
    do
      inner loop of straight insertion.
      if (arr(j-inc) <= v) exit
      arr(j)=arr(j-inc)
      j=j-inc
      if (j <= inc) exit
    end do
    arr(j)=v
  end do
  if (inc <= 1) exit
end do
END SUBROUTINE sort_shell

The routine sort_shell is renamed from Volume 1’s shell. Shell’s Method, a diminishing increment sort, is not directly parallelizable. However, one can write a fully parallel routine (though not an especially fast one — see remarks at beginning of this chapter) in much the same spirit:

SUBROUTINE sort_byreshape(arr)
USE nrtype; USE nrutil, ONLY : swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
Sort an array arr by bubble sorting a succession of reshapings into array slices. The method is similar to Shell sort, but allows parallelization within the vectorized masked swap calls.
REAL(SP), DIMENSION(:,:), ALLOCATABLE :: tab
REAL(SP), PARAMETER :: big=huge(arr)
INTEGER(I4B) :: inc,n,m
n=size(arr)
inc=1
do
  find the largest increment that fits.
    inc=2*inc+1
    if (inc > n) exit
  end do
  find the different shapes for the reshaped array.
  allocate(tab(inc,m))
  allocate space and reshape the array big end
  tab=reshape(arr, (/inc,m/) , (/big/) )
  bubble sort all the rows in parallel.
call swap(tab(:,1:m-1:2),tab(:,2:m:2), &
  tab(:,1:m-1:2)>tab(:,2:m:2))
call swap(tab(:,2:m-1:2),tab(:,3:m:2), &
  tab(:,2:m-1:2)>tab(:,3:m:2))
end do
The basic idea is to reshape the given one-dimensional array into a succession of two-dimensional arrays, starting with “tall and narrow” (many rows, few columns), and ending up with “short and wide” (many columns, few rows). At each stage we sort all the rows in parallel by a bubble sort, giving something close to Shell’s diminishing increments.

We now arrive at those routines, based on the Quicksort algorithm, that we actually intend for use with general $N$ on serial machines:

```fortran
SUBROUTINE sort(arr)
USE nrtype; USE nrutil, ONLY : swap,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
INTEGER(I4B), PARAMETER :: NN=15, NSTACK=50

Sorts an array $arr$ into ascending numerical order using the Quicksort algorithm. $arr$ is replaced on output by its sorted rearrangement.
Parameters: $NN$ is the size of subarrays sorted by straight insertion and $NSTACK$ is the required auxiliary storage.

REAL(SP) :: a
INTEGER(I4B) :: n,k,i,j,jstack,l,r
INTEGER(I4B), DIMENSION(NSTACK) :: istack
n=size(arr)
jstack=0
l=1
r=n

if (r-l < NN) then
  Insertion sort when subarray small enough.
  do j=l+1,r
    a=arr(j)
    do i=j-1,l,-1
      if (arr(i) <= a) exit
      arr(i+1)=arr(i)
    end do
    arr(i+1)=a
  end do
  if (jstack == 0) RETURN
  r=istack(jstack)
  Pop stack and begin a new round of partitioning.
  l=istack(jstack-1)
  jstack=jstack-2
else
  Choose median of left, center, and right elements as partitioning element $a$. Also rearrange so that $a(l) \leq a(l+1) \leq a(r)$.
  k=(l+r)/2
  call swap(arr(k),arr(l+1))
  call swap(arr(l),arr(l+1))
  call swap(arr(l+1),arr(r),arr(l+1)>arr(r))
  i=l+1
  j=r
  a=arr(l+1)
  do
    i=i+1
    if (arr(i) > a) exit
    call swap(arr(i),arr(i-1))
  end do
  if (i > l+NN/2) a=arr(i)
  j=i
  i=l
  do
    if (arr(i) > a) exit
    call swap(arr(i),arr(i-1))
    i=i+1
  end do
  k=(i+j)/2
  call swap(arr(k),arr(l+1))
  call swap(arr(l),arr(l+1))
  call swap(arr(l+1),arr(r),arr(l+1)>arr(r))
  call swap(arr(l),arr(l+1),arr(l)>arr(l+1))
  j=r
  a=arr(l+1)
  do
    j=j-1
    if (arr(j) < a) exit
    call swap(arr(j),arr(j+1))
  end do
  If necessary, exchange $a$ for med
  end do
end do
END SUBROUTINE sort
```
Chapter B8. Sorting

if (arr(i) >= a) exit
end do

j=j-1
if (arr(j) <= a) exit
end do

if (j < i) exit
Pointers crossed. Exit with partitioning complete.
call swap(arr(i),arr(j))
Exchange elements.

arr(l+1)=arr(j)
Insert partitioning element.

arr(j)=a
jstack=jstack+2
Push pointers to larger subarray on stack; process smaller subarray immediately.
if (jstack > NSTACK) call nrerror('sort: NSTACK too small')
if (r-i+1 >= j-l) then
istack(jstack)=r
istack(jstack-1)=i
r=j-1
else
istack(jstack)=j-1
istack(jstack-1)=l
l=i
end if

end if
end do

END SUBROUTINE sort

One might think twice about putting all these external function calls (to nrutil routines) in the inner loop of something as streamlined as a sort routine, but here they are executed only once for each partitioning.

call swap(arr(i),arr(j)) This call is in a loop, but not the innermost loop. Most modern machines are very fast at the “context changes” implied by subroutine calls and returns; but in a time-critical context you might code this swap in-line and see if there is any timing difference.

SUBROUTINE sort2(arr,slave)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : indexx
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave
Sorts an array arr into ascending order using Quick sort, while making the corresponding rearrangement of the same-size array slave. The sorting and rearrangement are performed by means of an index array.

INTEGER(I4B) :: ndum
INTEGER(I4B), DIMENSION(size(arr)) :: index
ndum=assert_eq(size(arr),size(slave),'sort2')
call indexx(arr,index) Make the index array.
arr=arr(index)
Sort arr.
slave=slave(index) Rearrange slave.
END SUBROUTINE sort2

A close surrogate for the Quicksort partition-exchange algorithm can be coded, parallelizable, by using Fortran 90’s pack intrinsic. On real compilers, unfortunately, the resulting code is not very efficient as compared with (on serial machines) the tightness of sort’s inner loop, above, or (on parallel machines) supplied library sort routines. We illustrate the principle nevertheless in the following routine.
RECURSIVE SUBROUTINE sort_bypack(arr)
USE nrtype; USE nrutil, ONLY : array_copy, swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
Sort an array arr by recursively applying the Fortran 90 pack intrinsic. The method is similar to Quicksort, but this variant allows parallelization by the Fortran 90 compiler.
REAL(SP) :: a
INTEGER(I4B) :: n,k,nl,nerr
INTEGER(I4B), SAVE :: level=0
LOGICAL, DIMENSION(:), ALLOCATABLE, SAVE :: mask
REAL(SP), DIMENSION(:), ALLOCATABLE, SAVE :: temp
n=size(arr)
if (n <= 1) RETURN
k=(1+n)/2
call swap(arr(1),arr(k),arr(1)>arr(k)) Pivot element is median of first, middle, and last.
call swap(arr(k),arr(n),arr(k)>arr(n))
call swap(arr(1),arr(k),arr(1)>arr(k))
if (n <= 3) RETURN
level=level+1 Keep track of recursion level to avoid allocation overhead.
if (level == 1) allocate(mask(n),temp(n))
a=arr(k)
mask(1:n) = (arr <= a) Which elements move to left?
mask(k) = .false.
call array_copy(pack(arr,mask(1:n)),temp,nl,nerr) Move them.
mask(k) = .true.
temp(nl+2:n)=pack(arr,.not. mask(1:n)) Move others to right.
temp(nl+1)=a
arr=temp(1:n)
call sort_bypack(arr(1:nl)) And recurse.
call sort_bypack(arr(nl+2:n))
if (level == 1) deallocate(mask,temp)
level=level-1
END SUBROUTINE sort_bypack

* * *

The following routine, sort_heap, is renamed from Volume 1’s hpsort.

SUBROUTINE sort_heap(arr)
USE nrtype
USE nrutil, ONLY : swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
Sorts an array arr into ascending numerical order using the Heapsort algorithm. arr is replaced on output by its sorted rearrangement.
INTEGER(I4B) :: i,n
n=size(arr)
do i=n/2,1,-1
The index i, which here determines the “left” range of the sift-down, i.e., the element to be sifted down, is decremented from n/2 down to 1 during the “hiring” (heap creation) phase.
call sift_down(i,n)
end do
do i=n,2,-1
Here the “right” range of the sift-down is decremented from n-1 down to 1 during the “retirement-and-promotion” (heap selection) phase.
call swap(arr(i),arr(1)) Clear a space at the end of the array, and
call sift_down(1,i-1) retire the top of the heap into it.
end do
CONTAINS
SUBROUTINE sift_down(i,n)
INTEGER(I4B), INTENT(IN) :: i,n
Carry out the sift-down on element \texttt{arr(l)} to maintain the heap structure.

\begin{verbatim}
INTEGER(I4B) :: j, jold
REAL(SP) :: a
a = \texttt{arr(l)}
jold = l
j = 1 + l

\textbf{do}  
  \textbf{if} (\texttt{j > r}) \textbf{exit}
  \textbf{if} (\texttt{j < r}) \textbf{then}
    \textbf{if} (\texttt{arr(j)} < \texttt{arr(j+1)}) \texttt{j} = \texttt{j+1}
  \textbf{end if}
  \textbf{if} (\texttt{a} >= \texttt{arr(j)}) \textbf{exit}
  \texttt{arr(jold)} = \texttt{arr(j)}
  \texttt{jold} = \texttt{j}
  \texttt{j} = \texttt{j} + \texttt{j}
\textbf{end do}
\texttt{arr(jold)} = \texttt{a}
\end{verbatim}

Another opportunity provided by Fortran 90 for a fully parallelizable sort, at least pedagogically, is to use the language's allowed access to the actual floating-point representation and to code a radix sort \cite{1} on its bits. This is not efficient, but it illustrates some Fortran 90 language features perhaps worthy of study for other applications.

\begin{verbatim}
SUBROUTINE sort_radix(arr)
  \textbf{USE} nrtype; \textbf{USE} nrutil, \textbf{ONLY} : \texttt{array_copy}, \texttt{nrerror}
  \textbf{IMPLICIT} \textbf{NONE}
  \textbf{REAL(SP), DIMENSION(:), INTENT(INOUT)} :: \texttt{arr}
  \textbf{Sort an array }\texttt{arr} \textbf{by radix sort on its bits.}
  \textbf{INTEGER(I4B), DIMENSION(size(arr)) :: narr, temp}
  \textbf{LOGICAL, DIMENSION(size(arr)) :: msk}
  \textbf{INTEGER(I4B) :: k, negm, ib, ia, n, nl, nerr}

  \textbf{Because we are going to transfer reals to integers, we must check that the number of bits is the same in each:}
  \texttt{ib} = \texttt{bit_size(narr)}
  \texttt{ia} = \texttt{ceiling(log(real(max exponent(arr) - min exponent(arr), sp))/log(2.0_sp))} \texttt{&
    digits(arr)}

  \textbf{if (ib /= ia) call nrerror('sort_radix: bit sizes not compatible')}
  \texttt{negm} = \texttt{not(ishftc(1,-1))}

  \texttt{n} = \texttt{size(arr)}
  \texttt{narr} = \texttt{transfer(arr, narr, n)}

  \textbf{where (btest(narr, ib-1)) narr = ieor(narr, negm)}
  \textbf{Flip all bits on neg. numbers.}
  \textbf{do k = 0, ib-2}

    \textbf{Work from low- to high-order bits, and partition the array according to the value of the bit.}
    \texttt{msk} = \texttt{btest(narr, k)}
    \textbf{call array_copy(pack(narr, not. msk), temp, nl, nerr)}
    \texttt{temp(nl+1:n) = pack(narr, msk)}
    \texttt{narr = temp}
  \textbf{end do}

  \texttt{msk} = \texttt{btest(narr, ib-1)}
  \textbf{call array_copy(pack(narr, msk), temp, nl, nerr)}

  \textbf{The sign bit gets separate treatment, since here 1 comes before 0.}
  \texttt{temp(nl+1:n) = pack(narr, not. msk)}
  \texttt{narr = temp}

  \texttt{where (btest(narr, ib-1)) narr = ieor(narr, negm)}
  \texttt{Unflip all bits on neg. numbers.}
  \texttt{arr} = \texttt{transfer(narr, arr, n)}
\end{verbatim}
We overload the generic name \texttt{indexx} with two specific implementations, one for \texttt{SP} floating values, the other for \texttt{I4B} integers. (You can of course add more overloading if you need them.)

\begin{verbatim}
SUBROUTINE indexx_sp(arr,index)
USE nrttype; USE nrutil, ONLY : arth,assert_eq,nrerror,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: index
INTEGER(I4B), PARAMETER :: NN=15, NSTACK=50

Indexes an array \texttt{arr}, i.e., outputs the array index of length \(N\) such that \texttt{arr(index(j))} is in ascending order for \(j = 1,2,\ldots,N\). The input quantity \texttt{arr} is not changed.

REAL(SP) :: a
INTEGER(I4B) :: n,k,i,j,indext,jstack,l,r
INTEGER(I4B), DIMENSION(NSTACK) :: istack
n=assert_eq(size(index),size(arr),'indexx_sp')
index=arth(1,1,n)
jstack=0
l=1
r=n
do
if (r-l < NN) then
    do j=l+1,r
        indext=index(j)
        a=arr(indext)
        do i=j-1,l,-1
            if (arr(index(i)) <= a) exit
            index(i+1)=index(i)
        end do
        index(i+1)=indext
    end do
    if (jstack == 0) RETURN
    r=istack(jstack)
    l=istack(jstack-1)
    jstack=jstack-2
else
    k=(l+r)/2
    call swap(index(k),index(l+1))
    call icomp_xchg(index(l),index(r))
    call icomp_xchg(index(l+1),index(r))
    call icomp_xchg(index(l),index(l+1))
    i=l+1
    j=r
    indext=index(l+1)
    a=arr(indext)
    do
        i=i+1
        if (arr(index(i)) >= a) exit
    end do
    j=j-1
    if (arr(index(j)) <= a) exit
    if (j < i) exit
    call swap(index(i),index(j))
end do
index(l+1)=index(j)
index(j)=indext
jstack=jstack+2
if (jstack > NSTACK) call nrerror('indexx: NSTACK too small')
if (r-l+1 >= j-l) then
    istack(jstack)=r
end if
end do
end do
end subroutine indexx_sp
\end{verbatim}
istack(jstack-1)=i
r=j-1
else
istack(jstack)=j-1
istack(jstack-1)=l
l=i
end if
end if
end do
CONTAINS
SUBROUTINE icsmp_xchg(i,j)
INTEGER(I4B), INTENT(INOUT) :: i,j
INTEGER(I4B) :: swp
if (arr(j) < arr(i)) then
swp=i
i=j
j=swp
end if
END SUBROUTINE icsmp_xchg
END SUBROUTINE indexx_sp

SUBROUTINE icsmp_xchg(iarr,index)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror,swap
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iarr
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: index
INTEGER(I4B), PARAMETER :: NN=15, NSTACK=50
INTEGER(I4B) :: a
INTEGER(I4B) :: n,k,i,j,indext,jstack,l,r
INTEGER(I4B), DIMENSION(NSTACK) :: istack
n=assert_eq(size(index),size(iarr),'indexx_sp')
index=arth(1,1,n)
jstack=0
l=1
r=n
do
if (r-l < NN) then
do j=l+1,r
indext=index(j)
a=iarr(indext)
do i=j-1,l,-1
if (iarr(index(i)) <= a) exit
index(i+1)=index(i)
end do
index(i+1)=indext
end do
if (jstack == 0) RETURN
r=istack(jstack)
l=istack(jstack-1)
jstack=jstack-2
else
k=(l+r)/2
call swap(index(k),index(l+1))
call icsmp_xchg(index(l),index(k))
call icsmp_xchg(index(k),index(l+1))
call icsmp_xchg(index(l),index(1+1))
i=l+1
j=r
indext=index(l+1)
a=iarr(indext)
do
i=i+1
    if (iarr(index(i)) >= a) exit
end do
    j=j-1
    if (iarr(index(j)) <= a) exit
end do
if (j < i) exit
    call swap(index(i),index(j))
end do
    index(l+1)=index(j)
    index(j)=indext
    jstack=jstack+2
    if (jstack > NSTACK) call nrerror('indexx: NSTACK too small')
if (r-i+1 >= j-l) then
    istack(jstack)=r
    istack(jstack-1)=i
    r=j-1
else
    istack(jstack)=j-1
    istack(jstack-1)=l
    l=i
end if
end if
end do
CONTAINS
SUBROUTINE icomp_xchg(i,j)
INTEGER(I4B), INTENT(INOUT) :: i,j
INTEGER(I4B) :: swp
if (iarr(j) < iarr(i)) then
    swp=i
    i=j
    j=swp
end if
END SUBROUTINE icomp_xchg
END SUBROUTINE indexx_i4b

SUBROUTINE sort3(arr,slave1,slave2)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : indexx
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave1,slave2
Sorts an array arr into ascending order using Quicksort, while making the corresponding rearrangement of the same-size arrays slave1 and slave2. The sorting and rearrangement are performed by means of an index array.
INTEGER(I4B) :: ndum
INTEGER(I4B), DIMENSION(size(arr)) :: index
ndum=assert_eq(size(arr),size(slave1),size(slave2),'sort3')
call indexx(arr,index)        Make the index array.
arr=arr(index)                Sort arr.
slave1=slave1(index)          Rearrange slave1,
slave2=slave2(index)          and slave2.
END SUBROUTINE sort3

* * *
FUNCTION rank(index)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: index
INTEGER(I4B), DIMENSION(size(index)) :: rank
Given index as output from the routine indexx, this routine returns a same-size array rank, the corresponding table of ranks.
rank(index(:))=arth(1,1,size(index))
END FUNCTION rank

⋆⋆⋆

Just as in the case of sort, where an approximation of the underlying Quicksort partition-exchange algorithm can be captured with the Fortran 90 pack intrinsic, the same can be done with indexx. As before, although it is in principle parallelizable by the compiler, it is likely not competitive with library routines.

RECURSIVE SUBROUTINE index_bypack(arr,index,partial)
USE nrtype; USE nrutil, ONLY : array_copy,arth,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: index
INTEGER, OPTIONAL, INTENT(IN) :: partial
Indexes an array arr, i.e., outputs the array index of length N such that arr(index(j)) is in ascending order for j = 1, 2, ..., N. The method is to apply recursively the Fortran 90 pack intrinsic. This is similar to Quicksort, but allows parallelization by the Fortran 90 compiler. partial is an optional argument that is used only internally on the recursive calls.
REAL(SP) :: a
INTEGER(I4B) :: n,k,nl,indext,nerr
INTEGER(I4B), SAVE :: level=0
LOGICAL, DIMENSION(:), ALLOCATABLE, SAVE :: mask
INTEGER(I4B), DIMENSION(:), ALLOCATABLE, SAVE :: temp
if (present(partial)) then
    n=size(index)
else
    n=assert_eq(size(index),size(arr),'indexx_bypack')
end if
if (n <= 1) RETURN
k=(1+n)/2
call icomp_xchg(index(1),index(k)) Call the pivot element.
call icomp_xchg(index(k),index(n))
call icomp_xchg(index(1),index(k))
if (n <= 3) RETURN
level=level+1 Keep track of recursion level to avoid allocation overhead.
if (level == 1) allocate(mask(n),temp(n))
indext=index(k)
a=arr(indext)
index=x(index(k))
mask(1:n) = (arr(index) <= a) Which elements move to left?
mask(k) = .false.
call array_copy(pack(index,mask(1:n)),temp,nl,nerr) Move them.
mask(k) = .true.
temp(nl+2:n)=pack(index,.not. mask(1:n)) Move others to right.
temp(nl+1)=indext
index=temp(1:n)
call index_bypack(arr,index(1:nl),partial=1) And recurse.
call index_bypack(arr,index(nl+2:n),partial=1)
if (level == 1) deallocate(mask,temp)
level=level-1
CONTAINS
SUBROUTINE icomp_xchg(i,j)
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: i,j
    Swap or don't swap integer arguments, depending on the ordering of their corresponding elements in an array arr.
INTEGER(I4B) :: swp
    if (arr(j) < arr(i)) then
        swp=i
        i=j
        j=swp
    end if
END SUBROUTINE icomp_xchg
END SUBROUTINE index_bypack

FUNCTION select(k,arr)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: k
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
REAL(SP) :: select
    Returns the kth smallest value in the array arr. The input array will be rearranged to have this value in location arr(k), with all smaller elements moved to arr(1:k-1) (in arbitrary order) and all larger elements in arr(k+1:) (also in arbitrary order).
INTEGER(I4B) :: i,r,j,l,n
REAL(SP) :: a
    n=size(arr)
call assert(k >= 1, k <= n, 'select args')
l=1
r=n
do
    if (r-l <= 1) then
        Active partition contains 1 or 2 elements.
        select=arr(k)
        RETURN
    else
        Choose median of left, center, and right elements as partitioning element a. Also rearrange so that arr(l) ≤ arr(l+1) ≤ arr(r).
i=(l+r)/2
        call swap(arr(i),arr(l+1))
call swap(arr(l),arr(r),arr(l)>arr(r))
call swap(arr(l+1),arr(r),arr(l+1)>arr(r))
call swap(arr(l),arr(l+1),arr(l)>arr(l+1))
i=l+1
j=r
    a=arr(l+1)
    i=1
    do
        if (arr(i) >= a) exit
    end do
    do
        if (arr(j) <= a) exit
    end do
    if (j < i) exit
    Pointers crossed. Exit with partitioning complete.
    call swap(arr(i),arr(j))
end do
    arr(l+1)=arr(j)
    arr(j)=a
    if (j >= k) r=j-1
    Partitioning element.
    Exchange elements.
do
    i=i+1
    if (arr(i) >= a) exit
end do
    do
        if (arr(j) <= a) exit
end do
    if (j < i) exit
    Points crossed. Exit with partitioning complete.
call swap(arr(i),arr(j))
end do
    arr(l+1)=arr(j)
    arr(j)=a
    if (j >= k) r=j-1
    Keep active the partition that contains the kth element.
if (j <= k) l=i
end if
end do
END FUNCTION select

The following routine, select_inplace, is renamed from Volume 1’s selip.

FUNCTION select_inplace(k,arr)
USE nrtype
USE nr, ONLY : select
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: k
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
REAL(SP) :: select_inplace

Returns the kth smallest value in the array arr, without altering the input array. In Fortran 90’s assumed memory-rich environment, we just call select in scratch space.

REAL(SP), DIMENSION(size(arr)) :: tarr
tarr=arr
select_inplace=select(k,tarr)
END FUNCTION select_inplace

Volume 1’s selip routine uses an entirely different algorithm, for the purpose of avoiding any additional memory allocation beyond that of the input array. Fortran 90 presumes a richer memory environment, so select_inplace simply does the obvious (destructive) selection in scratch space. You can of course use the old selip if your in-core or in-cache memory is at a premium.

FUNCTION select_bypack(k,arr)
USE nrtype; USE nrutil, ONLY : array_copy,assert,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: k
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
REAL(SP) :: select_bypack

Returns the kth smallest value in the array arr. The input array will be rearranged to have this value in location arr(k), with all smaller elements moved to arr(1:k-1) (in arbitrary order) and all larger elements in arr(k+1:) (also in arbitrary order). This implementation allows parallelization in the Fortran 90 pack intrinsic.

LOGICAL, DIMENSION(size(arr)) :: mask
REAL(SP), DIMENSION(size(arr)) :: temp
INTEGER(I4B) :: i,r,j,l,n,nl,nerr
REAL(SP) :: a
data(size(arr))

l=1
r=n

do
if (r-l <= 1) exit
i=(l+r)/2
call swap(arr(l),arr(i),arr(l)>arr(i))  
 Pivot element is median of first, middle, and last.
call swap(arr(i),arr(r),arr(i)>arr(r))
call swap(arr(l),arr(i),arr(l)>arr(i))
a=arr(i)
mask(l:r) = (arr(l:r) <= a)
Which elements move to left?
mask(i) = .false.
call array_copypack(arr(l:r),mask(l:r)),temp(l:),nl,nerr)  
 Move them.
j=l+nl

do
The above routine `select_bypack` is parallelizable, but as discussed above (`sort_bypack`, `index_bypack`) it is generally not very efficient.

```
mask(i) = .true.
temp(j+1:r)=pack(arr(l:r),.not. mask(l:r)) Move others to right.
temp(j)=a
arr(l:r)=temp(l:r)
if (k > j) then
  l=j+1
else if (k < j) then
  r=j-1
else
  l=j
  r=j
end if
end do
if (r-l == 1) call swap(arr(l),arr(r),arr(l)>arr(r)) Case of only two left.
select_bypack=arr(k)
END FUNCTION select_bypack
```

The following routine, `select_heap`, is renamed from Volume I's `hpsel`.

```
SUBROUTINE select_heap(arr, heap)
USE nrtype; USE nrutil, ONLY : nrerror, swap
USE nr, ONLY : sort
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
REAL(SP), DIMENSION(:), INTENT(OUT) :: heap

Return in heap, an array of length \( M \), the largest \( M \) elements of the array \( \text{arr} \) of length \( N \), with \( \text{heap}(1) \) guaranteed to be the the \( M \)th largest element. The array \( \text{arr} \) is not altered. For efficiency, this routine should be used only when \( M \ll N \).

INTEGER(I4B) :: i,j,k,m,n
m=size(heap)
n=size(arr)
if (m > n/2 .or. m < 1) call nrerror('probable misuse of select_heap')
heap=arr(1:m)
call sort(heap) Create initial heap by overkill! We assume \( m \ll n \).
do i=m+1,n For each remaining element...
  if (arr(i) > heap(1)) then Put it on the heap?
    heap(1)=arr(i)
    j=1 Sift down.
    do k=2*j
        if (k > m) exit
        if (k /= m) then
          if (heap(k) > heap(k+1)) k=k+1 end if
        if (heap(j) <= heap(k)) exit
        call swap(heap(k),heap(j))
        j=k
    end do
  end if
end do
END SUBROUTINE select_heap
```

⋆⋆⋆
FUNCTION eclass(lista,listb,n)
USE nrtype; USE nrutil, ONLY : arth, assert_eq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: lista,listb
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), DIMENSION(n) :: eclass
Given $M$ equivalences between pairs of $n$ individual elements in the form of the input arrays lista and listb of length $M$, this routine returns in an array of length $n$ the number of the equivalence class of each of the $n$ elements, integers between 1 and $n$ (not all such integers used).

INTEGER :: j,k,l,m
m=assert_eq(size(lista),size(listb),'eclass')
eclass(1:n)=arth(1,1,n) Initialize each element its own class.
do i=1,m
  j=lista(i)
do
    if (eclass(j) == j) exit
    j=eclass(j)
  end do
  k=listb(i)
do
    if (eclass(k) == k) exit
    k=eclass(k)
  end do
  if (j /= k) eclass(j)=k If they are not already related, make them so.
end do
end do
Final sweep up to highest ancestors.
do
  if (eclass(j) == eclass(eclass(j))) exit
  eclass(j)=eclass(eclass(j))
end do
end do
END FUNCTION eclass

FUNCTION eclazz(equiv,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
INTERFACE
  FUNCTION equiv(i,j)
    USE nrtype
    IMPLICIT NONE
    LOGICAL(LGT) :: equiv
    INTEGER(I4B), INTENT(IN) :: i,j
  END FUNCTION equiv
END INTERFACE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), DIMENSION(n) :: eclazz
Given a user-supplied logical function equiv that tells whether a pair of elements, each in the range 1...$n$, are related, return in an array of length $n$ equivalence class numbers for each element.

INTEGER :: i,j
eclazz(1:n)=arth(1,1,n)
do i=2,n
  do j=1,i-1
    eclazz(j)=eclazz(eclazz(j)) Sweep up this much.
    if (equiv(i,j)) eclazz(eclass(eclazz(j)))=i Good exercise for the reader to figure out why this much ancestry is necessary!
  end do
end do
end do
Only this much sweeping is needed finally.
end do
END FUNCTION eclazz
CITED REFERENCES AND FURTHER READING:
Chapter B9. Root Finding and Nonlinear Sets of Equations

SUBROUTINE scrsho(func)
USE nrtype
IMPLICIT NONE
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE

INTEGER(I4B), PARAMETER :: ISCR=60, JSCR=21

For interactive 'dumb terminal' use. Produce a crude graph of the function func over the prompted-for interval x1,x2. Query for another plot until the user signals satisfaction.

Parameters: Number of horizontal and vertical positions in display.

INTEGER(I4B) :: i,j,jz
REAL(SP) :: dx,dyj,x,x1,x2,ybig,ysml
REAL(SP), DIMENSION(ISCR) :: y
CHARACTER(1), DIMENSION(ISCR,JSCR) :: scr
CHARACTER(1) :: blank=' ',zero='-',yy='l',xx='-',ff='x'

DO

WRITE (*,*) ' Enter x1,x2 (= to stop)'  ! Query for another plot; quit if x1=x2.
READ (*,*) x1,x2
IF (x1 == x2) RETURN
        SCR(1,1:JSCR)=YY
        SCR(ISCR,1:JSCR)=YY
        SCR(2:ISCR-1,1)=XX
        SCR(2:ISCR-1,JSCR)=XX
        SCR(2:ISCR-1,2:JSCR-1)=BLANK
        DX=(X2-X1)/(ISCR-1)
        X=X1
        DO I=1,ISCR
            Y(I)=FUNC(X)
            X=X+DX
        END DO
        YSMALL=MIN(MINVAL(Y(:)),0.0_SP)
        YBIG=MAX(MAXVAL(Y(:)),0.0_SP)
        IF (YBIG == YSMALL) YBIG=YSMALL+1.0
        DYJ=(JSCR-1)/(YBIG-YSMALL)
        JZ=1-YSMALL*DYJ
        SCR(I:ISCR,JZ)=ZERO
        DO I=1,ISCR
            J=1+(Y(I)-YSMALL)*DYJ
            SCR(I,J)=FF
        END DO
        WRITE (*,'(1X,1P,E10.3,1X,80A1)') YBIG,(SCR(I,JSCR),I=1,ISCR)
        DO J=JSCR-1,2,-1
            WRITE (*,'(12X,80A1)') (SCR(I,J),I=1,ISCR)
        END DO
        WRITE (*,'(1X,1P,E10.3,1X,80A1)') YSMALL,(SCR(I,1),I=1,ISCR)
        END DO

STOP
In Fortran 90, the length of variables of type character should be declared as CHARACTER(1) or CHARACTER(len=1) (for a variable of length 1), rather than the older form CHARACTER*1. While the older form is still legal syntax, the newer one is more consistent with the syntax of other type declarations. (For variables of length 1, you can actually omit the length specifier entirely, and just say CHARACTER.)

```fortran
SUBROUTINE zbrac(func,x1,x2,succes)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(INOUT) :: x1,x2
LOGICAL(LGT), INTENT(OUT) :: succes
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: NTRY=50
REAL(SP), PARAMETER :: FACTOR=1.6_sp
Given a function func and an initial guessed range x1 to x2, the routine expands the range geometrically until a root is bracketed by the returned values x1 and x2 (in which case succes returns as .true.) or until the range becomes unacceptably large (in which case succes returns as .false.).

INTEGER(I4B) :: j
REAL(SP) :: f1,f2
if (x1 == x2) call nrerror('zbrac: you have to guess an initial range')
f1=func(x1)
f2=func(x2)
succes=.true.
do j=1,NTRY
if ((f1 > 0.0 .and. f2 < 0.0) .or. &
   (f1 < 0.0 .and. f2 > 0.0)) RETURN
if (abs(f1) < abs(f2)) then
   x1=x1+FACTOR*(x1-x2)
f1=func(x1)
else
   x2=x2+FACTOR*(x2-x1)
f2=func(x2)
end if
end do
succes=.false.
END SUBROUTINE zbrac
```
SUBROUTINE zbrak(func,x1,x2,n,xb1,xb2,nb)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), INTENT(OUT) :: nb
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), DIMENSION(:), POINTER :: xb1,xb2
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE

Given a function func defined on the interval from x1-x2 subdivide the interval into n equally spaced segments, and search for zero crossings of the function. nb is returned as the number of bracketing pairs xb1(1:nb), xb2(1:nb) that are found. xb1 and xb2 are pointers to arrays of length nb that are dynamically allocated by the routine.

INTEGER(I4B) :: i
REAL(SP) :: dx
REAL(SP), DIMENSION(0:n) :: f,x
LOGICAL(LGT), DIMENSION(1:n) :: mask
LOGICAL(LGT), SAVE :: init=.true.
if (init) then
   init=.false.
nullify(xb1,xb2)
end if
if (associated(xb1)) deallocate(xb1)
if (associated(xb2)) deallocate(xb2)
dx=(x2-x1)/n
x=x1+dx*arth(0,1,n)
do i=0,n
   f(i)=func(x(i))
   end do
mask=f(1:n)*f(0:n-1) <= 0.0
   nb=count(mask)
   allocate(xb1(nb),xb2(nb))
xb1(1:nb)=pack(x(0:n-1),mask)
xb2(1:nb)=pack(x(1:n),mask)
END SUBROUTINE zbrak

This routine shows how to return arrays xb1 and xb2 whose size is not known in advance. The coding is explained in the subsection on pointers in §21.5.

FUNCTION rtbis(func,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtbis
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=40
Using bisection, find the root of a function func known to lie between x1 and x2. The root, returned as rtbis, will be refined until its accuracy is ±xacc.
Parameter: MAXIT is the maximum allowed number of bisections.

INTEGER(I4B) :: j
REAL(SP) :: dx,f,fmid,xmid

fmid=func(x2)
f=func(x1)
if (f*fmid >= 0.0) call nrerror('rtbis: root must be bracketed')
if (f < 0.0) then Orient the search so that f>0 lies at x+dx.
    rtbis=x1
dx=x2-x1
else rtbis=x2
dx=x1-x2
end if

do j=1,MAXIT Bisection loop.
dx=dx*0.5_sp
xmid=rtbis+dx
fmid=func(xmid)
    if (fmid <= 0.0) rtbis=xmid
    if (abs(dx) < xacc .or. fmid == 0.0) RETURN
end do
call nrerror('rtbis: too many bisections')
END FUNCTION rtbis

FUNCTION rtflsp(func,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror,swap
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtflsp
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE

INTEGER(I4B), PARAMETER :: MAXIT=30
Using the false position method, find the root of a function func known to lie between x1 and x2. The root, returned as rtflsp, is refined until its accuracy is ±xacc.
Parameter: MAXIT is the maximum allowed number of iterations.

INTEGER(I4B) :: j
REAL(SP) :: del,dx,f,fh,fl,xh

fl=func(x1)
fh=func(x2)
if (((fl > 0.0 .and. fh > 0.0) .or. &
    (fl < 0.0 .and. fh < 0.0)) call &
    nrerror('rtflsp: root must be bracketed between arguments'))
if (fl < 0.0) then Identify the limits so that xL corresponds to the low side.
xL=x1
xH=x2
else xL=x2
xH=x1
end if
dx=xH-xL
do j=1,MAXIT
    rtf lsp=xl+dx*fl/(fl-fh)
    f=func(rtf lsp)
    if (f < 0.0) then
        del=xl-rtf lsp
        xl=rtf lsp
        fl=f
    else
        del=xh-rtf lsp
        xh=rtf lsp
        fh=f
    end if
    dx=xh-xl
    if (abs(del) < xacc .or. f == 0.0) RETURN
    Convergence.
end do
END FUNCTION rtf lsp

FUNCTION rtsec(func,x1,x2,xacc)
USE nrtype, USE nrutil, ONLY : nrerror,swap
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtsec
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=30
Using the secant method, find the root of a function func thought to lie between x1 and x2. The root, returned as rtsec, is refined until its accuracy is ±xacc.
Parameter: MAXIT is the maximum allowed number of iterations.
INTEGER(I4B) :: j
REAL(SP) :: dx,f,f1,x1
f=func(x1)
f=func(x2)
if (abs(f1) < abs(f)) then
    rtsec=x1
    x1=x2
    call swap(f1,f)
else
    x1=x1
    rtsec=x2
end if
do j=1,MAXIT
    dx=(x1-rtsec)*f/(f-f1)
    x1=rtsec
    fl=func(rtsec)
    rtsec=rtsec+dx
    f=func(rtsec)
    if (abs(dx) < xacc .or. f == 0.0) RETURN
    Convergence.
end do
END FUNCTION rtsec

*     *     *
FUNCTION zriddr(func, x1, x2, xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1, x2, xacc
REAL(SP) :: zriddr
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=60

Using Ridders' method, return the root of a function func known to lie between x1 and x2. The root, returned as zriddr, will be refined to an approximate accuracy xacc.

REAL(SP), PARAMETER :: UNUSED=-1.11e30_sp
INTEGER(I4B) :: j
REAL(SP) :: fh, fl, fm, fnew, s, xh, xl, xm, xnew
fl=func(x1)
fh=func(x2)
if ((fl > 0.0 .and. fh < 0.0) .or. (fl < 0.0 .and. fh > 0.0)) then
   xl=x1
   xh=x2
   zriddr=UNUSED
   Any highly unlikely value, to simplify logic below.
do j=1,MAXIT
   xm=0.5_sp*(xl+xh)
   fm=func(xm)
   s=sqrt(fm**2-fl*fh)
   if (s == 0.0) RETURN
   xnew=xm+(xm-xl)*(sign(1.0_sp,fl-fh)*fm/s)
   Updating formula.
   if (abs(xnew-zriddr) <= xacc) RETURN
   zriddr=xnew
   fnew=func(zriddr)
   if (fnew == 0.0) RETURN
   if (sign(fm,fnew) /= fm) then
      Bookkeeping to keep the root bracketed on next iteration.
      xl=xm
      fl=fm
      xh=xriddr
      fh=fnew
   else if (sign(fl,fnew) /= fl) then
      xh=xm
      fnew=func(zriddr)
   else if (sign(fh,fnew) /= fh) then
      xl=xriddr
      f1=fnew
   else
      call nrerror('zriddr: never get here')
   end if
   if (abs(xh-xl) <= xacc) RETURN
   end do
   call nrerror('zriddr: exceeded maximum iterations')
else if (fl == 0.0) then
   zriddr=x1
else if (fh == 0.0) then
   zriddr=x2
else
   call nrerror('zriddr: root must be bracketed')
end if
END FUNCTION zriddr
FUNCTION zbrent(func,x1,x2,tol)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,tol
REAL(SP) :: zbrent
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: EPS=epsilon(x1)

Using Brent's method, find the root of a function func known to lie between x1 and x2.
The root, returned as zbrent, will be refined until its accuracy is tol.
Parameters: Maximum allowed number of iterations, and machine floating-point precision.

INTEGER(I4B) :: iter
REAL(SP) :: a,b,c,d,e,fa,fb,fc,p,q,r,s,tol1,xm
a=x1
b=x2
fa=func(a)
fb=func(b)
if ((fa > 0.0 .and. fb > 0.0) .or. (fa < 0.0 .and. fb < 0.0)) &
call nrerror('root must be bracketed for zbrent')
c=b
fc=fb
do iter=1,ITMAX
  if ((fb > 0.0 .and. fc > 0.0) .or. (fb < 0.0 .and. fc < 0.0)) then
    c=a
    Rename a, b, c and adjust bounding interval d.
    d=b-a
    e=d
  end if
  if (abs(fc) < abs(fb)) then
    a=b
    b=c
    c=a
    fa=fb
    fb=fc
    fc=fa
  end if
  tol1=2.0_sp*EPS*abs(b)+0.5_sp*tol
  Convergence check.
  xm=0.5_sp*(c-b)
  if (abs(xm) <= tol1 .or. fb == 0.0) then
    zbrent=b
    RETURN
  end if
  if (abs(e) >= tol1 .and. abs(fa) > abs(fb)) then
    s=fb/fa
    Attempt inverse quadratic interpolation.
    if (a == c) then
      p=2.0_sp*xm*s
      q=1.0_sp-s
    else
      q=fa(fc)
      r=fb(fc)
      p=s*(2.0_sp*xm&q*(q-r)-(b-a)*(r-1.0_sp))
      q=(q-1.0_sp)*(r-1.0_sp)*(s-1.0_sp)
    end if
    if (p > 0.0) q=-q
    Check whether in bounds.
    p=abs(p)
    if (2.0_sp*p < min(3.0_sp*xm&q-abs(tol1*q),abs(e*q))) then
      e=d
      Accept interpolation.


\[ d = \frac{p}{q} \]

else

\[ d = x_m \]

end if

else

Bounds decreasing too slowly; use bisection.

\[ d = x_m \]

\[ e = d \]

end if

Move last best guess to \( a \).

\[ a = b \]

\[ f_a = f(b) \]

Evaluate new trial root.

\[ f(b) = \text{func}(b) \]

end do

call nrerror('zbrent: exceeded maximum iterations')

zbrent = b

END FUNCTION zbrent

REAL(SP), PARAMETER :: EPS = epsilon(x1)  
The routine zbrent works best when EPS is exactly the machine precision. The Fortran 90 intrinsic function epsilon allows us to code this in a portable fashion.

FUNCTION rtnewt(funcd, x1, x2, xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1, x2, xacc
REAL(SP) :: rtnewt
INTERFACE
SUBROUTINE funcd(x, fval, fderiv)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: fval, fderiv
END SUBROUTINE funcd
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT = 20
Using the Newton-Raphson method, find the root of a function known to lie in the interval \([x_1, x_2]\). The root \( rtnewt \) will be refined until its accuracy is known within \( \pm xacc \).
\text{funcd} \ is a user-supplied subroutine that returns both the function value and the first derivative of the function.

Parameter: \( \text{MAXIT} \) is the maximum number of iterations.

INTEGER(I4B) :: j
REAL(SP) :: df, dx, f
rtnewt = 0.5_sp*(x1+x2)  
Initial guess.

do j=1,MAXIT
    call funcd(rtnewt, f, df)
    dx = f/df
    rtnewt = rtnewt - dx
    if ((x1 - rtnewt) * (rtnewt - x2) < 0.0) &
        call nrerror('rtnewt: values jumped out of brackets')
    if (abs(dx) < xacc) RETURN  
Convergence.
end do

call nrerror('rtnewt exceeded maximum iterations')
END FUNCTION rtnewt

* * *

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. Further reproduction is strictly prohibited (including for Internet uses). To order Numerical Recipes books or CD-ROMs, visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).
FUNCTION rtsafe(funcd,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtsafe
INTERFACE
  SUBROUTINE funcd(x,fval,fderiv)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: fval,fderiv
END SUBROUTINE funcd
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=100

Using a combination of Newton-Raphson and bisection, find the root of a function bracketed between \( x_1 \) and \( x_2 \). The root, returned as the function value \( rtsafe \), will be refined until its accuracy is known within \( \pm xacc \). \( funcd \) is a user-supplied subroutine that returns both the function value and the first derivative of the function.

Parameter: \( MAXIT \) is the maximum allowed number of iterations.

INTEGER(I4B) :: j
REAL(SP) :: df, dx, dxold, f, fh, fl, temp, xh, xl
call funcd(x1,fl,df)
call funcd(x2,fh,df)
if ((fl > 0.0 .and. fh > 0.0) .or. 
    (fl < 0.0 .and. fh < 0.0)) &
    call nrerror('root must be bracketed in rtsafe')
if (fl == 0.0) then
  rtsafe=x1
  RETURN
else if (fh == 0.0) then
  rtsafe=x2
  RETURN
else if (fl < 0.0) then
  xl=x1
  xh=x2
else
  xh=x1
  xl=x2
end if
rtsafe=0.5_sp*(xl+x2)  
    the “stepsize before last,”
dx=dxold
    and the last step.
call funcd(rtsafe,f,df)
do j=1,MAXIT
    Loop over allowed iterations.
    if (((rtsafe-xh)*df-f)*((rtsafe-xl)*df-f) > 0.0 .or. &
        abs(2.0_sp*f) > abs(dxold*df) ) then
        Bisect if Newton out of range, or not decreasing fast enough.
dxold=dx
    else
        Newton step acceptable. Take it.
dx=f/df
    temp=rtsafe
dxold=dx
dx=f/df
temp=rtsafe
    if (temp == rtsafe) RETURN
rtsafe=rtsafe-dx
    Change in root is negligible.
else
    Newton step acceptable. Take it.
dxold=dx
dx=f/df
temp=rtsafe
    if (temp == rtsafe) RETURN
end if
if (abs(dx) < xacc) RETURN
    Convergence criterion.
call funcd(rtsafe,f,df)
if (f < 0.0) then
    Maintain the bracket on the root.
xl=rtsafe
else
    xh=rtsafe
SUBROUTINE laguer(a,x,its)
USE nrtype; USE nrutil, ONLY : nrerror,poly,poly_term
IMPLICIT NONE
INTEGER(14B), INTENT(OUT) :: its
COMPLEX(SPC), INTENT(INOUT) :: x
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
REAL(SP), PARAMETER :: EPS=epsilon(1.0_sp)
INTEGER(14B), PARAMETER :: MR=8,MT=10,MXIT=MT*MR

Given an array of \( M + 1 \) complex coefficients \( a \) of the polynomial \( \sum_{i=1}^{M+1} a(i)x^{i-1} \), and
given a complex value \( x \), this routine improves \( x \) by Laguerre’s method until it converges,
within the achievable roundoff limit, to a root of the given polynomial. The number of
iterations taken is returned as \( \text{its} \).

Parameters:
- \( \text{EPS} \) is the estimated fractional roundoff error. We try to break (rare) limit
cycles with \( \text{MR} \) different fractional values, once every \( \text{MT} \) steps, for \( \text{MAXIT} \) total allowed
iterations.

INTEGER(14B) :: iter,m
REAL(SP) :: abx,abp,abm,err
COMPLEX(SPC) :: dx,x1,f,g,h,sq,gp,gm,g2
COMPLEX(SPC), DIMENSION(size(a)) :: b,d
REAL(SP), DIMENSION(MR) :: frac = &
(/ 0.5_sp,0.25_sp,0.75_sp,0.13_sp,0.38_sp,0.62_sp,0.88_sp,1.0_sp /)

Fractions used to break a limit cycle.
m=size(a)-1
do iter=1,MXIT
    its=iter
    abx=abs(x)
    b(m+1:m+1)=poly_term(a(m+1:m+1),x)
    d(m:m+1)=poly_term(b(m+1:m+1),x)
    f=poly(x,d(2:m)) f
    err=EPS*poly(abx,abs(b(1:m+1)))
    if (abs(b(1)) <= err) RETURN
    g=d(1)/b(1)
    g2=g*g
    h=g2-2.0_sp*f/b(1)
    sq=sqrt((m-1)*(m*h-g2))
    gp=g+sq
    gm=g-sq
    abp=abs(gp)
    abm=abs(gm)
    if (abp < abm) gp=gm
    else
        dx=m/gp
    end if
    x1=x-dx
    if (x == x1) RETURN
    Converged.
    if (mod(iter,MT) /= 0) then
        x=x-dx*frac(iter/MT)
    end if
end do

call nrerror('laguer: too many iterations')
The intrinsic function `cmplx` returns a quantity of type default complex unless the kind argument is present. To facilitate converting our routines from single to double precision, we always include the kind argument explicitly so that when you redefine `spc` in `nrtype` to be double-precision complex the conversions are carried out correctly.

The intrinsic function `cmplx` returns a quantity of type default complex unless the kind argument is present. To facilitate converting our routines from single to double precision, we always include the kind argument explicitly so that when you redefine `spc` in `nrtype` to be double-precision complex the conversions are carried out correctly.

```fortran
x = cmplx(0.0_sp, kind=spc)
```

See the discussion of why we include `kind=spc` just above. Note that while `real(x)` returns type default real if `x` is integer or real, it returns single or double precision correctly if `x` is complex.

```fortran
x = cmplx(0.0_sp, kind=spc)
```
SUBROUTINE zrhqr(a,rtr,rti)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : balanc,hqr,indexx
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: a
REAL(SP), DIMENSION(:,), INTENT(OUT) :: rtr,rti
Find all the roots of a polynomial with real coefficients, \( \sum_{i=1}^{M+1} a(i)x^{i-1} \), given the array of \( M + 1 \) coefficients \( a \). The method is to construct an upper Hessenberg matrix whose
eigenvalues are the desired roots, and then use the routines balanc and hqr. The real and
imaginary parts of the \( M \) roots are returned in \( rtr \) and \( rti \), respectively.
INTEGER(I4B) :: k,m
INTEGER(I4B), DIMENSION(size(rtr)) :: indx
REAL(SP), DIMENSION(size(a)-1,size(a)-1) :: hess
m=assert_eq(size(rtr),size(rti),size(a)-1,'zrhqr')
if (a(m+1) == 0.0) call &
    nrerror('zrhqr: Last value of array a must not be 0')
hess(1,:)=-a(m:1:-1)/a(m+1)        Construct the matrix.
hess(2:m,:)=0.0
do k=1,m-1
    hess(k+1,k)=1.0
end do
end do
call balanc(hess)                    Find its eigenvalues.
call hqr(hess,rtr,rti)               Sort roots by their real parts.
call indexx(rtr,indx)
rtr=rtr(indx)
rti=rti(indx)
END SUBROUTINE zrhqr

SUBROUTINE qroot(p,b,c,eps)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : poldiv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: p
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: eps
INTEGER(I4B), PARAMETER :: ITMAX=20
REAL(SP), PARAMETER :: TINY=1.0e-6_sp
Given an array of \( N \) coefficients \( p \) of a polynomial of degree \( N-1 \), and trial values for the
coefficients of a quadratic factor \( x^2 + bx + c \), improve the solution until the coefficients
\( b,c \) change by less than \( \epsilon \). The routine poldiv of §5.3 is used.
Parameters: ITMAX is the maximum number of iterations, TINY is a small number.
INTEGER(I4B) :: iter,n
REAL(SP) :: delb,delc,div,r,rb,rc,s,sc,sb
REAL(SP), DIMENSION(3) :: d
REAL(SP), DIMENSION(size(p)) :: q,qq,rem
n=size(p)
d(3)=1.0
do iter=1,ITMAX
    d(2)=b
d(1)=c
    call poldiv(p,d,q,rem)
    s=rem(1)                      First division gives \( r,s \).
    r=rem(2)
    call poldiv(q(1:n-1),d(:,),qq(1:n-1),rem(1:n-1))
    sc=rem(1)                     Second division gives partial \( r,s \) with respect
    rc=rem(2)                     to \( c \).
    sb=cr
    rc=bc-sc
    div=1.0_sp/(sb*rc-sc*rb)     Solve 2x2 equation.
\[ \text{delb} = (r*sc - s*rc) \cdot \text{div} \]
\[ \text{delc} = (-r*sb + s*rb) \cdot \text{div} \]
\[ b = b + \text{delb} \]
\[ c = c + \text{delc} \]
\[ \text{if } (\text{abs(} \text{delb} \text{)} \leq \text{eps} \cdot \text{abs(b)} \text{ or abs(b) < TINY}) \text{ and } (\text{abs(} \text{delc} \text{)} \leq \text{eps} \cdot \text{abs(c)} \text{ or abs(c) < TINY})) \text{ RETURN Coefficients converged.} \]
end do

end SUBROUTINE qroot
SUBROUTINE lnsrch(xold, fold, g, p, f, stpmax, check, func)
USE nrtype; USE nrutil, ONLY : assert_eq, nrerror, vabs
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: xold, g
REAL(SP), DIMENSION(:,), INTENT(IN) :: p
REAL(SP), INTENT(IN) :: fold, stpmax
REAL(SP), DIMENSION(:,), INTENT(OUT) :: x
REAL(SP), INTENT(OUT) :: f
LOGICAL(LGT), INTENT(OUT) :: check
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP) :: func
REAL(SP), DIMENSION(:), INTENT(IN) :: x
END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: ALF=1.0e-4_sp, TOLX=epsilon(x)

Given an \( N \)-dimensional point \( x_{\text{old}} \), the value of the function and gradient there, \( f_{\text{old}} \) and \( g \), and a direction \( p \), finds a new point \( x \) along the direction \( p \) from \( x_{\text{old}} \) where the function \( \text{func} \) has decreased "sufficiently." \( x_{\text{old}}, g, p \), and \( x \) are all arrays of length \( N \). The new function value is returned in \( f \). \( stpmax \) is an input quantity that limits the length of the steps so that you do not try to evaluate the function in regions where it is undefined or subject to overflow. \( p \) is usually the Newton direction. The output quantity \( \text{check} \) is false on a normal exit. It is true when \( x \) is too close to \( x_{\text{old}} \). In a minimization algorithm, this usually signals convergence and can be ignored. However, in a zero-finding algorithm the calling program should check whether the convergence is spurious.

Parameters: \( ALF \) ensures sufficient decrease in function value; \( TOLX \) is the convergence criterion on \( \Delta x \).

INTEGER(I4B) :: ndum
REAL(SP) :: a, alam, alamin, b, disc, f2, pabs, rhs1, rhs2, slope, tmplam

\( \text{ndum} = \text{assert_eq(size(g), size(p), size(x), size(xold), 'lnsrch')} \)
\( \text{check} = \text{false}. \)
\( \text{pabs} = \text{vabs(p(:,))} \)
\( \text{if (pabs > stpmax) p(:)=p(:)*stpmax/pabs} \) Scale if attempted step is too big.
\( \text{slope} = \text{dot_product(g, p)} \)
\( \text{if (slope >= 0.0) call nrerror('roundoff problem in lnsrch')} \)
\( \text{alam} = \text{TOLX/maxval(abs(p(:,))/max(abs(xold(:,)), 1.0_sp))} \) Compute \( \lambda_{\text{min}} \).
\( \text{alam} = 1.0 \) Always try full Newton step first.

\( \text{do} \)
\( \text{x(:)=xold(:)+alam*p(:)} \) Start of iteration loop.
\( \text{f=f(func(x))} \)
\( \text{if (alam < alamin) then} \)
\( \text{x(:)=xold(:)} \) Convergence on \( \Delta x \). For zero finding, the calling program should verify the convergence.
\( \text{check} = \text{true}. \) \( \text{RETURN} \)
\( \text{else if (f <= fold+ALF*alam*slope) then} \)
\( \text{Sufficient function decrease.} \) \( \text{RETURN} \)
\( \text{else if (alam == 1.0) then} \)
\( \text{First time.} \)
\( \text{tmplam} = \text{slope}/(2.0_sp*(f-fold-slope)) \)
\( \text{else} \) \( \text{Subsequent backtracks.} \)
\( \text{rhs1=f-fold-alam*slope} \)
\( \text{rhs2=f2-fold-alam2*slope} \)
\( \text{a=(rhs1/alam**2-rhs2/alam2**2)/(alam-alam2)} \)
\( \text{b=(-alam2*rhs1/alam**2+alam*rhs2/alam2**2)/(alam-alam2)} \)
\( \text{if (a == 0.0) then} \)
\( \text{tmplam} = \text{slope}/(2.0_sp*b) \)
\( \text{else} \)
\( \text{disc=b*b-3.0_sp*a*slope} \)
\( \text{if (disc < 0.0) then} \)
\( \text{tmplam} = \text{0.5_sp*alam} \)
\( \text{else if (b <= 0.0) then} \)
tmplam=(-b+sqrt(disc))/(3.0_sp*a)
else
  tmplam=-slope/(b+sqrt(disc))
end if
if (tmplam > 0.5_sp*alam) tmplam=0.5_sp*alam  λ ≤ 0.5λ1.
end if
alam2=alam
f2=f
alam=max(tmplam,0.1_sp*alam)  λ ≥ 0.1λ1.
end do
Try again.
END SUBROUTINE lnsrch

SUBROUTINE newt(x,check)
USE nrtype; USE nrutil, ONLY : nrerror,vabs
USE nr, ONLY : fdjac,lnsrch,lubksb,ludcmp
USE fminln
Communicates with fmin.
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
LOGICAL(LGT), INTENT(OUT) :: check
INTEGER(I4B), PARAMETER :: MAXITS=200
REAL(SP), PARAMETER :: TOLF=1.0e-4_sp,TOLMIN=1.0e-6_sp,TOLX=epsilon(x),&
STPMX=100.0
Given an initial guess x for a root in N dimensions, find the root by a globally convergent
Newton's method. The length N vector of functions to be zeroed, called fvec in the rou-
tine below, is returned by a user-supplied routine that must be called funcv and have the
declaration FUNCTION funcv(x). The output quantity check is false on a normal return
and true if the routine has converged to a local minimum of the function fmin defined
below. In this case try restarting from a different initial guess.
Parameters: MAXITS is the maximum number of iterations; TOLF sets the convergence
criterion on function values; TOLMIN sets the criterion for deciding whether spurious con-
vergence to a minimum of fmin has occurred; TOLX is the convergence criterion on δx;
STPMX is the scaled maximum step length allowed in line searches.
INTEGER(14B) :: its
INTEGER(14B), DIMENSION(size(x)) :: indx
REAL(SP) :: d,f,fold,stpmax
REAL(SP) :: g,p,xold
REAL(SP), DIMENSION(x) :: fvec
REAL(SP), DIMENSION(size(x),size(x)) :: fjac
fmin_fvecp=>fvec
f=fmin(x) fvec is also computed by this call.
if (maxval(abs(fvec(:))) < 0.01_sp*TOLF) then  Test for initial guess being a root.
  check=.false.  Use more stringent test than simply TOLF.
RETURN
end if
stpmax=STPMX*max(vabs(x(:)),real(size(x),sp))  Calculate stpmax for line searches.
do its=1,MAXITS  Start of iteration loop.
call fdjac(x,fvec,fjac)  If analytic Jacobian is available, you can replace the routine fdjac below with your own
routine.
g(:)=matmul(fvec(:),fjac(:,:))  Compute ∇f for the line search.
zold(:)=x(:)  Store x.
fold=f  and f.
p(:)=fvec(:)  Right-hand side for linear equations.
call ludcmp(fjac,indx,d)  Solve linear equations by LU decomposition.
call lubksb(fjac,indx,p)
call lnsrch(xold,fold,g,p,x,f,stpmax,check,fmin)
  lnsrch returns new x and f. It also calculates fvec at the new x when it calls fmin.
if (maxval(abs(fvec(:))) < TOLF) then  Test for convergence on function val-
  check=.false.  ues.
RETURN

if (check) then  
  Check for gradient of \( f \) zero, i.e., spurious convergence.
  check=(maxval(abs(g(:))*max(abs(x(:)),1.0_sp) / &
  max(f,0.5_sp*size(x))) < TOLMIN)
  RETURN
end if
if (maxval(abs(x(:)-xold(:))/max(abs(x(:)),1.0_sp)) < TOLX) &
  RETURN
end do

call nrerror('MAXITS exceeded in newt')
END SUBROUTINE newt

USE fmin

Here we have an example of how to pass an array \( fvec \) to a function \( fmin \) without making it an argument of \( fmin \). In the language of §21.5, we are using Method 2: We define a pointer \( fmin\_fvecp \) in the module \( fminln \):

```fortran
REAL(SP), DIMENSION(:,), POINTER :: fmin_fvecp
```

\( fvec \) itself is declared as an automatic array of the appropriate size in \( newt \):

```fortran
REAL(SP), DIMENSION(size(x)), TARGET :: fvec
```

On entry into \( newt \), the pointer is associated:

```fortran
fmin_fvecp=>fvec
```

The pointer is then used in \( fmin \) as a synonym for \( fvec \). If you are sufficiently paranoid, you can test whether \( fmin\_fvecp \) has in fact been associated on entry into \( fmin \). Heeding our admonition always to deallocate memory when it no longer is needed, you may ask where the deallocation takes place in this example. Answer: On exit from \( newt \), the automatic array \( fvec \) is automatically freed.

The Method 1 way of setting up this task is to declare an allocatable array in the module:

```fortran
REAL(SP), DIMENSION(:,), ALLOCATABLE :: fvec
```

On entry into \( newt \) we allocate it appropriately:

```fortran
allocate(fvec,size(x))
```

and it can now be used in both \( newt \) and \( fmin \). Of course, we must remember to deallocate explicitly \( fvec \) on exit from \( newt \). If we forget, all kinds of bad things would happen on a second call to \( newt \). The status of \( fvec \) on the first return from \( newt \) becomes undefined. The status cannot be tested with if(allocated(\( \ldots \))), and \( fvec \) may not be referenced in any way. If we tried to guard against this by adding the \texttt{SAVE} attribute to the declaration of \( fvec \), then we would generate an error from trying to allocate an already-allocated array.
FUNCTION funcv(x)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: funcv
END FUNCTION funcv
END INTERFACE
REAL(SP), PARAMETER :: EPS=1.0e-4_sp

Computes forward-difference approximation to Jacobian. On input, x is the point at which the Jacobian is to be evaluated, and fvec is the vector of function values at the point, both arrays of length N. df is the N x N output Jacobian. FUNCTION funcv(x) is a fixed-name, user-supplied routine that returns the vector of functions at x.

Parameter: EPS is the approximate square root of the machine precision.

integer(I4B) :: j,n
REAL(SP), DIMENSION(size(x)) :: xsav,xph,h
n=assert_eq(size(x),size(fvec),size(df,1),size(df,2),'fdjac')
xsav=x
h=EPS*abs(xsav)
where (h == 0.0) h=EPS
xph=xsav+h
Trick to reduce finite precision error.
h=xph-xsav

do j=1,n
  x(j)=xph(j)
  df(:,j)=(funcv(x)-fvec(:))/h(j)
  Forward difference formula.
  x(j)=xsav(j)
end do
END SUBROUTINE fdjac

MODULE fminln
USE nrtype; USE nrutil, ONLY : nrerror
REAL(SP), DIMENSION(:), POINTER :: fmin_fvecp
CONTAINS

FUNCTION fmin(x)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP) :: fmin

Returns \( f = \frac{1}{2} \mathbf{F} \cdot \mathbf{F} \) at x. FUNCTION funcv(x) is a fixed-name, user-supplied routine that returns the vector of functions at x. The pointer fmin_fvecp communicates the function values back to newt.

INTERFACE
FUNCTION funcv(x)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: funcv
END FUNCTION funcv
END INTERFACE

if (.not. associated(fmin_fvecp)) call & nrerror(’fmin: problem with pointer for returned values’)
fmin_fvecp=funcv(x)
fmin=0.5_sp*dot_product(fmin_fvecp,fmin_fvecp)
END FUNCTION fmin
END MODULE fminln

*   *   *
SUBROUTINE broydn(x, check)
USE nrtype; USE nrutil, ONLY : get_diag, lower_triangle, nrerror, 
outerprod, put_diag, unit_matrix, vabs
USE nr, ONLY : fdjac, lnsrch, qrdcmp, qrupd, rsolv
USE fmin
Communicates with fmin.
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
LOGICAL(LGT), INTENT(OUT) :: check
INTEGER(I4B), PARAMETER :: MAXITS = 200
REAL(SP), PARAMETER :: EPS = epsilon(x), TOLF = 1.0e-4_sp, TOLMIN = 1.0e-6_sp, 
TOLX = EPS, STPMX = 100.0

Given an initial guess \( x \) for a root in \( N \) dimensions, find the root by Broyden’s method 
embedded in a globally convergent strategy. The length \( N \) vector of functions to be ze-
roed, called \( fvec \) in the routine below, is returned by a user-supplied routine that 
must be called funcv and have the declaration FUNCTION funcv(x). The subroutine fdjac and the 
function fmin from next are used. The output quantity check is false on a normal 
return and true if the routine has converged to a local minimum of the function fmin or if 
Broyden’s method can make no further progress. In this case try restarting from a different 
initial guess.

Parameters: MAXITS is the maximum number of iterations; EPS is the machine precision; 
TOLF sets the convergence criterion on function values; TOLMIN sets the criterion for de-
ciding whether spurious convergence to a minimum of fmin has occurred; TOLX is the 
convergence criterion on \( \delta x \); STPMX is the scaled maximum step length allowed in line 
searches.

INTEGER(I4B) :: i, its, k, n
REAL(SP) :: f, fold, stpmax
REAL(SP), DIMENSION(size(x)), TARGET :: fvec, fvcold
REAL(SP), DIMENSION(size(x),size(x)) :: qt, r
REAL(SP), DIMENSION(size(x)) :: c, d, s, t, w, xold
LOGICAL :: restrt, sing

FUNCTION funcv(x)
  c, d, fvec, g, p, s, t, w, xold

fvec is also computed by this call.
if (maxval(abs(fvec(:))) < 0.01_sp*TOLF) then 
  check = .false.
  RETURN
end if

stpmax = STPMX * max(abs(x(:)), real(n, sp)) 
  Calculate stpmax for line searches.
  Ensure initial Jacobian gets computed.
do its = 1, MAXITS
  if (restrt) then 
    call fdjac(x, fvec, r)
    call qrdcmp(r, c, d, sing)
    if (sing) call nrerror('singular Jacobian in broydn')
    call unit_matrix(qt)
    do k = 1, n-1
      if (c(k) /= 0.0) then 
        qt(k:n,:) = qt(k:n,:) - outerprod(r(k:n,k),
        matmul(r(k:n,k), qt(k:n,:)))/c(k)
      end if
    end do
    where (lower_triangle(n,n)) r(:,i)=0.0
    call put_diag(d(:,r(:,i)))
  else 
    s(:)=x(:)-xold(:)
    do i = 1, n
      t(i) = dot_product(r(i,:), s(:))
    end do
    w(:)=fvec(:)-fvcold(:)-matmul(t(:), qt(:,:))
    where (abs(w(:)) < EPS * (abs(fvec(:))+abs(fvcold(:)))) 
    w(:) = 0.0
    do i = 1, n
      if (any(w(:) /= 0.0)) then 
        w(:) = matmul(qt(:,i), w(:))
        t(:) = matmul(qt(:,i), w(:))
        s(:) = s(:)-t(:)*dot_product(s, s)
        store s/(s · s) in s.
        Don't update with noisy components of w.
      end if
    end do
  end if
  Store s/(s · s) in s.
  Use more stringent test than simply TOLF.
  RETURN
call qrupdt(r,qt,t,s)  
Update $R$ and $Q^T$.

d(:)=get diag(r(:,:))  
Diagonal of $R$ stored in d.

if (any(d(:) == 0.0)) &
call nrerror('r singular in broydn')
end if

end if

p(:)=matmul(qt(:,:),fvec(:))  
r.h.s. for linear equations is $-Q^T \cdot F$.
do i=1,n

g(i)=dot_product(r(1:i,1),p(1:i))  
Compute $\nabla f \approx (Q \cdot R)^T \cdot F$ for the line search.

end do

xold(:)=x(:)  
Store $x$, $F$, and $f$.

fvcold(:)=fvec(:)

end do

xold(:)=x(:)  
Store $x$, $F$, and $f$.

fvcold(:)=fvec(:)

end do

call rsolv(r,d,p)  
Solve linear equations.
call lnsrch(xold,fold,g,p,x,f,stpmax,check,fmin)

lnsrch returns new $x$ and $f$. It also calculates $fvec$ at the new $x$ when it calls fmin.

if (maxval(abs(fvec(:))) < TOLF) then  
Test for convergence on function values.
check=.false.
RETURN
end if

if (check) then

if (restrt .or. maxval(abs(g(:))*max(abs(x(:)), &
1.0_sp)/max(f,0.5_sp*n)) < TOLMIN) RETURN

Try reinitializing the Jacobian.
The other test is for gradient of $f$ zero, i.e., spurious convergence.
restrt=.true.

else

restrt=.false.

if (maxval((abs(x(:)-xold(:)))/max(abs(x(:)), &
1.0_sp)) < TOLX) RETURN

Test for convergence on $\delta x$.

end if

end if

call nrerror('MAXITS exceeded in broydn')

END SUBROUTINE broydn

USE fmin  
See discussion for newt on p. 1197.

qt(x:n,:)=...outerprod...matmul  
Another example of the coding of equation (22.1.6).

where (lower_triangle(n,n))...  
The lower_triangle function in nrutil returns a lower triangular logical mask. As used here, the mask is true everywhere in the lower triangle of an $n \times n$ matrix, excluding the diagonal. An optional integer argument extra allows additional diagonals to be set to true. With extra=1 the lower triangle including the diagonal would be true.

call put_diag(d(:),r(:,:))  
This subroutine in nrutil sets the diagonal values of the matrix $r$ to the values of the vector $d$. It is overloaded so that $d$ could be a scalar, in which case the scalar value would be broadcast onto the diagonal of $r$. 

f90
Chapter B10. Minimization or Maximization of Functions

SUBROUTINE mnbrak(ax,bx,cx,fa,fb,fc,func)
USE nrtype; USE nrutil, ONLY : swap
IMPLICIT NONE
REAL(SP), INTENT(INOUT) :: ax,bx
REAL(SP), INTENT(OUT) :: cx,fa,fb,fc
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: GOLD=1.618034_sp,GLIMIT=100.0_sp,TINY=1.0e-20_sp
Given a function \( \text{func} \), and given distinct initial points \( ax \) and \( bx \), this routine searches in the downhill direction (defined by the function as evaluated at the initial points) and returns new points \( ax, bx, cx \) that bracket a minimum of the function. Also returned are the function values at the three points, \( fa, fb, fc \).

Parameters: \( \text{GOLD} \) is the default ratio by which successive intervals are magnified; \( \text{GLIMIT} \) is the maximum magnification allowed for a parabolic-fit step.

REAL(SP) :: fu,q,r,u,ulim
fa=func(ax)
fb=func(bx)
if (fb > fa) then
   call swap(ax,bx)
call swap(fa,fb)
end if
cx=bx+GOLD*(bx-ax)
fu=func(u)
end if
if (fb < fc) RETURN
Switch roles of \( a \) and \( b \) so that we can go downhill in the direction from \( a \) to \( b \).
First guess for \( c \).
Do-while-loop: Keep returning here until we bracket.
Compute \( u \) by parabolic extrapolation from \( a, b, c \). \( \text{TINY} \) is used to prevent any possible division by zero.
r=(bx-ax)*(fb-fc)
q=(bx-cx)*(fb-fa)
u=q*(bx-cx)/(2.0_sp*sign(max(abs(q-r),TINY),q-r))
ulim=bx+GLIMIT*(cx-bx)
We won’t go farther than this. Test various possibilities:
if ((bx-u)*(u-cx) > 0.0) then
   Parabolic \( u \) is between \( b \) and \( c \): try it.
   Got a minimum between \( b \) and \( c \).
   fu=func(u)
   if (fu < fc) then
      ax=bx
      fa=fb
      bx=u
      fb=fu
      RETURN
   else if (fu > fb) then
      Got a minimum between \( a \) and \( u \).
      cx=u
      fc=fu
      RETURN
end if
end if
end if
u=cx+GOLD*(cx-bx)
u=func(u)
else if (((cx-u)*(u-ulim) > 0.0) then
fu=func(u)
if (fu < fc) then
bx=cx
u=cx+GOLD*(cx-bx)
call shft(fb,fc,fu,func(u))
else
fu=func(u)
else if (((u-ulim)*(ulim-cx) >= 0.0) then
u=ulim
fu=func(u)
else
u=cx+GOLD*(cx-bx)
fu=func(u)
end if
call shft(ax,bx,cx,u)
call shft(fb,fc,fc,fu)
end do
CONTAINS
SUBROUTINE shft(a,b,c,d)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: d
a=b
b=c
c=d
END SUBROUTINE shft
END SUBROUTINE mnbrak

There are three places in mnbrak where we need to shift four variables around. Rather than repeat code, we make shft an internal subroutine, coming after a CONTAINS statement. It is invisible to all procedures except mnbrak.

FUNCTION golden(ax,bx,cx,func,tol,xmin)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: ax,bx,cx,tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: golden
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: R=0.61803399_sp,C=1.0_sp-R
Given a function func, and given a bracketing triplet of abscissas ax, bx, cx (such that bx is between ax and cx, and func(bx) is less than both func(ax) and func(cx)), this routine performs a golden section search for the minimum, isolating it to a fractional precision of about tol. The abscissa of the minimum is returned as xmin, and the minimum
function value is returned as golden, the returned function value.

Parameters: The golden ratios.

```fortran
REAL(SP) :: f1,f2,x0,x1,x2,x3
```

At any given time we will keep track of four points, \(x_0, x_1, x_2, x_3\).

Make \(x_0\) to \(x_1\) the smaller segment, and fill in the new point to be tried.

```fortran
x0=ax
x3=cx
if (abs(cx-bx) > abs(bx-ax)) then
    x1=bx
    x2=bx+C*(cx-bx)
else
    x2=bx
    x1=bx-C*(bx-ax)
end if
f1=func(x1)
f2=func(x2)
```

The initial function evaluations. Note that we never need to evaluate the function at the original endpoints.

Do-while-loop: We keep returning here.

```fortran
do
    if (abs(x3-x0) <= tol*(abs(x1)+abs(x2))) exit
    if (f2 < f1) then
        call shft3(x0,x1,x2,R*x2+C*x3)
    else
        call shft3(x3,x2,x1,R*x1+C*x0)
    end if
end do
```

Back to see if we are done.

```fortran
if (f1 < f2) then
    golden=f1
    xmin=x1
else
    golden=f2
    xmin=x2
end if
```

CONTAINS

```fortran
SUBROUTINE shft2(a,b,c)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b
REAL(SP), INTENT(IN) :: c
a=b
b=c
END SUBROUTINE shft2
```

```fortran
SUBROUTINE shft3(a,b,c,d)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: d
a=b
b=c
a=d
END SUBROUTINE shft3
```

END FUNCTION golden
FUNCTION brent(ax, bx, cx, func, tol, xmin)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: ax, bx, cx, tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: brent

INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE

INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: CGOLD=0.3819660_sp, ZEPS=1.0e-3_sp*epsilon(ax)

Given a function func, and given a bracketing triplet of abscissas ax, bx, cx (such that bx is between ax and cx, and func(bx) is less than both func(ax) and func(cx)), this routine isolates the minimum to a fractional precision of about tol using Brent's method.

The abscissa of the minimum is returned as xmin, and the minimum function value is returned as brent, the returned function value.

Parameters: Maximum allowed number of iterations; golden ratio; and a small number that protects against trying to achieve fractional accuracy for a minimum that happens to be exactly zero.

INTEGER(I4B) :: iter
REAL(SP) :: a, b, d, e, etemp, fu, fv, fx, p, q, r, tol1, tol2, u, v, w, x, x0

a = min(ax, cx) and b must be in ascending order, though the input abscissas need not be.

v = bx
w = v
x = v
e = 0.0
This will be the distance moved on the step before last.
f0 = func(x)
f1 = func(x)
f2 = func(x)

do iter=1, ITMAX
Main program loop.
x0 = 0.5_sp*(a+b)
tol1 = tol*abs(x0) + ZEPS
tol2 = 2.0_sp*tol1
if (abs(x-x0) <= (tol2 - 0.5_sp*(b-a))) then
Test for done here.
xmin = x
brent = f0
RETURN
end if
if (abs(e) > tol1) then
Construct a trial parabolic fit.
r = (x-w)*(f0-f1)
q = (x-v)*(f0-f2)
p = (x-v)*q - (x-w)*r
q = 2.0_sp*(q-r)
if (q > 0.0) p = -p
q = abs(q)
etemp = e
if (abs(p) >= abs(0.5_sp*q*etemp) .or. &
p <= q*(a-x) .or. p >= q*(b-x)) then
The above conditions determine the acceptability of the parabolic fit. Here it is not o.k., so we take the golden section step into the larger of the two segments.
e = merge(a-x, b-x, x >= x0 )
d = CGOLD*e
else
Take the parabolic step.
d = p/q
u = x+d
if (u-a < tol1 .or. b-u < tol1) d = sign(tol1, x0-x)
end if
else
  e=merge(a-x,b-x, x >= xm )
  d=CGOLD*e
end if
u=merge(x+d,x+sign(tol1,d), abs(d) >= tol1 )

Arrive here with d computed either from parabolic fit, or else from golden section.

fu=func(u)

This is the one function evaluation per iteration.
if (fu <= fx) then
  if (u >= x) then
    a=x
  else
    b=x
  end if
  call shft(v,w,x,u)
  call shft(fv,fw,fx,fu)
else
  if (u < x) then
    a=u
  else
    b=u
  end if
  if (fu <= fw .or. w == x) then
    v=w
    fv=fw
    w=u
    fw=fu
  else if (fu <= fv .or. v == x .or. v == w) then
    v=u
    fv=fu
  end if
end if
end do

Done with housekeeping. Back for another iteration.
call nrerror('brent: exceed maximum iterations')

CONTAINS

SUBROUTINE shft(a,b,c,d)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b,c,d
a=b
b=c
c=d
END SUBROUTINE shft

FUNCTION brent

FUNCTION dbrent(ax,bx,cx,func,dfunc,tol,xmin)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: ax,bx,cx,tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: dbrent
INTERFACE
  FUNCTION func(x)
  USE nrtype
  IMPLICIT NONE
  REAL(SP), INTENT(IN) :: x
  REAL(SP) :: func
  END FUNCTION func
  FUNCTION dfunc(x)

  else
    Take the golden section step into the larger of the two segments.
end if
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: dfunc
END FUNCTION dfunc
END INTERFACE

INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: ZEPS=1.0e-3_sp*epsilon(ax)

Given a function func and its derivative function dfunc, and given a bracketing triplet of abscissas ax, bx, cx [such that bx is between ax and cx, and func(bx) is less than both func(ax) and func(cx)], this routine isolates the minimum to a fractional precision of about tol using a modification of Brent’s method that uses derivatives. The abscissa of the minimum is returned as xmin, and the minimum function value is returned as dbrent, the returned function value.

Parameters: Maximum allowed number of iterations, and a small number that protects against trying to achieve fractional accuracy for a minimum that happens to be exactly zero.

INTEGER(I4B) :: iter
REAL(SP) :: a,b,d,d1,d2,du,dv,dw,dx,e,fu,fv,fw,fx,olde,tol1,tol2,&
     u,u1,u2,v,v1,v2,w,x,xm

Comments following will point out only differences from the routine brent. Read that routine first.

LOGICAL :: ok1,ok2
Will be used as flags for whether proposed steps are acceptable or not.

a=min(ax,cx)
b=max(ax,cx)
v=bx
w=v
x=v
e=0.0
fx=func(x)
fv=fx
fw=fx
dx=dfunc(x)
dv=dx
dw=dx
do iter=1,ITMAX
    xm=0.5_sp*(a+b)
tol1=tol*abs(x)+ZEPS
tol2=2.0_sp*tol1
    if (abs(x-xm) <= (tol2-0.5_sp*(b-a))) exit
    if (abs(e) > tol1) then
        d1=2.0_sp*(b-a)
        Initialize these d’s to an out-of-bracket value.
        d2=d1
        if (dw /= dx) d1=(w-x)*dx/(dx-dw)
        Secant method with each point.
        if (dv /= dx) d2=(v-x)*dx/(dx-dv)

        Which of these two estimates of d shall we take? We will insist that they be within
        the bracket, and on the side pointed to by the derivative at x:
        u1=x+d1
        u2=x+d2
        ok1=((a-u1)*(u1-b) > 0.0) .and. (dx*d1 <= 0.0)
        ok2=((a-u2)*(u2-b) > 0.0) .and. (dx*d2 <= 0.0)
        olde=e
        Movement on the step before last.
        e=d
        if (ok1 .or. ok2) then
            Take only an acceptable d, and if
            both are acceptable, then take
            d=merge(d1,d2, abs(d1) < abs(d2)) the smallest one.
            else
                d=merge(d1,d2,ok1)
            end if
        if (abs(d) <= abs(0.5_sp*olde)) then
            u=v+d
        if (u-a < tol2 .or. b-u < tol2) &
            d=sign(tol1,xm-x)
        else
```fortran
program dbrent
    implicit none
    real(kind=sp) :: xmin, dbrent, x, a, b, c
    real(kind=sp) :: dx, x, dx, tol1, tol2
    integer :: itmax, iter

    x = ...  
    tol1 = ...  
    tol2 = ...  
    itmax = ...  
    xmin = ...  
    dbrent = ...  

    call metis(x, a, b, c, dx, tol1, tol2)
    call move3(a, b, c, dx, tol1, tol2)
    call move3(c, a, b, dx, tol1, tol2)
    call move3(b, c, a, dx, tol1, tol2)

    xmin = x
    dbrent = func(x)

    call nrerror('dbrent: exceeded maximum iterations')

END PROGRAM dbrent
```
SUBROUTINE amoeba(p,y,ftol,func,iter)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,iminloc,nrerror,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: p
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B) :: ihi,ndim
REAL(SP), DIMENSION(size(p,2)) :: psum
CALL amoeba_private
CONTAINS
SUBROUTINE amoeba_private
IMPLICIT NONE
INTEGER(I4B) :: i,ilo,inhi
REAL(SP) :: rtol,ysave,ytry,ytmp
ndim=assert_eq(size(p,2),size(p,1)-1,size(y)-1,'amoeba')
iter=0
psum(:)=sum(p(:,:),dim=1)
DO
ilo=iminloc(y(:))
ihi=imaxloc(y(:))
ytmp=y(ihi)
y(ihi)=y(ilo)
inhi=imaxloc(y(:))
y(ihi)=ytmp
rtol=2.0_sp*abs(y(ihi)-y(ilo))/(abs(y(ihi))+abs(y(ilo))+TINY)
IF (rtol < ftol) THEN
CALL swap(y(1),y(ilo))
CALL swap(p(1,:),p(ilo,:))
RETURN
END IF
IF (iter >= ITMAX) CALL nrerror('ITMAX exceeded in amoeba')
BEGIN A NEW ITERATION. FIRST EXTRAPOLATE BY A FACTOR −1 THROUGH THE FACE OF THE SIMPLEX ACROSS FROM THE HIGH POINT, I.E., REFLECT THE SIMPLEX FROM THE HIGH POINT.
YTRY=AMOTRY(-1.0_sp)
ITER=ITER+1
IF (YTRY <= Y(ILON)) THEN
YTRY=AMOTRY(2.0_sp)
ITER=ITER+1
ELSE IF (YTRY >= Y(INHI)) THEN
YSAVE=Y(IHI)
YTRY=AMOTRY(0.5_sp)
ITER=ITER+1
END IF
END DO
END SUBROUTINE amoeba_private

Minimization of the function func in N dimensions by the downhill simplex method of Nelder and Mead. The \((N+1) \times N\) matrix p is input. Its \(N+1\) rows are \(N\)-dimensional vectors that are the vertices of the starting simplex. Also input is the vector y of length \(N+1\), whose components must be preinitialized to the values of func evaluated at the \(N+1\) vertices (rows) of p; and ftol the fractional convergence tolerance to be achieved in the function value (n.b.!). On output, p and y will have been reset to \(N+1\) new points all within ftol of a minimum function value, and iter gives the number of function evaluations taken.

Parameters: The maximum allowed number of function evaluations, and a small number.
if (ytry >= ysave) then
  Can't seem to get rid of that high point. Better contract around the lowest
  (best) point.
p(:, :) = 0.5_sp * p(:, :) + spread(p(ilo, :), 1, size(p, 1))
do i = 1, ndim + 1
    if (i /= ilo) y(i) = func(p(i, :))
  end do
  iter = iter + ndim
  Keep track of function evaluations.
psum(:) = sum(p(:, :), dim=1)
end if
end do
End subroutine amoeba_private

Function amotry(fac)
Implicit None
Real(s p), Intent(in) :: fac
Real(s p) :: amotry
  Extrapolates by a factor fac through the face of the simplex across from the high point,
  tries it, and replaces the high point if the new point is better.
Real(s p), dimension(size(p, 2)) :: ptry
fac1 = (1.0_sp - fac) / ndim
fac2 = fac1 - fac
ptry(:) = psum(:) * fac1 - p(ihi, :) * fac2
ytry = func(ptry)
  Evaluate the function at the trial point.
if (ytry < y(ihi)) then
  If it's better than the highest, then replace
  the highest.
y(ihi) = ytry
  psum(:) = psum(:) - p(ihi, :) + ptry(:)
p(ihi, :) = ptry(:)
end if
amotry = ytry
End function amotry
End subroutine amoeba

The only action taken by the subroutine amoeba is to call the internal
subroutine amoeba_private. Why this structure? The reason has to do
with meeting the twin goals of data hiding (especially for “safe” scope
of variables) and program readability. The situation is this: Logically,
amoeba does most of the calculating, but calls an internal subroutine
amotry at several different points, with several values of the parameter fac. However, fac is not the only piece
of data that must be shared with amotry; the latter also needs access to several
shared variables (ihi, ndim, psum) and arguments of amoeba (p, y, func).

The obvious (but not best) way of coding this would be to put the computational
guts in amoeba, with amotry as the sole internal subprogram. Assuming that fac
is passed as an argument to amotry (it being the parameter that is being rapidly
altered), one must decide whether to pass all the other quantities to amotry (i) as
additional arguments (as is done in the Fortran 77 version), or (ii) “automatically,”
i.e., doing nothing except using the fact that an internal subprogram has automatic
access to all of its host’s entities. Each of these choices has strong disadvantages.
Choice (i) is inefficient (all those arguments) and also obscures the fact that fac is
the primary changing argument. Choice (ii) makes the program extremely difficult to
read, because it wouldn’t be obvious without careful cross-comparison of the routines
which variables in amoeba are actually global variables that are used by amotry.

Choice (ii) is also “unsafe scoping” because it gives a nontrivially complicated
internal subprogram, amotry, access to all the variables in its host. A common
and difficult-to-find bug is the accidental alteration of a variable that one “thought”
was local, but is actually shared. (Simple variables like i, j, and n are the most common culprits.)

We are therefore led to reject both choice (i) and choice (ii) in favor of a structure previously described in the subsection on Scope, Visibility, and Data Hiding in §21.5. The guts of amoeba are put in amoeba_private, a sister routine to amotry. These two siblings have mutually private name spaces. However, any variables that they need to share (including the top-level arguments of amoeba) are declared as variables in the enclosing amoeba routine. The presence of these “global variables” serves as a warning flag to the reader that data are shared between routines.

An alternative attractive way of coding the above situation would be to use a module containing amoeba and amotry. Everything would be declared private except the name amoeba. The global variables would be at the top level, and the arguments of amoeba that need to be passed to amotry would be handled by pointers among the global variables. Unfortunately, Fortran 90 does not support pointers to functions. Sigh!

ilo=iminloc...ihi=imaxloc... See discussion of these functions on p. 1017.

call swap(y(1)...call swap(p(1,:)... Here the swap routine in nrutil is called once with a scalar argument and once with a vector argument. Inside nrutil scalar and vector versions have been overloaded onto the single name swap, hiding all the implementation details from the calling routine.

⋆⋆⋆

SUBROUTINE powell(p,xi,ftol,iter,fret)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : linmin
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: xi
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), INTENT(OUT) :: fret
INTERFACE
  FUNCTION func(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP) :: func
  END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=200
REAL(SP), PARAMETER :: TINY=1.0e-25_sp
Minimization of a function func of N variables. (func is not an argument, it is a fixed function name.) Input consists of an initial starting point p, a vector of length N; an initial N \times N matrix xi whose columns contain the initial set of directions (usually the N unit vectors); and ftol, the fractional tolerance in the function value such that failure to decrease by more than this amount on one iteration signals doneness. On output, p is set to the best point found, xi is the then-current direction set, fret is the returned function value at p, and iter is the number of iterations taken. The routine linmin is used.

Parameters: Maximum allowed iterations, and a small number.

INTEGER(I4B) :: i,ibig,n
REAL(SP) :: del,fp,ftt,t
REAL(SP), DIMENSION(size(p)) :: pt,ptt,xit
  n=assert_eq(size(p),size(xi,1),size(xi,2),'powell')
fret=func(p)
pt(:,)=p(:,)
iter=0
do
    iter=iter+1
    fp=fret
    ibig=0
    del=0.0
    do i=1,n
        xit(:,)=xi(:,i)
        fptt=fret
        call linmin(p,xit,fret)
        if (fptt-fret > del) then
            del=fptt-fret
            ibig=i
        end if
    end do
    if (2.0_sp*(fp-fret) <= ftol*(abs(fp)+abs(fret))+TINY) RETURN
    if (iter == ITMAX) call &
      nrerror('powell exceeding maximum iterations')
    ptt(:,)=2.0_sp*p(:,)-pt(:,)
    xit(:,)=p(:,)-pt(:,)
    pt(:,)=p(:,)
    fptt=func(ptt)
    if (fptt >= fp) cycle
    if (t >= 0.0) cycle
    call linmin(p,xit,fret)
    xi(:,ibig)=xi(:,n)
    xi(:,n)=xit(:,)
end do
END SUBROUTINE powell

---

MODULE f1dim_mod
USE nrtype
INTEGER(I4B) :: ncom
REAL(SP), DIMENSION(:), POINTER :: pcom,xicom
CONTAINS
FUNCTION f1dim(x)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: f1dim
    Used by linmin as the one-dimensional function passed to mnbrak and brent.
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:,), INTENT(IN) :: x
    REAL(SP) :: func
END FUNCTION func
END INTERFACE
REAL(SP), DIMENSION(:,), ALLOCATABLE :: xt
allocate(xt(ncom))
x(:,)=pcom(:,)+x*xicom(:)
f1dim=func(xt)
deallocate(xt)
END FUNCTION f1dim
END MODULE f1dim_mod
SUBROUTINE linmin(p,xi,fret)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : mnbrak,brent
USE f1dim_mod
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), TARGET, INTENT(INOUT) :: p,xi
REAL(SP), PARAMETER :: TOL=1.0e-4_sp

Given an $N$-dimensional point $p$ and an $N$-dimensional direction $\xi$, both vectors of length $N$, moves and resets $p$ to where the fixed-name function $\text{func}$ takes on a minimum along the direction $\xi$ from $p$, and replaces $\xi$ by the actual vector displacement that $p$ was moved. Also returns as $fret$ the value of $\text{func}$ at the returned location $p$. This is actually all accomplished by calling the routines $\text{mnbrak}$ and $\text{brent}$.

Parameter: Tolerance passed to $\text{brent}$.

REAL(SP) :: ax,bx,fa,fb,fx,xmin,xx
ncom=assert_eq(size(p),size(xi),'linmin')
pcom=>p
Communicate the global variables to $\text{f1dim}$.
xicom=>xi
ax=0.0
Initial guess for brackets.
xx=1.0
call $\text{mnbrak}(ax,xx,bx,fa,fb,fx,f1dim)$
$fret=\text{brent}(ax,xx,bx,f1dim,TOL,xmin)$
$\xi=xmin*\xi$
Construct the vector results to return.
$p=p+\xi$
END SUBROUTINE linmin

USE f1dim_mod At first sight this situation is like the one involving $\text{USE fminln}$ in $\text{newt}$ on p. 1197: We want to pass arrays $p$ and $\xi$ from $\text{linmin}$ to $\text{f1dim}$ without having them be arguments of $\text{f1dim}$. If you recall the discussion in §21.5 and on p. 1197, there are two ways of effecting this: via pointers or via allocatable arrays. There is an important difference here, however. The arrays $p$ and $\xi$ are themselves arguments of $\text{linmin}$, and so cannot be allocatable arrays in the module. If we did want to use allocatable arrays in the module, we would have to copy $p$ and $\xi$ into them. The pointer implementation is much more elegant, since no unnecessary copying is required. The construction here is identical to the one in $\text{fminln}$ and $\text{newt}$, except that $p$ and $\xi$ are arguments instead of automatic arrays.

* * *

MODULE df1dim_mod Used for communication from $\text{dlinmin}$ to $\text{f1dim}$ and $\text{df1dim}$.
USE nrtype
INTEGER(I4B) :: ncom
REAL(SP), DIMENSION(:), POINTER :: pcom,xicom
CONTAINS
FUNCTION f1dim(x)
IMPLICIT NONE
REAL(SP), DIMENSION(:), POINTER :: xcom
$f1dim=xcom$
END FUNCTION f1dim

Used by $\text{dlinmin}$ as the one-dimensional function passed to $\text{mnbrak}$.

INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
REAL(SP), DIMENSION(:), ALLOCATABLE :: xt
allocate(xt(ncom))
x(:)=pcom(:)+x*xicom(:)
f1dim=func(xt)
deallocate(xt)
END FUNCTION f1dim

FUNCTION df1dim(x)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: df1dim

Used by dlinmin as the one-dimensional function passed to dbrent.

INTERFACE
FUNCTION dfunc(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: dfunc
END FUNCTION dfunc
END INTERFACE

REAL(SP), DIMENSION(:), ALLOCATABLE :: xt,df
allocate(xt(ncom),df(ncom))
x(:)=pcom(:)+x*xicom(:)
df(:)=dfunc(xt)
df1dim=dot_product(df,xicom)
deallocate(xt,df)
END FUNCTION df1dim
END MODULE df1dim_mod

SUBROUTINE dlinmin(p,xi,fret)
USE nrtype; USE nrutil, ONLY: assert_eq
USE nr, ONLY: mnbrak,dbrent
USE df1dim_mod
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), TARGET :: p,xi
REAL(SP), PARAMETER :: TOL=1.0e-4_sp

Given an N-dimensional point p and an N-dimensional direction xi, both vectors of length N, moves and resets p to where the fixed-name function func takes on a minimum along the direction xi from p, and replaces xi by the actual vector displacement that p was moved. Also returns as fret the value of func at the returned location p. This is actually all accomplished by calling the routines mnbrak and dbrent. dfunc is a fixed-name user-supplied function that computes the gradient of func.

Parameter: Tolerance passed to dbrent.

REAL(SP) :: ax,bx,fa,fb,fx,xmin,xx
ncom=assert_eq(size(p),size(xi),'dlinmin')
pcom=>p
xicom=>xi
ax=0.0 Initial guess for brackets.
xx=1.0
call mnbrak(ax,xx,fa,fx,fb,f1dim)
fret=dbrent(ax,xx,fx,df1dim,df1dim,TOL,xmin)
xi=xmin*xi
Construct the vector results to return.
p=p+xi
END SUBROUTINE dlinmin

f90 USE df1dim_mod See discussion of USE f1dim_mod on p. 1212.
SUBROUTINE frprmn(p, ftol, iter, fret)
USE nrtype; USE arutil, ONLY : nrerror
USE nr, ONLY : linmin
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
INTERFACE
FUNCTION func(p)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: p
REAL(SP) :: func
END FUNCTION func
FUNCTION dfunc(p)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: p
REAL(SP), DIMENSION(size(p)) :: dfunc
END FUNCTION dfunc
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=200
REAL(SP), PARAMETER :: EPS=1.0e-10_sp
Given a starting point \( p \) that is a vector of length \( N \), Fletcher-Reeves-Polak-Ribiere minimization is performed on a function \( \text{func} \), using its gradient as calculated by a routine \( \text{dfunc} \). The convergence tolerance on the function value is input as \( \text{ftol} \). Returned quantities are \( p \) (the location of the minimum), \( \text{iter} \) (the number of iterations that were performed), and \( \text{fret} \) (the minimum value of the function). The routine \( \text{linmin} \) is called to perform line minimizations.
Parameters: \( \text{ITMAX} \) is the maximum allowed number of iterations; \( \text{EPS} \) is a small number to rectify the special case of converging to exactly zero function value.
INTEGER(I4B) :: its
REAL(SP) :: dgg, fp, gam, gg
REAL(SP), DIMENSION(size(p)) :: g, h, xi
fp=func(p)                              ! Initializations.
xi=dfunc(p)
g=-xi
h=g
xi=h
do its=1,ITMAX
   iter=its
   call linmin(p,xi,fret)            ! Next statement is the normal return:
   if (2.0_sp*abs(fret-fp) <= ftol*(abs(fret)+abs(fp)+EPS)) RETURN
   fp=fret
   xi=dfunc(p)
g=g+gam*h
   xi=h
   end do
   call nrerror('frprmn: maximum iterations exceeded')
END SUBROUTINE frprmn
SUBROUTINE dfpmin(p, gtol, iter, fret, func, dfunc)
USE nrtype; USE nrutil, ONLY : nrerror, outerprod, unit_matrix, vabs
USE nr, ONLY : lnsrch
IMPLICIT NONE
INTEGER(4), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: gtol
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: p
INTERFACE
  FUNCTION func(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:,), INTENT(IN) :: p
    REAL(SP) :: func
  END FUNCTION func
  FUNCTION dfunc(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:,), INTENT(IN) :: p
    REAL(SP), DIMENSION(size(p)) :: dfunc
  END FUNCTION dfunc
END INTERFACE
INTEGER(4), PARAMETER :: ITMAX=200
REAL(SP), PARAMETER :: STPMX=100.0_sp, EPS=epsilon(p), TOLX=4.0_sp*EPS

Given a starting point p that is a vector of length N, the Broyden-Fletcher-Goldfarb-Shanno variant of Davidon-Fletcher-Powell minimization is performed on a function func, using its gradient as calculated by a routine dfunc. The convergence requirement on zeroing the gradient is input as gtol. Returned quantities are p (the location of the minimum), iter (the number of iterations that were performed), and fret (the minimum value of the function). The routine lnsrch is called to perform approximate line minimizations.

Parameters: ITMAX is the maximum allowed number of iterations; STPMX is the scaled maximum step length allowed in line searches; EPS is the machine precision; TOLX is the convergence criterion on \( \Delta x \) values.

INTEGER(4) :: its
LOGICAL :: check
REAL(SP) :: den, fac, fad, fae, fp, stpmax, sumdg, sumxi
REAL(SP), DIMENSION(size(p)) :: dg, g, hdg, pnew, xi
REAL(SP), DIMENSION(size(p),size(p)) :: hessin
fp=func(p)  Calculate starting function value and gradient.
g=dfunc(p)  call unit_matrix(hessin)  Initialize inverse Hessian to the unit matrix.
xi=-g  Initial line direction.
stpmax=STPMX*max(vabs(p),real(size(p),sp))  do its=1,ITMAX  Main loop over the iterations.
  iter=its  call lnsrch(p,fp,g,xi,pnew,fret,stpmax,check,func)  The new function evaluation occurs in lnsrch; save the function value in fp for the next line search. It is usually safe to ignore the value of check.
  fp=fret  Update the line direction, and the current point.
  xi=pnew
  if (maxval(abs(xi)/max(abs(p),1.0_sp)) < TOLX) RETURN  Test for convergence on \( \Delta x \).
  dg=g  Save the old gradient.
  g=dfunc(p)  and get the new gradient.
  den=max(fret,1.0_sp)  if (maxval(abs(g)*max(abs(p),1.0_sp)/den) < gtol) RETURN  Test for convergence on zero gradient.
  dg=g-dg  Compute difference of gradients, and differences times current matrix.
  fac=dot_product(dg,xi)  fae=dot_product(dg,hdg)  Calculate dot products for the denominators.
  sumdg=dot_product(dg,dg)
call unit_matrix(hessin) 

The unit_matrix routine in nrutil does exactly what its name suggests. The routine dfpmin makes use of dot_product from nrutil, as well as the matrix intrinsics matmul and outerprod, to simplify and parallelize the coding.

SUBROUTINE dfpmin

f
90

The BFGS updating formula.

end if

fac=1.0_sp/fac

Vector that makes BFGS different from DFP.

dg=fac*xi-fad*hdg

hessin=hessin+fac*outerprod(xi,xi)-

fad=1.0_sp/fae

f=outerprod(hdg,hdg)+fae*outerprod(dg,dg)

hessin+hessin+fac*outerprod(xi,xi)-&

fad*outerprod(hdg,hdg)+fae*outerprod(dg,dg)

end if

xi=-matmul(hessin,g)

Now calculate the next direction to go,

end do

and go back for another iteration.

call nrerror('dfpmin: too many iterations')

END SUBROUTINE dfpmin

⋆⋆⋆
if (nl1 > 0) then
    kp=ll(maxloc(abs(a(1+1:n+1)),1:nl1+1)))
else
    bmax=0.0
end if

phasea: do
    if (bmax <= EPS .and. a(m+2,1) < -EPS) then
        Auxiliary objective function is still negative and can't be improved, hence no feasible solution exists.
        icase=-1
        RETURN
    else if (bmax <= EPS .and. a(m+2,1) <= EPS) then
        Auxiliary objective function is zero and can't be improved. This signals that we have a feasible starting vector. Clean out the artificial variables corresponding to any remaining equality constraints and then eventually exit phase one.
        do ip=m1+m2+1,m
            if (iposv(ip) == ip+n) then
                Found an artificial variable for an equality constraint.
                if (nl1 > 0) then
                    kp=l1(maxloc(abs(a(ip+1,l1(1:nl1)+1))))
                    bmax=a(ip+1,kp+1)
                else
                    bmax=0.0
                end if
                if (bmax > EPS) exit phasea
                Exchange with column corresponding to maximum pivot element in row.
            end if
        end do
        where (spread(l3(1:m2),2,n+1) == 1) &
        a(m1+2:m1+m2+1,1:n+1)=-a(m1+2:m1+m2+1,1:n+1)
        Change sign of row for any m2 constraints still present from the initial basis.
        exit phase1
    end if
    call simp1
    if (ip == 0) then
        Maximum of auxiliary objective function is unbounded, so no feasible solution exists.
        icase=-1
        RETURN
    end if
    exit phasea
end do phasea

call simp2(m+1,n)
Exchange a left- and a right-hand variable.
if (iposv(ip) >= n+m1+m2+1) then
    Exchanged out an artificial variable for an equality constraint. Make sure it stays out by removing it from the l1 list.
    k=iposv(ip)-m1-n
    if (kh >= 1) then
        Exchanged out an m2 type constraint.
        if (l3(kh) /= 0) then
            If it's the first time, correct the pivot column for the minus sign and the implicit artificial variable.
            a(m+2,kp+1)=a(m+2,kp+1)+1.0_sp
            a(1:m+2,kp+1)=-a(1:m+2,kp+1)
        end if
    end if
    end if
    call swap(izrov(kp),iposv(ip))
    Update lists of left- and right-hand variables.
end do phase1

phase2: do
    We have an initial feasible solution. Now optimize it.
    if (nl1 > 0) then
        kp=ll(maxloc(abs(a(1,1:n+1)),1:nl1+1)))
        Test the z-row for doneness.
        bmax=a(1,kp+1)
    else
        bmax=0.0
    end if
if (bmax <= EPS) then
  icase=0
  RETURN
end if

call simp1
if (ip == 0) then
  Objective function is unbounded. Report and return.
  icase=1
  RETURN
end if

call simp2(m,n)
Exchange a left- and a right-hand variable, update lists of left- and right-hand variables, and return for another iteration.

CONTAINS

SUBROUTINE simp1
Locate a pivot element, taking degeneracy into account.

IMPLICIT NONE
INTEGER(I4B) :: i,k
REAL(SP) :: q,q0,q1,qp

ip=0
i=ifirstloc(a(2:m+1,kp+1) < -EPS)
if (i > m) RETURN
No possible pivots. Return with message.
q1=-a(i+1,1)/a(i+1,kp+1)

do i=ip+1,m
  if (a(i+1,kp+1) < -EPS) then
    q=-a(i+1,1)/a(i+1,kp+1)
    if (q < q1) then
      ip=i
      q1=q
    else if (q == q1) then
      We have a degeneracy.
      do k=1,n
        qp=-a(ip+1,k+1)/a(ip+1,kp+1)
        q0=-a(i+1,k+1)/a(i+1,kp+1)
        if (q0 /= qp) exit
      end do
      if (q0 < qp) ip=i
    end if
  end if
end do

END SUBROUTINE simp1

SUBROUTINE simp2(i1,k1)
Matrix operations to exchange a left-hand and right-hand variable (see text).

IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: i1,k1
REAL(SP) :: piv
INTEGER(I4B), DIMENSION(k1) :: icol
INTEGER(I4B), DIMENSION(i1) :: irow
INTEGER(I4B), DIMENSION(max(i1,k1)+1) :: itmp

ip1=ip+1
kp1=kp+1
piv=1.0_sp/a(ip1,kp1)
itmp(1:k1+1)=arth(1,1,k1+1)
icol=pack(itmp(1:k1+1),itmp(1:k1+1) /= kp1)
itmp(1:i1+1)=arth(1,1,i1+1)
irow=pack(itmp(1:i1+1),itmp(1:i1+1) /= ip1)
a(irow,kp1)=a(irow,kp1)*piv
a(irow,icol)=a(irow,icol)-outerprod(a(irow,kp1),a(ip1,icol))
a(ip1,kp1)=piv

END SUBROUTINE simp2

END SUBROUTINE simpLx
The routine \texttt{simplx} makes extensive use of named \texttt{do}-loops to control the program flow. The various \texttt{exit} statements have the names of the \texttt{do}-loops attached to them so we can easily tell where control is being transferred to. We believe that it is almost never necessary to use \texttt{goto} statements: Code will always be clearer with well-constructed block structures.

This is not a real \texttt{do}-loop: It is executed only once, as you can see from the unconditional \texttt{exit} before the \texttt{end do}. We use this construction to define a block of code that is traversed only once but that has several possible exit points.

These lines are equivalent to

\begin{verbatim}
main_procedure: do
The routine \texttt{simplx} makes extensive use of named 
do-loops to control the program flow. The various \texttt{exit} statements have the names of the \texttt{do}-loops attached to them so we can easily tell where control is being transferred to. We believe that it is almost never necessary to use \texttt{goto} statements: Code will always be clearer with well-constructed block structures.

phase1a: do...end do phase1a  
This is not a real do-loop: It is executed only once, as you can see from the unconditional exit before the end do. We use this construction to define a block of code that is traversed only once but that has several possible exit points.

where (spread(l3(1:m12-m1),2,n+1) == 1) &
a(m1+2:m12+1,1:n+1)=-a(m1+2:m12+1,1:n+1)
\end{verbatim}

** SUBROUTINE anneal(x,y,iorder) **

\begin{verbatim}
SUBROUTINE anneal(x,y,iorder)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,swap
USE nr, ONLY : ran1
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
This algorithm finds the shortest round-trip path to \( N \) cities whose coordinates are in the length \( N \) arrays \( x, y \). The length \( N \) array \texttt{iorder} specifies the order in which the cities are visited. On input, the elements of \texttt{iorder} may be set to any permutation of the numbers 1...\( N \). This routine will return the best alternative path it can find.

INTEGER(I4B), DIMENSION(6) :: n
INTEGER(I4B) :: i1,i2,j,k,nlimit,ncity,nn,nover,nsucc
REAL(SP) :: de,harvest,path,t,tfactr
LOGICAL(LGT) :: ans
ncity=assert_eq(size(x),size(y),size(iorder),'anneal')
nover=100*ncity
nlimit=10*ncity
tfactr=0.9_sp
t=0.5_sp
path=sum(alen_v(x(iorder(1:ncity-1)),x(iorder(2:ncity)),
y(iorder(1:ncity-1)),y(iorder(2:ncity))))
\end{verbatim}
call ran1(harvest)

Decide whether to do a reversal or transport.
if (harvest < 0.5) then
  call ran1(harvest)
  n(3)=n(2)+int(abs(nn-2)*harvest)+1
  path=path+de
  call trncst(x,y,iorder,n,de)
  call metrop(de,t,ans)
  if (ans) then
    nauc=nauc+1
    path=path+de
    call transpt(iorder,n)
  end if
else
  call revcst(x,y,iorder,n,de)
  call metrop(de,t,ans)
  if (ans) then
    nauc=nauc+1
    path=path+de
    call revers(iorder,n)
  end if
end if
if (nauc >= nlimit) exit
Finish early if we have enough successful changes.
do
  call ran1(harvest)
  ans=(de < 0.0) .or. (harvest < exp(-de/t))
end do
CONTAINS
FUNCTION alen(x1,x2,y1,y2)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,y1,y2
REAL(SP) :: alen
Computes distance between two cities.
alen=sqrt((x2-x1)**2+(y2-y1)**2)
END FUNCTION alen
FUNCTION alen_v(x1,x2,y1,y2)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1,x2,y1,y2
REAL(SP), DIMENSION(size(x1)) :: alen_v
Computes distances between pairs of cities.
alen_v=sqrt((x2-x1)**2+(y2-y1)**2)
END FUNCTION alen_v
SUBROUTINE metrop(de,t,ans)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: de,t
LOGICAL(LGT), INTENT(OUT) :: ans
Metropolis algorithm. ans is a logical variable that issues a verdict on whether to accept a reconfiguration that leads to a change de in the objective function. If de<0, ans=.true., while if de>0, ans is only .true. with probability exp(-de/t), where t is a temperature determined by the annealing schedule.
call ran1(harvest)
ans=(de < 0.0) .or. (harvest < exp(-de/t))
END SUBROUTINE metrop
SUBROUTINE revcst(x,y,iorder,n,de)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: n
REAL(SP), INTENT(OUT) :: de
This subroutine returns the value of the cost function for a proposed path reversal. The arrays \( x \) and \( y \) give the coordinates of these cities. \( iorder \) holds the present itinerary. The first two values \( n(1) \) and \( n(2) \) of array \( n \) give the starting and ending cities along the path segment which is to be reversed. On output, \( de \) is the cost of making the reversal. The actual reversal is not performed by this routine.

```fortran
INTEGER(I4B) :: ncity
REAL(SP), DIMENSION(4) :: xx,yy
ncity=size(x)
ncity=size(iorder)
n(3)=1+mod((n(1)+ncity-2),ncity)  \( \text{Find the city before } n(1) \ldots \)
ncity=size(iorder)
n(4)=1+mod(n(2),ncity)  \( \ldots \text{and the city after } n(2). \)
x(1:4)=x(iorder(n(1:4)))  \( \text{Find coordinates for the four cities involved.} \)
yy(1:4)=y(iorder(n(1:4)))
de=alen(xx(1),xx(3),yy(1),yy(3))&  \( \text{Calculate cost of disconnecting the segment} \)
-alen(xx(2),xx(4),yy(2),yy(4))&  \( \text{at both ends and reconnecting in the opposite order.} \)
+alen(xx(1),xx(4),yy(1),yy(4))&
+alen(xx(2),xx(3),yy(2),yy(3))
END SUBROUTINE revcst
```

### Subroutine \textit{revers}(iorder,n)

```fortran
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
```

This routine performs a path segment reversal. \( iorder \) is an input array giving the present itinerary. The vector \( n \) has as its first four elements the first and last cities \( n(1), n(2) \) of the path segment to be reversed, and the two cities \( n(3) \) and \( n(4) \) that immediately precede and follow this segment. \( n(3) \) and \( n(4) \) are found by subroutine \textit{revcst}. On output, \( iorder \) contains the segment from \( n(1) \) to \( n(2) \) in reversed order.

```fortran
INTEGER(I4B) :: j,k,l,nn,ncity
ncity=size(iorder)
n=1+mod(n(2)-n(1)+ncity,ncity)/2  \( \text{This many cities must be swapped to effect} \)
do j=1,nn  \( \text{the reversal.} \)
k=1+mod((n(1)+j-2),ncity)  \( \text{Start at the ends of the segment and swap} \)
l=1+mod((n(2)-j+ncity),ncity)  \( \text{pairs of cities, moving toward the center.} \)
call swap(iorder(k),iorder(l))
end do
END SUBROUTINE revers
```

### Subroutine \textit{trncst}(x,y,iorder,n,de)

```fortran
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: n
REAL(SP), INTENT(OUT) :: de
```

This subroutine returns the value of the cost function for a proposed path segment transport. Arrays \( x \) and \( y \) give the city coordinates. \( iorder \) is an array giving the present itinerary. The first three elements of array \( n \) give the starting and ending cities of the path to be transported, and the point among the remaining cities after which it is to be inserted. On output, \( de \) is the cost of the change. The actual transport is not performed by this routine.

```fortran
INTEGER(I4B) :: ncity
REAL(SP), DIMENSION(6) :: xx,yy
ncity=size(x)
n(4)=1+mod(n(3),ncity)  \( \text{Find the city following } n(3) \ldots \)
n(5)=1+mod(n(1)+ncity-2,ncity)  \( \ldots \text{and the one preceding } n(1) \ldots \)
n(6)=1+mod(n(2),ncity)  \( \ldots \text{and the one following } n(2). \)
xx(1:6)=x(iorder(n(1:6)))
yy(1:6)=y(iorder(n(1:6)))
de=alen(xx(2),xx(6),yy(2),yy(6))&
alen(xx(1),xx(5),yy(1),yy(5))&
alen(xx(3),xx(4),yy(3),yy(4))&
alen(xx(1),xx(3),yy(1),yy(3))&
alen(xx(2),xx(4),yy(2),yy(4))&
alen(xx(5),xx(6),yy(5),yy(6))
END SUBROUTINE trncst
```

### Subroutine \textit{trnspt}(iorder,n)

```fortran
IMPLICIT NONE
```
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
This routine does the actual path transport, once metrop has approved. iorder is an input array giving the present itinerary. The array n has as its six elements the beginning n(1) and end n(2) of the path to be transported, the adjacent cities n(3) and n(4) between which the path is to be placed, and the cities n(5) and n(6) that precede and follow the path. n(4), n(5), and n(6) are calculated by subroutine trncst. On output, iorder is modified to reflect the movement of the path segment.

INTEGER(I4B) :: m1,m2,m3,nn,ncity
INTEGER(I4B), DIMENSION(size(iorder)) :: jorder
ncity=size(iorder)
m1=1+mod((n(2)-n(1)+ncity),ncity) Find number of cities from n(1) to n(2) ...
m2=1+mod((n(5)-n(4)+ncity),ncity) ...and the number from n(4) to n(5)
m3=1+mod((n(3)-n(6)+ncity),ncity) ...and the number from n(6) to n(3).
jorder(1:m1)=iorder(1+mod((arth(1,1,m1)+n(1)-2),ncity)) Copy the chosen segment. nn=m1
jorder(nn+1:nn+m2)=iorder(1+mod((arth(1,1,m2)+n(4)-2),ncity)) Then copy the segment from n(4) to n(5).
nn=nn+m2
jorder(nn+1:nn+m3)=iorder(1+mod((arth(1,1,m3)+n(6)-2),ncity)) Finally, the segment from n(6) to n(3).
jorder(1:ncity)=jorder(1:ncity) Copy jorder back into iorder.
END SUBROUTINE trnspt
END SUBROUTINE anneal

* * *

SUBROUTINE amebsa(p,y,pb,yb,ftol,func,iter,temptr)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,iminloc,swap
USE nr, ONLY : ran1
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: iter
REAL(SP), INTENT(INOUT) :: yb
REAL(SP), INTENT(IN) :: ftol,temptr
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y,pb
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: p
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: NMAX=200
Minimization of the N-dimensional function func by simulated annealing combined with the downhill simplex method of Nelder and Mead. The (N+1) x N matrix p is input. Its N+1 rows are N-dimensional vectors that are the vertices of the starting simplex. Also input is the vector y of length N+1, whose components must be preinitialized to the values of func evaluated at the N+1 vertices (rows) of p; ftol, the fractional convergence tolerance to be achieved in the function value for an early return; iter, and tempotr. The routine makes iter function evaluations at an annealing temperature tempotr, then returns. You should then decrease tempotr according to your annealing schedule, reset iter, and call the routine again (leaving other arguments unaltered between calls). If iter is returned with a positive value, then early convergence and return occurred. If you initialize yb to a very large value on the first call, then yb and pb (an array of length N) will subsequently return the best function value and point ever encountered (even if it is no longer a point in the simplex).
INTEGER(I4B) :: ihi,ndim
REAL(SP) :: yhi
REAL(SP), DIMENSION(size(p,2)) :: psum
call amebsa_private
SUBROUTINE amebsa_private

INTEGER(I4B) :: i, ilo, inhi
REAL(SP) :: rtol, ylo, yhi, ytry
REAL(SP), DIMENSION(size(y)) :: yt, harvest
ndim = assert_eq(size(p,2), size(p,1)-1, size(y)-1, size(pb),'amebsa')
psum(:) = sum(p(:,,:),dim=1)
do
  call rani(harvest)
  yt(:) = y(:) - temptr*log(harvest)
  Whenever we "look at" a vertex, it gets a random thermal fluctuation.
  ilo = iminloc(yt(:))
  Determine which point is the highest (worst),
  ylo = yt(ilo)
  next-highest, and lowest (best).
  ihi = imaxloc(yt(:))
  yhi = yt(ihi)
  yt(ihi) = ylo
  inhi = imaxloc(yt(:))
  ynhi = yt(inhi)
  rtol = 2.0_sp*abs(yhi-ylo)/(abs(yhi)+abs(ylo))
  Compute the fractional range from highest to lowest and return if satisfactory,
  if (rtol < ftol .or. iter < 0) then
    If returning, put best point and value in
    call swap(y(1),y(ilo))
    slot 1.
    call swap(p(1,:),p(ilo,:))
    RETURN
  end if
  Begin a new iteration. First extrapolate by a factor $-1$ through the face of the simplex
  across from the high point, i.e., reflect the simplex from the high point.
  ytry = amotsa(-1.0_sp)
  iter = iter - 1
  if (ytry <= ylo) then
    gives a result better than the best point, so
    ytry = amotsa(2.0_sp)
    try an additional extrapolation by a fac-
    tor of 2.
  else if (ytry >= ynhi) then
    The reflected point is worse than the second-
    highest, so look for an intermediate lower
    point, i.e., do a one-dimensional contrac-
    tion.
    ysave = yhi
    ytry = amotsa(0.5_sp)
    iter = iter - 1
    if (ytry >= ysave) then
      Can't seem to get rid of that high point. Better contract around the lowest
      (best) point.
      p(:, :) = 0.5_sp*(p(:, :)+spread(p(ilo,:),1,size(p,1))
      do i=1, ndim + 1
        if (i /= ilo) y(i) = func(p(i,:))
        end do
      iter = iter - ndim
      Keep track of function evaluations.
      psum(:) = sum(p(:,,:),dim=1)
    end if
  end if
end do
END SUBROUTINE amebsa_private

FUNCTION amotsa(fac)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: fac
REAL(SP) :: amotsa
Extrapolates by a factor fac through the face of the simplex across from the high point,
tries it, and replaces the high point if the new point is better.
REAL(SP) :: fac1, fac2, yflu, ytry, harv
REAL(SP), DIMENSION(size(p,2)) :: ptry
fac1 = (1.0_sp-fac)/ndim
fac2 = fac1*fac
ptry(:) = psum(:)*fac1-p(ilo,:)*fac2
ytry = func(ptry)
if (ytry <= yb) then
  Save the best-ever.
  pb(:) = ptry(:)
yb=ytry
end if
call ran1(harv)
yflu=ytry+temptr*log(harv)
if (yflu < yhi) then
  yhi=yflu
  psum(:)=psum(:)-p(ihi,:)+ptry(:)
  p(ihi,:)=ptry(:)
end if
amotsa=yflu
END FUNCTION amotsa
END SUBROUTINE amebsa

We added a thermal fluctuation to all the current vertices, but we subtract it here, so as to give the simplex a thermal Brownian motion: It likes to accept any suggested change.

See the discussion of amoeba on p. 1209 for why the routine is coded this way.
Chapter B11. Eigensystems

SUBROUTINE jacobi(a,d,v,nrot)
USE nrtype; USE nrutil, ONLY : assert_eq, get_diag, nrerror, unit_matrix, &
upper_triangle
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: nrot
REAL(SP), DIMENSION(:,), INTENT(OUT) :: d
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: v

Computes all eigenvalues and eigenvectors of a real symmetric \( N \times N \) matrix \( a \). On output,
elements of \( a \) above the diagonal are destroyed. \( d \) is a vector of length \( N \) that returns the
eigenvalues of \( a \). \( v \) is an \( N \times N \) matrix whose columns contain, on output, the normalized
eigenvectors of \( a \). \( nrot \) returns the number of Jacobi rotations that were required.

INTEGER(I4B) :: i,ip,iq,n
REAL(SP) :: c,g,h,s,sm,t,tau,theta,tresh
REAL(SP), DIMENSION(size(d)) :: b,z
n=assert_eq((/size(a,1),size(a,2),size(d),size(v,1),size(v,2)/),'jacobi')
call unit_matrix(v(:,:))
b(:)=get_diag(a(:,:))
d(:)=b(:)
z(:)=0.0
nrot=0
do i=1,50
  sm=sum(abs(a),mask=upper_triangle(n,n))
  if (sm == 0.0) RETURN
  tresh=merge(0.2_sp*sm/n**2,0.0_sp, i < 4 )
  on the first three sweeps, we will rotate only if tresh exceeded.
do ip=1,n-1
  do iq=ip+1,n
    g=100.0_sp*abs(a(ip,iq))
    After four sweeps, skip the rotation if the off-diagonal element is small.
    if ((i > 4) .and. (abs(d(ip))+g == abs(d(ip))) & &
    .and. (abs(d(iq))+g == abs(d(iq)))) then
      a(ip,iq)=0.0
    else if (abs(a(ip,iq)) > tresh) then
      h=d(iq)-d(ip)
      if (abs(h)+g == abs(h)) then
        t=a(ip,iq)/h
        theta=0.5_sp*h/a(ip,iq)
        Equation (11.1.10).
        t=1.0_sp/(abs(theta)+sqrt(1.0_sp+theta**2))
        if (theta < 0.0) t=-t
      else
        theta=0.5_sp*h/a(ip,iq)
        Equation (11.1.10).
        t=1.0_sp/(abs(theta)+sqrt(1.0_sp+theta**2))
        if (theta < 0.0) t=-t
      end if
      c=1.0_sp/sqrt(1+t**2)
      s=sign(c)
      tau=c/(1.0_sp+c)
      d(ip)=d(ip)+h
      d(iq)=d(iq)+h
      d(ip)=d(iq)+h
      d(iq)=d(iq)+h
      a(ip,iq)=0.0
  end do
  nrot=nrot+1
  sm=sum(abs(a),mask=upper_triangle(n,n))
end do
}
call jrotate(a(1:ip-1,ip),a(1:ip-1,iq))
Case of rotations 1 ≤ j < p.
call jrotate(a(ip,ip+1:iq-1),a(ip+1:iq-1,iq))
Case of rotations p < j < q.
call jrotate(a(ip,iq+1:n),a(iq,iq+1:n))
Case of rotations q < j ≤ n.
call jrotate(v(:,ip),v(:,iq))
nrot=nrot+1
end if
end do
end do
b(:)=b(:)+z(:)
d(:)=b(:)
Update d with the sum of \( ta_{pq} \), and reinitialize \( z \).
call nrerror('too many iterations in jacobi')
CONTAINS
SUBROUTINE jrotate(a1,a2)
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a1,a2
REAL(SP), DIMENSION(size(a1)) :: wk1
wk1(:)=a1(:)
end do
a1(:)=a1(:)-s*(a2(:)+a1(:)*tau)
a2(:)=a2(:)+s*(wk1(:)-a2(:)*tau)
END SUBROUTINE jrotate
END SUBROUTINE jacobi

As discussed in Volume 1, jacobi is generally not competitive with tqli in terms of efficiency. However, jacobi can be parallelized whereas tqli uses an intrinsically serial algorithm. The version of jacobi implemented here is likely to be adequate for a small-scale parallel (SSP) machine, but is probably still not competitive with tqli. For a massively multiprocessor (MMP) machine, the order of the rotations needs to be chosen in a more complicated pattern than here so that the rotations can be executed in parallel. In this case the Jacobi algorithm may well turn out to be the method of choice. Parallel replacements for tqli based on a divide and conquer algorithm have also been proposed. See the discussion after tqli on p. 1229.

These routines in nrutil both require access to the diagonal of a matrix, an operation that is not conveniently provided for in Fortran 90. We have split them off into nrutil in case your compiler provides parallel library routines so you can replace our standard versions.

The upper_triangle function in nrutil returns an upper triangular logical mask. As used here, the mask is true everywhere in the upper triangle of an \( n \times n \) matrix, excluding the diagonal. An optional integer argument extra allows additional diagonals to be set to true. With extra=1 the upper triangle including the diagonal would be true. By using the mask, we can conveniently sum over the desired matrix elements in parallel.

SUBROUTINE jrotate(a1,a2)  This internal subroutine also uses the values of \( s \) and \( tau \) from the calling subroutine jacobi. Variables in the calling routine are visible to an internal subprogram, but you should be circumspect in making use of this fact. It is easy to overwrite a value in the calling program inadvertently, and it is
often difficult to figure out the logic of an internal routine if not all its variables are declared explicitly. However, \texttt{jrotate} is so simple that there is no danger here.

\begin{verbatim}
* * *
SUBROUTINE eigsort(d,v)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: d
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: v
Given the eigenvalues \(d\) and eigenvectors \(v\) as output from \texttt{jacobi} (§11.1) or \texttt{tqli} (§11.3), this routine sorts the eigenvalues into descending order, and rearranges the columns of \(v\) correspondingly. The method is straight insertion.

INTEGER(I4B) :: i,j,n
n=assert_eq(size(d),size(v,1),size(v,2),'eigsort')
do i=1,n-1
  j=imaxloc(d(i:n))+i-1
  if (j /= i) then
    call swap(d(i),d(j))
    call swap(v(:,i),v(:,j))
  end if
end do
END SUBROUTINE eigsort
\end{verbatim}

\texttt{j=imaxloc...} See discussion of \texttt{imaxloc} on p. 1017.

\begin{verbatim}
* * *
SUBROUTINE tred2(a,d,e,novectors)
USE nrtype; USE nrutil, ONLY : assert_eq,outerprod
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: d,e
LOGICAL(LGT), OPTIONAL, INTENT(IN) :: novectors
Householder reduction of a real, symmetric, \(N \times N\) matrix \(a\). On output, \(a\) is replaced by the orthogonal matrix \(Q\) effecting the transformation. \(d\) returns the diagonal elements of the tridiagonal matrix, and \(e\) the off-diagonal elements, with \(e(1)=0\). If the optional argument \texttt{novectors} is present and \texttt{true}, only eigenvalues are to be found subsequently, in which case \(a\) contains no useful information on output.

INTEGER(I4B) :: i,j,l,n
REAL(SP) :: f,g,h,hh,scale
REAL(SP), DIMENSION(size(a,1)) :: gg
LOGICAL(LGT) :: yesvec
n=assert_eq(size(a,1),size(a,2),size(d),size(e),'tred2')
if (present(novectors)) then
  yesvec=.not. novectors
else
  yesvec=.true.
end if
do i=n,2,-1
  l=i-1
  h=0.0
  if (l > 1) then
    scale=sum(abs(a(i,1:l)))
    if (scale == 0.0) then
      Skip transformation.
    else
      e(1)=a(i,1,1)
      a(i,1:l)=a(i,1:l)/scale
      h=sum(a(i,1:l)**2)
      Use scaled \(a\)'s for transformation.
      Form \(\sigma\) in \(h\).
    \end{verbatim}
f = a(i,1)
g = -sign(sqrt(h), f)
e(i) = scale * g
h = h - f * g

Now \( h \) is equation (11.2.4).

if \( \text{yesvec} \) then a(1:l,i) = a(i,1:l) / h
end if

** ⋆ ⋆ ⋆

SUBROUTINE tqli(d,e,z)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : pythag
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: d,e
REAL(SP), DIMENSION(:,,:), OPTIONAL, INTENT(INOUT) :: z

QL algorithm with implicit shifts, to determine the eigenvalues and eigenvectors of a real, symmetric, tridiagonal matrix, or of a real, symmetric matrix previously reduced by tred2.
§11.2. \( \mathbf{d} \) is a vector of length \( N \). On input, its elements are the diagonal elements of the tridiagonal matrix. On output, it returns the eigenvalues. The vector \( \mathbf{e} \) inputs the subdiagonal elements of the tridiagonal matrix, with \( e(1) \) arbitrary. On output \( \mathbf{e} \) is destroyed. When finding only the eigenvalues, the optional argument \( \mathbf{z} \) is omitted. If the eigenvectors of a tridiagonal matrix are desired, the \( N \times N \) matrix \( \mathbf{z} \) is input as the identity matrix. If the eigenvectors of a matrix that has been reduced by \texttt{tred2} are required, then \( \mathbf{z} \) is input as the matrix output by \texttt{tred2}. In either case, the \( k \)th column of \( \mathbf{z} \) returns the normalized eigenvector corresponding to \( d(k) \).

```fortran
INTEGER(I4B) :: i,iter,l,m,n,ndum
REAL(SP) :: b,c,dd,f,g,p,r,s
REAL(SP), DIMENSION(size(e)) :: ff
n=assert_eq(size(d),size(e),'tqli: n')
if (present(z)) ndum=assert_eq(n,size(z,1),size(z,2),'tqli: ndum')
e(:)=eoshift(e(:),1)
    Convenient to renumber the elements of e.
do l=1,n
    iterate: do
        dd=abs(d(m))+abs(d(m+1))
            Look for a single small subdiagonal element to split the matrix.
        if (abs(e(m))+dd == dd) exit
    end do
    if (m == l) exit iterate
    if (iter == 30) call nrerror('too many iterations in tqli')
    iter=iter+1
    g=(d(l+1)-d(l))/(2.0_sp*e(l))
        Form shift.
    r=pythag(g,1.0_sp)
    g=(d(m)-d(l)+e(l)/(g+sign(r,g))
        This is \( d_m - k_s \).
    s=1.0
    c=1.0
    p=0.0
    do i=m-1,l,-1
        A plane rotation as in the original QL, followed by Givens rotations to restore tridiagonal form.
            f=s*e(i)
            b=c*e(i)
            r=pythag(f,g)
            e(i+1)=r
                Recover from underflow.
        if (r == 0.0) then
            d(i+1)=d(i+1)-p
            e(m)=0.0
            cycle iterate
        end if
            s=f/r
            c=g/r
            g=d(i+1)-p
                Form eigenvectors.
            r=(d(i)-g)*s+2.0_sp*c*b
            p=r
            if (present(z)) then
                ff(:)=z(:)
                z(:)=s*z(:)+c*ff(:)
                z(i)=c*z(i)-s*ff(i)
            end if
        end do
        d(l)=d(l)-p
        e(l)=g
        e(m)=0.0
    end do iterate
end do
END SUBROUTINE tqli
```

The routine \texttt{tqli} is intrinsically serial. A parallel replacement based on a divide and conquer algorithm has been proposed [1,2]. The idea is to split the tridiagonal matrix recursively into two tridiagonal matrices of
half the size plus a correction. Given the eigensystems of the two smaller tridiagonal matrices, it is possible to join them together and add in the effect of the correction. When some small size of tridiagonal matrix is reached during the recursive splitting, its eigensystem is found directly with a routine like \texttt{tqli}. Each of these small problems is independent and can be assigned to an independent processor. The procedures for sewing together can also be done independently. For very large matrices, this algorithm can be an order of magnitude faster than \texttt{tqli} even on a serial machine, and no worse than a factor of 2 or 3 slower, depending on the matrix. Unfortunately the parallelism is not well expressed in Fortran 90. Also, the sewing together requires quite involved coding. For an implementation see the LAPACK routine \texttt{SSTEDC}. Another parallel strategy for eigensystems uses inverse iteration, where each eigenvalue and eigenvector can be found independently [3].

This routine uses \texttt{z} as an optional argument that is required only if eigenvectors are being found as well as eigenvalues.

\begin{verbatim}
iterate: do See discussion of named do loops after \texttt{simplx} on p. 1219.

*   *   *

SUBROUTINE balanc(a)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: a
REAL(SP), PARAMETER :: RADX=radix(a),SQRADX=RADX**2

 Given an \( N \times N \) matrix \( a \), this routine replaces it by a balanced matrix with identical eigenvalues. A symmetric matrix is already balanced and is unaffected by this procedure.

 The parameter \( \text{RADX} \) is the machine’s floating-point radix.

INTEGER(I4B) :: i,last,ndum
REAL(SP) :: c,f,g,r,s
ndum=assert_eq(size(a,1),size(a,2),'balanc')
do
  last=1
do i=1,size(a,1)
  c=sum(abs(a(:,i)))-a(i,i)
  r=sum(abs(a(i,:)))-a(i,i)
  if (c /= 0.0 .and. r /= 0.0) then
    g=r/RADX
    f=1.0
    s=c+r
    do
      if (c >= g) exit
      f=f*RADX
      c=c*SQRADX
    end do
    g=r*RADX
    do
      if (c <= g) exit
      f=f/RADX
      c=c/SQRADX
    end do
    if ((c+r)/f < 0.95_sp*s)
      last=0
      g=1.0_sp/f
      a(i,:)=a(i,:)*g
      a(:,i)=a(:,i)*f
    end if
  end if

*   *   *
\end{verbatim}
REAL(SP), PARAMETER :: RADX=radix(a)...

Fortran 90 provides a nice collection of numeric inquiry intrinsic functions. Here we find the machine’s floating-point radix. Note that only the type of the argument a affects the returned function value.

⋆⋆⋆

SUBROUTINE elmhes(a)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,outerprod,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: a
Reduction to Hessenberg form by the elimination method. The real, nonsymmetric, \( N \times N \) matrix \( a \) is replaced by an upper Hessenberg matrix with identical eigenvalues. Recommended, but not required, is that this routine be preceded by balanc. On output, the Hessenberg matrix is in elements \( a(i,j) \) with \( i \leq j + 1 \). Elements with \( i > j + 1 \) are to be thought of as zero, but are returned with random values.

INTEGER(I4B) :: i,m,n
REAL(SP) :: x
REAL(SP), DIMENSION(size(a,1)) :: y

n=assert_eq(size(a,1),size(a,2),'elmhes')
do m=2,n-1  
   m is called \( r + 1 \) in the text.
   i=imaxloc(abs(a(m:n,m-1)))+m-1  
   Find the pivot.
   x=a(i,m-1)
   if (i /= m) then  
       Interchange rows and columns.
       call swap(a(i,m-1:n),a(m,m-1:n))
       call swap(a(:,i),a(:,m))
   end if
   if (x /= 0.0) then  
       Carry out the elimination.
       y(m+1:n)=a(m+1:n,m-1)/x
       a(m+1:n,m-1)=y(m+1:n)
       a(m+1:n,m:n)=a(m+1:n,m:n)-outerprod(y(m+1:n),a(m,m:n))
       a(:,m)=a(:,m)+matmul(a(:,m+1:n),y(m+1:n))
   end if
end do

END SUBROUTINE elmhes

⋆⋆⋆

If the four lines of code starting here were all coded for a serial machine in a single do-loop starting with \( i=\text{m+1},n \) (see Volume 1), it would pay to test whether \( y \) was zero because the next three lines could then be skipped for that value of \( i \). There is no convenient way to do this here, even with a where, since the shape of the arrays on each of the three lines is different. For a parallel machine it is probably best just to do a few unnecessary multiplies and skip the test for zero values of \( y \).

⋆⋆⋆
SUBROUTINE hqr(a,wr,wi)
USE nrtype, USE nrutil, ONLY : assert_eq, diagadd, nrerror, upper_triangle
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: wr, wi
REAL(SP), DIMENSION(:, :), INTENT(INOUT) :: a

Finds all eigenvalues of an \( N \times N \) upper Hessenberg matrix \( a \). On input \( a \) can be exactly as output from elmhes \S 11.5; on output it is destroyed. The real and imaginary parts of the \( N \) eigenvalues are returned in \( wr \) and \( wi \), respectively.

INTEGER(I4B) :: i, its, k, l, m, n, nn, mnnk
REAL(SP) :: anorm, p, q, r, s, t, u, v, x, y, z
REAL(SP), DIMENSION(size(a, 1)) :: pp

\( n = \text{assert\_eq}(\text{size}(a, 1), \text{size}(a, 1), \text{size}(wr), \text{size}(wi), 'hqr') \)

Compute matrix norm for possible use in locating single small subdiagonal element.

\( nn = n \)

\( t = 0.0 \)  

Begin search for next eigenvalue: "Do while \( nn \geq 1 \)".

if (\( nn < 1 \)) exit

its = 0

Begin iteration.

Look for single small subdiagonal element.

\( s = \text{abs}(a(l-1, l-1)) + \text{abs}(a(l, l)) \)

if (\( s = 0.0 \)) \( s = \text{anorm} \)

if (\( \text{abs}(a(l, l-1)) + s = s \)) then

\( a(l, l-1) = 0.0 \)  

exit small

end if

\( x = a(nn, nn) \)

if (\( l = nn \)) then

One root found.

\( \text{wr}(nn) = x + t \)

\( \text{wi}(nn) = 0.0 \)

\( \text{nn} = \text{nn} - 1 \)  

exit iterate

end if

\( y = a(nn-1, nn-1) \)

\( w = a(nn, nn-1) * a(nn-1, nn) \)

if (\( l = nn-1 \)) then

Two roots found ...

\( p = 0.5 \_ sp \times (y - x) \)

\( q = p \times 2 + v \)

\( z = \text{sqr}(\text{abs}(q)) \)

\( x = t \)

if (\( q = 0.0 \)) then

...a real pair ...

\( z = \text{sign}(z, p) \)

\( \text{wr}(nn) = x + z \)

\( \text{wr}(nn-1) = \text{wr}(nn) \)

if (\( z = 0.0 \)) \( \text{wr}(nn-1) = x - w / z \)

\( \text{wi}(nn) = 0.0 \)

\( \text{wi}(nn-1) = 0.0 \)

else

...a complex pair.

\( \text{wr}(nn) = x + p \)

\( \text{wr}(nn-1) = \text{wr}(nn) \)

\( \text{wi}(nn) = z \)

\( \text{wi}(nn-1) = -z \)

end if

\( nn = nn - 2 \)

exit iterate

Go back for next eigenvalue.

end if

No roots found. Continue iteration.

if (\( \text{its} = 30 \)) call nrerror('too many iterations in hqr')

if (\( \text{its} = 10 \) or. \( \text{its} = 20 \)) then

Form exceptional shift.

\( \text{t} = t + x \)

call diagadd(a(1:nn, 1:nn), -x)

\( s = \text{abs}(a(nn, nn-1)) * \text{abs}(a(nn-1, nn-2)) \)
Chapter B11. Eigensystems

\[
x = 0.75 \times s
\]
\[
y = x
\]
\[
w = -0.4375 \times s^2
\]
end if

its = its + 1

do m = nn - 2, 1, -1

Form shift and then look for 2 consecutive small subdiagonal elements.

\[
z = a(m, m)
\]
\[
r = x - z
\]
\[
s = y - z
\]
\[
p = (r + s - w) / a(m + 1, m + 1) - r - s
\]
\[
q = a(m, m + 1) - z - r
\]
\[
\begin{align*}
\text{Equation (11.6.23).} \\
\text{Scale to prevent overflow or underflow.}
\end{align*}
\]
\[
r = a(m + 2, m + 1)
\]
\[
s = \text{abs}(p) \times \text{abs}(q) \times \text{abs}(r)
\]
\[
p = p / s
\]
\[
q = q / s
\]
\[
r = r / s
\]
if (m == l) exit

\[
v = \text{abs}(a(m, m - 1)) \times (\text{abs}(q) + \text{abs}(r))
\]
\[
u = \text{abs}(p) \times (\text{abs}(a(m - 1, m - 1)) + \text{abs}(z) + \text{abs}(a(m + 1, m + 1)))
\]
end do

\[
do i = m + 2, nn
\]
\[
a(i, i - 2) = 0.0
\]
if (i /= m + 2) a(i, i - 3) = 0.0
end do

do k = m, nn - 1

Double QR step on rows 1 to nn and columns m to nn.

if (k /= m) then

\[
p = a(k, k - 1)
\]
\[
q = a(k + 1, k - 1)
\]
\[
r = 0.0
\]
if (k == nn - 1) r = a(k + 2, k - 1)
\[
x = \text{abs}(p) + \text{abs}(q) + \text{abs}(r)
\]
if (x /= 0.0) then
\[
p = p / x
\]
\[
q = q / x
\]
\[
r = r / x
\]
end if
\[
s = \text{sign}(\sqrt{p^2 + q^2 + r^2}, p)
\]
if (s /= 0.0) then
if (k == m) then
\[
\text{if (l /= m) a(k, k - 1) = -a(k, k - 1) }
\]
else
\[
a(k, k - 1) = -s \times x
\]
end if
\[
p = p + s
\]
\[
x = p / s
\]
\[
y = q / s
\]
\[
z = r / s
\]
\[
q = q / p
\]
\[
r = r / p
\]
\[
\text{Ready for row modification.}
\]
\[
pp(k:nn) = a(k, k:nn) + q \times a(k + 1, k:nn)
\]
if (k /= nn - 1) then
\[
pp(k:nn) = pp(k:nn) + r \times a(k + 2, k:nn)
\]
\[
a(k + 2, k:nn) = a(k + 2, k:nn) - pp(k:nn) \times z
\]
end if
\[
a(k + 1, k:nn) = a(k + 1, k:nn) - pp(k:nn) \times y
\]
\[
a(k, k:nn) = a(k, k:nn) - pp(k:nn) \times x
\]
\[
mnnk = \text{min}(nn, k + 3)
\]
\[
\text{Column modification.}
\]
\[
pp(1:mnnk) = x \times a(1:mnnk, k) + y \times a(1:mnnk, k + 1)
\]
if (k /= nn - 1) then
\[
pp(1:mnnk) = pp(1:mnnk) + r \times a(1:mnnk, k + 2)
\]
\[
a(1:mnnk, k + 2) = a(1:mnnk, k + 2) - pp(1:mnnk) \times r
\]
end if
a(l:mmnk,k+1)=a(l:mmnk,k+1)-pp(l:mmnk)*q  
end if
end do
end do iterate
end do
END SUBROUTINE hqr

anorm=sum(abs(a),mask=upper_triangle(n,n,extra=2)  See the discussion of upper_triangle after jacobi on p. 1226. Setting extra=2 here picks out the upper Hessenberg part of the matrix.

iterate: do  We use a named loop to improve the readability and structuring of the routine. The if-blocks that test for one or two roots end with exit iterate, transferring control back to the outermost loop and thus starting a search for the next root.

call diagadd...  The routines that operate on the diagonal of a matrix are collected in nrutil partly so you can write clear code and partly in the hope that compiler writers will provide parallel library routines. Fortran 90 does not provide convenient parallel access to the diagonal of a matrix.

CITED REFERENCES AND FURTHER READING:
Chapter B12. Fast Fourier Transform

The algorithms underlying the parallel routines in this chapter are described in §22.4. As described there, the basic building block is a routine for simultaneously taking the FFT of each row of a two-dimensional matrix:

```fortran
SUBROUTINE fourrow_sp(data,isign)
USE nrtype; USE nrutil, ONLY : assert, swap
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign

Replaces each row (constant first index) of data(1:M,1:N) by its discrete Fourier transform (transform on second index), if isign is input as 1; or replaces each row of data by N times its inverse discrete Fourier transform, if isign is input as -1. N must be an integer power of 2. Parallelism is M-fold on the first index of data.

INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(SPC), DIMENSION(size(data,1)) :: temp
COMPLEX(DPC) :: w,wp
Double precision for the trigonometric recurrences.
COMPLEX(SPC) :: ws
n=size(data,2)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourrow_sp')
n2=n/2
j=n2

This is the bit-reversal section of the routine.
do i=1,n-2
   if (j > i) call swap(data(:,j+1),data(:,i+1))
m=n2
   do
      if (m < 2 .or. j < m) exit
      j=j-m
      m=m/2
   end do
   j=j+m
end do
mmax=1

Here begins the Danielson-Lanczos section of the routine.
do
   Outer loop executed \( \log_2 N \) times.
   istep=2*mmax
   theta=PI_D/(isign*mmax)
   wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
   w=cmplx(1.0_dp,0.0_dp,kind=dpc)
   do m=1,mmax
      Here are the two nested inner loops.
      ws=w
      do i=m,n,istep
         j=i+mmax
         temp=ws*data(:,j)
         data(:,j)=data(:,i)-temp
         data(:,i)=data(:,i)+temp
      end do
      w=w*wp+w
   end do
   mmax=istep
```

1235
Chapter B12. Fast Fourier Transform

SUBROUTINE fourrow_dp(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(DPC), DIMENSION(:,,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(DPC), DIMENSION(size(data,1)) :: temp
COMPLEX(DPC) :: w,wp
COMPLEX(DPC) :: ws
n=size(data,2)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourrow_dp')
n2=n/2
j=n2
do i=1,n-2
   if (j > i) call swap(data(:,j+1),data(:,i+1))
   n=n2
do
   if (m < 2 .or. j < m) exit
   j=j-m
   m=m/2
end do
j=j+m
end do
mmax=1
do
   if (n <= mmax) exit
   istep=2*mmax
   theta=PI_D/(isign*mmax)
   wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
   w=cmplx(1.0_dp,0.0_dp,kind=dpc)
   do m=1,mmax
      ws=w
      do i=m,n,istep
         j=i+mmax
         temp=ws*data(:,j)
         data(:,j)=data(:,i)-temp
         data(:,i)=data(:,i)+temp
      end do
      w=w*wp+w
   end do
   mmax=istep
end do
END SUBROUTINE fourrow_dp

SUBROUTINE fourrow_3d(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
If isign is input as 1, replaces each third-index section (constant first and second indices) of data(1:L,1:M,1:N) by its discrete Fourier transform (transform on third index); or
Chapter B12. Fast Fourier Transform

replaces each third-index section of data by \( N \) times its inverse discrete Fourier transform, if \( \text{isign} \) is input as \(-1\). \( N \) must be an integer power of 2. Parallelism is \( L \times M \)-fold on the first and second indices of data.

\[
\begin{align*}
\text{INTEGER}(14B) & :: n,i,\text{istep},j,m,mmax,n2 \\
\text{REAL}(DP) & :: \theta \\
\text{COMPLEX}(SPC), \text{DIMENSION(size(data,1),size(data,2))} & :: \text{temp} \\
\text{COMPLEX}(DP) & :: w,wp \\
\text{COMPLEX}(SPC) & :: ws
\end{align*}
\]

Double precision for the trigonometric recurrences.

\[
n = \text{size(data,3)} \\
\text{call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourrow_3d')} \\
n2 = n/2 \\
j = n2 \\
\]

This is the bit-reversal section of the routine.

\[
\text{do } i=1,n-2 \\
\text{if } (j > i) \text{ call swap(data(:,j+1),data(:,i+1))} \\
m = n2 \\
\text{do} \\
\text{if } (m < 2 \text{ or } j < m) \text{ exit} \\
j = j-m \\
m = m/2 \\
\text{end do} \\
j = j+m \\
\text{end do} \\
mmax = 1
\]

Here begins the Danielson-Lanczos section of the routine.

\[
\text{do } \text{Outer loop executed } \log_2 N \text{ times.} \\
\text{if } (n <= mmax) \text{ exit} \\
\text{istep} = 2^{*\text{mmax}} \\
\theta = 2.0 \pi D/(\text{isign} * \text{mmax}) \\
w = \text{cmplx}(1.0,0.0,\text{kind=dp}) \\
wp = \text{cmplx}(1.0,0.0,\text{kind=dp}) \\
\text{do } m = 1,\text{mmax} \\
\text{ws} = w \\
\text{do } i = m,n,\text{istep} \\
j = i+\text{mmax} \\
\text{temp} = ws * \text{data}(:,j) \\
\text{data}(:,j) = \text{data}(:,i) - \text{temp} \\
\text{data}(:,i) = \text{data}(:,i) + \text{temp} \\
\text{end do} \\
w = wp * w \\
\text{end do} \\
mmax = \text{istep} \\
\text{end do} \\
\text{END SUBROUTINE fourrow_3d}
\]

\[\star \star \star\]

Exactly as in the preceding routines, we can take the FFT of each column of a two-dimensional matrix, and for each first-index section of a three-dimensional array.

\[
\text{SUBROUTINE fourcol(data,\text{isign})} \\
\text{USE nrtypes; USE nrutil, ONLY : assert,swap} \\
\text{IMPLICIT NONE} \\
\text{COMPLEX(SPC), DIMENSION(:,,:), INTENT(INOUT)} :: \text{data} \\
\text{INTEGER(14B), INTENT(IN)} :: \text{isign} \\
\]

Replaces each column (constant second index) of \( \text{data}(1:N,1:M) \) by its discrete Fourier transform (transform on first index), if \( \text{isign} \) is input as \(1\); or replaces each row of \( \text{data} \).
by \( N \) times its inverse discrete Fourier transform, if \( \text{isign} \) is input as \( -1 \). \( N \) must be an integer power of 2. Parallelism is \( M \)-fold on the second index of \( \text{data} \).

\[
\begin{align*}
\text{INTEGER(I4B)} &:: n, i, \text{istep}, j, m, \text{mmax}, n2 \\
\text{REAL(DP)} &:: \theta \\
\text{COMPLEX(SPC)}, \text{DIMENSION}(\text{size(data,2)}) &:: \text{temp} \\
\text{COMPLEX(DPC)} &:: w, \text{wp} \\
\text{COMPLEX(SPC)} &:: \text{ws}
\end{align*}
\]

\[
\begin{align*}
n &\text{=} \text{size(data,1)} \\
\text{call assert}(\text{iand}(n, n-1) == 0, 'n must be a power of 2 in fourcol') \\
n2 &\text{=} n/2 \\
j &\text{=} n2
\end{align*}
\]

This is the bit-reversal section of the routine.

\[
\begin{align*}
d &\text{=} i, n-2 \\
&\text{do } i = 1, n-2 \\
&\quad \text{if (} j > i \text{) call swap(data(}j+1,:),data(i+1,:)) \\
&\quad m = n2 \\
&\quad \text{do } \\
&\quad \quad \text{if (} m < 2 . \text{or. } j < m \text{) exit} \\
&\quad \quad j = j - m \\
&\quad \quad m = m/2 \\
&\quad \text{end do} \\
&\quad j = j + m \\
&\text{end do}
\end{align*}
\]

Here begins the Danielson-Lanczos section of the routine.

\[
\begin{align*}
d &\text{=} \text{mmax} \\
o &\text{=} n, \text{mmax} \\
\text{istep} &\text{=} 2 \times \text{mmax} \\
\theta &\text{=} \text{PI_D}(\text{sign} \times \text{mmax}) \\
wp &\text{=} \text{cmplx}(2.0_dp \times \sin(0.5_dp \times \theta)) \times 2, \text{sin} (\theta), \text{kind=} \text{dpc} \\
w &\text{=} \text{cmplx}(1.0_dp, 0.0_dp, \text{kind=} \text{dpc}) \\
&\text{do } m = 1, \text{mmax} \\
&\quad \text{ws} = w \\
&\quad \text{do } i = m, n, \text{istep} \\
&\quad \quad j = i + m \\
&\quad \quad \text{temp} = \text{ws} \times \text{data}(j,:) \\
&\quad \quad \text{data}(j,:) = \text{data}(i,:) - \text{temp} \\
&\quad \quad \text{data}(i,:) = \text{data}(i,:) + \text{temp} \\
&\quad \text{end do} \\
&\quad w = w \times \text{wp} + w \\
&\text{end do} \\
\text{mmax} &\text{=} \text{istep} \\
&\text{end do}
\end{align*}
\]

SUBROUTINE fourcol_3d(data, isign)
USE nrtype; USE nrutil, ONLY : assert, swap
IMPLICIT NONE
INTEGER(SPC), DIMENSION(:, :, :) , INTENT(INOUT) :: data
INTEGER(I4B) , INTENT(IN) :: isign

If \( \text{isign} \) is input as 1, replaces each first-index section (constant second and third indices) of \( \text{data}(:, 1:M, 1:L) \) by its discrete Fourier transform (transform on first index); or replaces each first-index section of \( \text{data} \) by \( N \) times its inverse discrete Fourier transform, if \( \text{isign} \) is input as \( -1 \). \( N \) must be an integer power of 2. Parallelism is \( M \times L \)-fold on the second and third indices of \( \text{data} \).

\[
\begin{align*}
n &\text{=} \text{size(data,1)} \\
\text{call assert}(\text{iand}(n, n-1) == 0, 'n must be a power of 2 in fourcol_3d') \\
n2 &\text{=} n/2
\end{align*}
\]
This is the bit-reversal section of the routine.

do i=1,n-2
  if (j > i) call swap(data(j+1,:,:),data(i+1,:,:))
  j=n2
  do
    if (m < 2 .or. j < m) exit
    j=j-m
    m=m/2
  end do
  j=j+m
end do

Here begins the Danielson-Lanczos section of the routine.

do Outer loop executed $\log_2 N$ times.
  istep=2*mmax
  theta=PI_D/(isign*mmax)
  wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
  w=cmplx(1.0_dp,0.0_dp,kind=dpc)
  do m=1,mmax
    temp=wp*data(j,:,:)
    data(j,:,:)=data(i,:,:)-temp
    data(i,:,:)=data(i,:,:)+temp
  end do
  w=w*wp+w
end do
mmax=istep
end do

Here now are implementations of the method of §22.4 for the FFT of one-dimensional single- and double-precision complex arrays:

SUBROUTINE four1_sp(data,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : fourrow
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign

Replaces a complex array data by its discrete Fourier transform, if isign is input as 1; or replaces data by its inverse discrete Fourier transform times the size of data, if isign is input as -1. The size of data must be an integer power of 2. Parallelism is achieved by internally reshaping the input array to two dimensions. (Use this version if fourrow is faster than fourcol on your machine.)

COMPLEX(SPC), DIMENSION(:,), ALLOCATABLE :: dat,temp
COMPLEX(DPC), DIMENSION(:,), ALLOCATABLE :: w,wp
REAL(DP), DIMENSION(:,), ALLOCATABLE :: theta
INTEGER(I4B) :: n,m1,m2,j
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_sp')

Find dimensions as close to square as possible, allocate space, and reshape the input array.

m1=2**ceiling(0.5_sp*log(real(n,sp))/0.693147_sp)
m2=n/m1
allocate(dat(m1,m2),theta(m1),w(m1),wp(m1),temp(m2,m1))
dat=reshape(data,shape(dat))
call fourrow(dat,isign) Transform on second index.
theta = arth(0, isign, m1) * TWOPI_D/n

wp = cmplx(-2.0_dp * sin(0.5_dp * theta)**2, sin(theta), kind=dpc)
w = cmplx(1.0_dp, 0.0_dp, kind=dpc)

do j=2, m2
    w = w * wp + w
    dat(:, j) = dat(:, j) * w
end do

temp = transpose(dat)
call fourrow(temp, isign)
data = reshape(temp, shape(data))
deallocate(dat, w, wp, theta, temp)

END SUBROUTINE four1_sp

SUBROUTINE four1_dp(data, isign)
USE nrtypes, USE nrutil, ONLY : arth, assert
USE nr, ONLY : fourrow
IMPLICIT NONE
COMPLEX(DPC), DIMENSION(:,), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(DPC), DIMENSION(:,,:), ALLOCATABLE :: dat,temp
COMPLEX(DPC), DIMENSION(:,), ALLOCATABLE :: w,wp
REAL(DP), DIMENSION(:,), ALLOCATABLE :: theta
INTEGER(I4B) :: n,m1,m2,j

n = size(data)
call assert(iand(n, n-1) == 0, 'n must be a power of 2 in four1_dp')
m1 = 2**(ceiling(log(real(n,sp))/0.693147_sp))
m2 = n/m1
allocate(dat(m1,m2), theta(m1), w(m1), wp(m1), temp(m2,m1))

dat = reshape(data, shape(dat))
call fourrow(dat, isign)
theta = arth(0, isign, m1) * TWOPI_D/n
wp = cmplx(-2.0_dp * sin(0.5_dp * theta)**2, sin(theta), kind=dpc)
w = cmplx(1.0_dp, 0.0_dp, kind=dpc)

do j=2, m2
    w = w * wp + w
    dat(:, j) = dat(:, j) * w
end do

temp = transpose(dat)
call fourrow(temp, isign)
data = reshape(temp, shape(data))
deallocate(dat, w, wp, theta, temp)

END SUBROUTINE four1_dp

The above routines use fourrow exclusively, on the assumption that it is faster than its sibling fourcol. When that is the case (as we typically find), it is likely that four1_sp is also faster than Volume 1’s scalar four1. The reason, on scalar machines, is that fourrow’s parallelism is taking better advantage of cache memory locality.

If fourrow is not faster than fourcol on your machine, then you should instead try the following alternative FFT version that uses fourcol only.

SUBROUTINE four1_alt(data, isign)
USE nrtypes, USE nrutil, ONLY : arth, assert
USE nr, ONLY : fourcol
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign

Replaces a complex array data by its discrete Fourier transform, if isign is input as 1; or replaces data by its inverse discrete Fourier transform times the size of data, if isign is
input as \(-1\). The size of data must be an integer power of 2. Parallelism is achieved by internally reshaping the input array to two dimensions. (Use this version only if fourcol is faster than fourrow on your machine.)

```fortran
COMPLEX(SPC), DIMENSION(:,,:), ALLOCATABLE :: dat,temp
COMPLEX(DPC), DIMENSION(:,), ALLOCATABLE :: w,wp
REAL(DP), DIMENSION(:,), ALLOCATABLE :: theta
INTEGER(I4B) :: n,m1,m2,j
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_alt')
Find dimensions as close to square as possible, allocate space, and reshape the input array.
m1=2**ceiling(0.5_sp*log(real(n,sp))/0.693147_sp)
m2=n/m1
allocate(dat(m1,m2),theta(m1),w(m1),wp(m1),temp(m2,m1))
dat=reshape(data,shape(dat))
temp=transpose(dat)
Transpose and transform on (original) second index.
call fourcol(temp,isign)
theta=arcth(0,isign,m1)*TWOPI_D/n
Set up recurrence.
w=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
do j=2,m2
Multiply by the extra phase factor.
w=w*wp+w
end do
temp=transpose(temp)
Transpose, and transform on (original) first index.
call fourcol(temp,isign)
data=transpose(temp)
Transpose and then reshape the result back to one dimension.
desallocate(dat,w,wp,theta,temp)
END SUBROUTINE four1_alt
```

With all the machinery of fourrow and fourcol, two-dimensional FFTs are extremely straightforward. Again there is an alternative version provided in case your hardware favors fourcol (which would be, we think, unusual).

```fortran
SUBROUTINE four2(data,isign)
USE nrtype
USE nr, ONLY : fourrow
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
Replaces a 2-d complex array `data` by its discrete 2-d Fourier transform, if `isign` is input as \(1\); or replaces `data` by its inverse 2-d discrete Fourier transform times the product of its two sizes, if `isign` is input as \(-1\). Both of `data`'s sizes must be integer powers of 2 (this is checked for in `fourrow`). Parallelism is by use of `fourrow`.
COMPLEX(SPC), DIMENSION(size(data,2),size(data,1)) :: temp
call fourrow(data,isign)
Transform in second dimension.
temp=transpose(data)
Tranpose.
call fourrow(temp,isign)
Transform in (original) first dimension.
data=transpose(temp)
Transpose into data.
END SUBROUTINE four2
```
SUBROUTINE four2_alt(data,isign)
USE nrtype
USE nr, ONLY : fourcol
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
Replaces a 2-d complex array data by its discrete 2-d Fourier transform, if isign is input as 1; or replaces data by its inverse 2-d discrete Fourier transform times the product of its two sizes, if isign is input as \(-1\). Both of data's sizes must be integer powers of 2 (this is checked for in fourcol). Parallelism is by use of fourcol. (Use this version only if fourcol is faster than fourrow on your machine.)
COMPLEX(SPC), DIMENSION(size(data,2),size(data,1)) :: temp
temp=transpose(data)  
Transposes.
call fourcol(temp,isign)  
Transform in (original) second dimension.
data=transpose(temp)  
Transposes.
call fourcol(data,isign)  
Transform in (original) first dimension.
END SUBROUTINE four2_alt

*   *   *

Most of the remaining routines in this chapter simply call one or another of the above FFT routines, with a small amount of auxiliary computation, so they are fairly straightforward conversions from their Volume 1 counterparts.

SUBROUTINE twofft(data1,data2,fft1,fft2)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : four1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: fft1,fft2
Given two real input arrays data1 and data2 of length \(N\), this routine calls four1 and returns two complex output arrays, fft1 and fft2, each of complex length \(N\), that contain the discrete Fourier transforms of the respective data arrays. \(N\) must be an integer power of 2.
INTEGER(I4B) :: n,n2
COMPLEX(SPC), PARAMETER :: C1=(0.5_sp,0.0_sp), C2=(0.0_sp,-0.5_sp)
COMPLEX, DIMENSION(size(data1)/2+1) :: h1,h2
n=assert_eq(size(data1),size(data2),size(fft1),size(fft2),'twofft')
call assert(iand(n,n-1)==0, 'n must be a power of 2 in twofft')
fft1=cmplx(data1,data2,kind=spc)  
Pack the two real arrays into one complex array.
call four1(fft1,1)  
Transform the complex array.
fft2(1)=cmplx(aimag(fft1(1)),0.0_sp,kind=spc)
fft2(1)=cmplx(real(fft1(1)),0.0_sp,kind=spc)
h1(2:n2)=C1*(fft1(2:n2)+conjg(fft1(n:n2:-1)))  
Use symmetries to separate the
h2(2:n2)=C2*(fft1(2:n2)-conjg(fft1(n:n2:-1)))  
two transforms.
fft1(2:n2)=h1(2:n2)  
Ship them out in two complex arrays.
fft1(n:n2:-1)=conjg(h1(2:n2))
fft2(2:n2)=h2(2:n2)
fft2(n:n2:-1)=conjg(h2(2:n2))
END SUBROUTINE twofft

*   *   *
SUBROUTINE realft_sp(data,isign,zdata)
USE nrtype; USE nrutil, ONLY : assert,assert_eq,zroots_unity
USE nr, ONLY : four1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(SPC), DIMENSION(:), OPTIONAL, TARGET :: zdata

When isign = 1, calculates the Fourier transform of a set of $N$ real-valued data points, input in the array data. If the optional argument zdata is not present, the data are replaced by the positive frequency half of its complex Fourier transform. The real-valued first and last components of the complex transform are returned as elements data(1) and data(2), respectively. If the complex array zdata of length $N/2$ is present, data is unchanged and the transform is returned in zdata. $N$ must be a power of 2. If isign = -1, this routine calculates the inverse transform of a complex data array if it is the transform of real data. (Result in this case must be multiplied by $2/N$.) The data can be supplied either in data, with zdata absent, or in zdata.

INTEGER(I4B) :: n,ndum,nh,nq
COMPLEX(SPC), DIMENSION(size(data)/4) :: w
COMPLEX(SPC), DIMENSION(size(data)/4-1) :: h1,h2
COMPLEX(SPC), DIMENSION(:), POINTER :: cdata
   Used for internal complex computations.
COMPLEX(SPC) :: z
REAL(SP) :: c1=0.5_sp,c2

n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in realft_sp')

if (present(zdata)) then
   ndum=assert_eq(n/2,size(zdata),'realft_sp')
cdata=>zdata
   Use zdata as cdata.
   if (isign == 1) cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
else
   allocate(cdata(n/2))
   Have to allocate storage ourselves.
   cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
end if

if (isign == 1) then
   c2=-0.5_sp
   call four1(cdata,+1)
The forward transform is here.
else
   c2=0.5_sp
   Otherwise set up for an inverse transform.
end if

w=zroots_unity(sign(n,isign),n/4)
w=cmplx(-aimag(w),real(w),kind=spc)
h1=c1*(cdata(2:nq)+conjg(cdata(nh:nq+2:-1)))
The two separate transforms are separated out of cdata.
h2=c2*(cdata(2:nq)-conjg(cdata(nh:nq+2:-1)))
Next they are recombined to form the true transform of the original real data:
cdata(2:nq)=h1*w(2:nq)+h2

z=cdata(1)
if (isign == 1) then
   Squeeze the first and last data together to get them all within the
   cdata(1)=cmplx(real(z)+aimag(z),real(z)-aimag(z),kind=spc) original array.
else
   cdata(1)=cmplx(c1*(real(z)+aimag(z)),c1*(real(z)-aimag(z)),kind=spc)
call four1(cdata,-1)
end if

if (present(zdata)) then
   if (isign /= 1) then
       Ship out answer in data if required.
data(1:n-1:2)=real(cdata)
data(2:n:2)=aimag(cdata)
   end if
else
   data(1:n-1:2)=real(cdata)
data(2:n:2)=aimag(cdata)
deallocate(cdata)
end if
SUBROUTINE realft_dp(data,isign,zdata)
USE nrtype; USE nrutil, ONLY : assert,assert_eq,zroots_unity
USE nr, ONLY : four1
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(DPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
INTEGER(I4B) :: n,ndum,nh,nq
COMPLEX(DPC), DIMENSION(size(data)/4) :: u
COMPLEX(DPC), DIMENSION(size(data)/4-1) :: h1,h2
COMPLEX(DPC), DIMENSION(:), POINTER :: cdata
COMPLEX(DPC) :: z
REAL(DP) :: c1=0.5_dp,c2
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in realft_dp')
nh=n/2
nq=n/4
if (present(zdata)) then
  ndum=assert_eq(n/2,size(zdata),'realft_dp')
cdata=>zdata
  if (isign == 1) cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
  else
    allocate(cdata(n/2))
    cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
  end if
  if (isign == 1) then
    c2=-0.5_dp
    call four1(cdata,+1)
  else
    c2=0.5_dp
    end if
w=zroots_unity(sign(n,isign),n/4)
w=cmplx(-aimag(w),real(w),kind=dpc)
h1=c1*(cdata(2:nq)+conjg(cdata(nh:nq+2:-1)))
h2=c2*(cdata(2:nq)-conjg(cdata(nh:nq+2:-1)))
cdata(2:nq)=h1+w(2:nq)*h2
cdata(nh:nq+2:-1)=conjg(h1-w(2:nq)*h2)
z=cdata(1)
if (isign == 1) then
  cdata(1)=cmplx(real(z)+aimag(z),real(z)-aimag(z),kind=dpc)
else
  cdata(1)=cmplx(c1*(real(z)+aimag(z)),c1*(real(z)-aimag(z)),kind=dpc)
  call four1(cdata,-1)
end if
if (present(zdata)) then
  if (isign /= 1) then
    data(1:n-1:2)=real(cdata)
    data(2:n:2)=aimag(cdata)
  else
    data(1:n-1:2)=real(cdata)
    data(2:n:2)=aimag(cdata)
  deallocate(cdata)
end if
END SUBROUTINE realft_dp

SUBROUTINE sinft(y)
USE nrtype; USE nrutil, ONLY : assert,cumsum,zroots_unity
USE nr, ONLY : realft
IMPLICIT NONE

REAL(SP), DIMENSION (:), INTENT (INOUT) :: y

Calculates the sine transform of a set of \( N \) real-valued data points stored in array \( y \). The number \( N \) must be a power of 2. On exit \( y \) is replaced by its transform. This program, without changes, also calculates the inverse sine transform, but in this case the output array should be multiplied by \( \frac{2}{N} \).

REAL(SP), DIMENSION(size(y)/2+1) :: wi
REAL(SP), DIMENSION(size(y)/2) :: y1,y2
INTEGER(I4B) :: n,nh

n=size(y)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in sinft')

nh=n/2
wi=aimag(zroots_unity(n+n,nh+1))

y(1)=0.0

y1=wi*(y(2:nh+1)+y(n:nh+1:-1))

Construct the two pieces of the auxiliary array.

y2=0.5_sp*(y(2:nh+1)-y(n:nh+1:-1))

Put them together to make the auxiliary array.

y(2:nh+1)=y1+y2
y(n:nh+1:-1)=y1-y2
call realft(y(1:n),+1)

Transform the auxiliary array.

y(1:nh+1)=0.5_sp*y(1)

Initialize the sum used for odd terms.

y(2:nh+1)=y1+aimag(wi)

Calculate the sine for the auxiliary array.

END SUBROUTINE sinft

SUBROUTINE cosft1(y)
USE nrtype; USE nrutil, ONLY : assert,cumsum,zroots_unity
USE nr, ONLY : realft
IMPLICIT NONE

REAL(SP), DIMENSION (:), INTENT (INOUT) :: y

Calculates the cosine transform of a set of \( N+1 \) real-valued data points \( y \). The transformed data replace the original data in array \( y \). \( N \) must be a power of 2. This program, without changes, also calculates the inverse cosine transform, but in this case the output array should be multiplied by \( \frac{2}{N} \).

COMPLEX(SPC), DIMENSION((size(y)-1)/2) :: w
REAL(SP), DIMENSION((size(y)-1)/2-1) :: y1,y2
REAL(SP) :: summ
INTEGER(I4B) :: n,nh

n=size(y)-1
call assert(iand(n,n-1)==0, 'n must be a power of 2 in cosft1')

nh=n/2
w=zroots_unity(n,nh)

y(1)=0.5_sp*(y(1)-y(n+1))

y1=0.5_sp*y(2:nh+1)+y(n:nh+2:-1)

y2=0:2:nh-2:-1

y(2)=0.0

y1=cumsum(y(1:n-1:2))

Odd terms are determined by this running sum.

y(1:n-1:2)=y(2:n:2)

Even terms in the transform are determined directly.

y(2:n:2)=y1
call realft(y(1:n),1)

Calculate the transform of the auxiliary function.

y(n+1)=y(2)

y(2)=summ

summ is the value of \( F_1 \) in equation (12.3.21).

Equation (12.3.20).

END SUBROUTINE cosft1
SUBROUTINE cosft2(y,isign)
USE nrtype; USE nrutil, ONLY : assert,cumsum,zroots_unity
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
INTEGER(I4B), INTENT(IN) :: isign

Calculates the "staggered" cosine transform of a set of \( N \) real-valued data points \( y \). The transformed data replace the original data in array \( y \). \( N \) must be a power of 2. Set \( \text{isign} \) to +1 for a transform, and to -1 for an inverse transform. For an inverse transform, the output array should be multiplied by \( 2/N \).

COMPLEX(SPC), DIMENSION(size(y)) :: w
REAL(SP), DIMENSION(size(y)/2) :: y1,y2
REAL(SP) :: ytemp
INTEGER(I4B) :: n,nh
n=size(y)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in cosft2')

w=zroots_unity(4*n,n)
if (isign == 1) then
Forward transform.
y1=0.5_sp*(y(1:nh)+y(n:nh+1:-1))
y2=aimag(w(2:n:2))*(y(1:nh)-y(n:nh+1:-1))
y(1:nh)=y1+y2
y(n:nh+1:-1)=y1-y2
call realft(y,1)
y1(1:nh-1)=y(3:n-1:2)*real(w(3:n-1:2))
-4(y(4:n:2)*aimag(w(3:n-1:2))
y2(1:nh-1)=y(4:n:2)*real(w(3:n-1:2))
+y(3:n-1:2)*aimag(w(3:n-1:2))
y(3:n-1:2)=y1(1:nh-1)
y(4:n:2)=y2(1:nh-1)
ytemp=0.5_sp*y(2)
Initializerecurrence foroddterms with \( \frac{1}{2}R_{N/2} \).
y(n-2:2:-2)=cumsum(y(n:4:-2),ytemp)
Recurrence for odd terms.
y(n)=ytemp
else if (isign == -1) then
Inverse transform.
ytemp=y(n)
y(4:n:2)=y(2:n-2:2)-y(4:n:2)
Form difference of odd terms.
y2(2)=2.0_sp*ytemp
y1(1:nh-1)=y(3:n-1:2)*real(w(3:n-1:2))
+y(4:n:2)*aimag(w(3:n-1:2))
y2(1:nh-1)=y(4:n:2)*real(w(3:n-1:2))
-y(3:n-1:2)*aimag(w(3:n-1:2))
y(3:n-1:2)=y(1:nh-1)
y(4:n:2)=y2(1:nh-1)
call realft(y,-1)
y1=y(1:nh)+y(n:nh+1:-1)
Invert auxiliary array.
y2=(0.5_sp/aimag(w(2:n:2)))*(y(1:nh)-y(n:nh+1:-1))
y(1:nh)=0.5_sp*(y1+y2)
y(n:nh+1:-1)=0.5_sp*(y1-y2)
end if
END SUBROUTINE cosft2

SUBROUTINE four3(data,isign)
USE nrtype
USE nr, ONLY : fourrow_3d
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:,), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
Replaces a 3-d complex array \( data \) by its discrete 3-d Fourier transform, if \( \text{isign} \) is input as 1, or replaces \( data \) by its inverse 3-d discrete Fourier transform times the product of its
three sizes, if isign is input as $-1$. All three of data's sizes must be integer powers of 2 (this is checked for in fourrow_3d). Parallelism is by use of fourrow_3d.

The reshape intrinsic, used with an order= parameter, is the multidimensional generalization of the two-dimensional transpose operation. The line

\[ \text{dat2} = \text{reshape}(\text{data},\text{shape} = \text{shape} (\text{dat2}),\text{order} = (/3,1,2/)) \]

is equivalent to the do-loop

\[
\text{do } j=1,\text{size(data,1)} \\
\text{dat2}(\cdot,\cdot,j) = \text{data}(j,\cdot,\cdot) \\
\text{end do}
\]

Incidentally, we have found some Fortran 90 compilers that (for scalar machines) are significantly slower executing the reshape than executing the equivalent do-loop. This, of course, shouldn't happen, since the reshape basically is an implicit do-loop. If you find such inefficient behavior on your compiler, you should report it as a bug to your compiler vendor! (Only thus will Fortran 90 compilers be brought to mature states of efficiency.)
Note that \( \texttt{four3} \) uses \( \texttt{fourrow}_\text{3d} \), the three-dimensional counterpart of \( \texttt{fourrow} \), while \( \texttt{four3}_\text{alt} \) uses \( \texttt{fourcol}_\text{3d} \), the three-dimensional counterpart of \( \texttt{fourcol} \). You may want to time these programs to see which is faster on your machine.

\[\star \star \star\]

In Volume 1, a single routine named \( \texttt{rlft3} \) was able to serve both as a three-dimensional real FFT, and as a two-dimensional real FFT. The trick is that the Fortran 77 version doesn’t care whether the input array data is dimensioned as two- or three-dimensional. Fortran 90 is not so indifferent, and better programming practice is to have two separate versions of the algorithm:

```fortran
SUBROUTINE rlft2(data,spec,speq,isign)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : four2
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: data
COMPLEX(SPC), DIMENSION(:,,:), INTENT(INOUT) :: spec
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: speq
INTEGER(I4B), INTENT(IN) :: isign
Given a two-dimensional real array \( \text{data}(1:\text{M},1:\text{N}) \), this routine returns (for \( \text{isign}=1 \)) the complex fast Fourier transform as two complex arrays: On output, \( \text{spec}(1:\text{M}/2,1:\text{N}) \) contains the zero and positive frequency values of the first frequency component, while \( \text{speq}(1:\text{N}) \) contains the Nyquist critical frequency values of the first frequency component. The second frequency components are stored for zero, positive, and negative frequencies, in standard wrap-around order. For \( \text{isign}=-1 \), the inverse transform (times \( \text{M} \times \text{N}/2 \) as a constant multiplicative factor) is performed, with output \( \text{data} \) deriving from input \( \text{spec} \) and \( \text{speq} \). For inverse transforms on data not generated first by a forward transform, make sure the complex input data array satisfies property (12.5.2). The size of all arrays must always be integer powers of 2.

INTEGER :: i1,j1,nn1,nn2
REAL(DP) :: theta
COMPLEX(SPC) :: c1=(0.5_sp,0.0_sp),c2,h1,h2,w
COMPLEX(SPC), DIMENSION(size(data,2)-1) :: h1a,h2a
COMPLEX(DPC) :: ww,wp
nn1=assert_eq(size(data,1),2*size(spec,1),'rlft2: nn1')
nn2=assert_eq(size(data,2),size(spec,2),size(speq),'rlft2: nn2')
call assert(iand((/nn1,nn2/),(/nn1,nn2/)-1)==0, & 'dimensions must be powers of 2 in rlft2')
c2=cmplx(0.0_sp,-0.5_sp*isign,kind=spc)
theta=TWOPI_D/(isign*nn1)
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=spc)
if (isign == 1) then
  Case of forward transform.
  spec(:,)=cmplx(data(1:nn1:2,:),data(2:nn1:2,:),kind=spc)
call four2(spec,isign)
  Here is where most all of the compute time
  speq=spec(1,:)
  is spent.
end if
h1=c1*(spec(1,1)+conjg(speq(1)))
h1a=c1*(spec(1,2:nn2)+conjg(speq(nn2:2:-1)))
h2=c2*(spec(1,1)-conjg(speq(1)))
h2a=c2*(spec(1,2:nn2)-conjg(speq(nn2:2:-1)))
spec(1,1)=h1+h2
spec(1,2:nn2)=h1a+h2a
spec(1,nn2:2:-1)=conjg(h1a+h2a)
ww=cmplx(1.0_dp,0.0_dp,kind=dpc)
do i1=2,nn1/4+1
  j1=nn1/2-i1+2
  w=ww*wp
  h1=c1*(spec(i1,1)+conjg(speq(j1,1)))
  h1a=c1*(spec(i1,2:nn2)+conjg(speq(j1,nn2:2:-1)))
  Initialize trigonometric recurrence.
do j1=1,nn1/2-1+n+1
  w=ww*wp
  h1=c1*(spec(i1,1)+conjg(speq(j1,1)))
  h1a=c1*(spec(i1,2:nn2)+conjg(speq(j1,nn2:2:-1)))
  Do the trig recurrence.
do i1=2,nn1/4+1
  Corresponding negative frequency.
excall assert_eq(size(data,1),2*size(spec,1),'rlft2: nn1')
excall assert_eq(size(data,2),size(spec,2),size(speq),'rlft2: nn2')
excall assert(iand((/nn1,nn2/),(/nn1,nn2/)-1)==0, &
  'dimensions must be powers of 2 in rlft2')
exc2=cmplx(0.0_sp,-0.5_sp*isign,kind=spc)
extheta=TWOPI_D/(isign*nn1)
exwp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=spc)
exif (isign == 1) then
  Case of forward transform.
exspec(:,)=cmplx(data(1:nn1:2,:),data(2:nn1:2,:),kind=spc)
excall four2(spec,isign)
exspeq=spec(1,:)
exelse
  speq=spec(1,:)
exendif
```

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).
h2*c2*(spec(i1,1)-conjg(spec(j1,1)))
h2a=c2*(spec(i1,2:nn2)-conjg(spec(j1,nn2:2:-1)))
spec(i1,1)=h1+w*h2
spec(i1,2:nn2)=h1a+w*h2a
spec(j1,1)=conjg(h1-w*h2)
spec(j1,nn2:2:-1)=conjg(h1a-w*h2a)
end do
if (isign == -1) then  
      Case of reverse transform.
call four2(spec,isign)
data(1:nn1:2,:)=real(spec)
data(2:nn1:2,:)=imag(spec)
end if

SUBROUTINE rfft3(data,spec,speq,isign)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : four3
REAL(SP), DIMENSION(:, :, :) , INTENT(INOUT) :: data
COMPLEX(SPC), DIMENSION(:, :, :) , INTENT(INOUT) :: spec
COMPLEX(SPC), DIMENSION(:, :) , INTENT(INOUT) :: speq
INTEGER(I4B), INTENT(IN) :: isign
      Given a three-dimensional real array data(1:L,1:M,1:N), this routine returns (for
      isign=1) the complex Fourier transform as two complex arrays: On output, the zero
      and positive frequency values of the first frequency component are in spec(1:L/2,1:M,1:N),
      while spec(1:M,1:N) contains the Nyquist critical frequency values of the first frequency
      component. The second and third frequency components are stored for zero, positive, and
      negative frequencies, in standard wrap-around order. For isign=-1, the inverse transform
      (times L × M × N/2 as a constant multiplicative factor) is performed, with output data
      deriving from input spec and speq. For inverse transforms on data not generated first by a
      forward transform, make sure the complex input data array satisfies property (12.5.2). The
      size of all arrays must always be integer powers of 2.
      INTEGER :: i1,i3,j1,j3,nn1,nn2,nn3
      REAL(DP) :: theta
      COMPLEX(SPC) :: c1=(0.5_sp,0.0_sp),c2,h1,h2,w
      COMPLEX(DPC) :: ww,wp
c2=cmplx(0.0_sp,-0.5_sp*isign,kind=spc)
nn1=assert_eq(size(data,1),2*size(spec,1),'rlft2: nn1')
nn2=assert_eq(size(data,2),size(spec,2),size(speq,1),'rlft2: nn2')
nn3=assert_eq(size(data,3),size(spec,3),size(speq,2),'rlft2: nn3')
call assert((/nn1,nn2,nn3/),(/nn1,nn2,nn3/)-1)==0, &
      'dimensions must be powers of 2 in rfft3'
theta=TWOPI_D/(isign*nn1)
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
if (isign == 1) then  
      Case of forward transform.
spec(:, :, :)=cmplx(data(:, :, :),data(:, :, :),kind=spc)
call four3(spec,isign)
      Here is where most all of the compute time
      is spent.
end if
do i3=1,nn3
  if (i3 /= 1) j3=nn3-i3+2
  h1=c1*(spec(1,1,i3)+conjg(speq(1,j3)))
h2=c2*(spec(1,2:nn2,i3)+conjg(speq(nn2:2:-1,j3)))
h2a=c2*(spec(1,2:nn2,i3)-conjg(speq(nn2:2:-1,j3)))
spec(1,i3)=h1+w*h2
spec(1,2:nn2,i3)=h1a+w*h2a
end do
**Chapter B12. Fast Fourier Transform**

```fortran
spq(1,j3)=conjg(h1-h2)
spq(nn2:2:-1,j3)=conjg(h1a-h2a)
ww=cmplx(1.0_dp,0.0_dp,kind=dpc)  ! Initialize trigonometric recurrence.
do i1=2,nn1/4+1
  j1=nn1/2-i1+2  ! Corresponding negative frequency.
  uww=ww*wp+ww  ! Do the trig recurrence.
  w=ww
  h1=c1*(spec(1,1,i3)+conjg(spec(1,1,j3)))  ! Equation (12.3.5).
  h1a=c1*(spec(1,2:nn2,i3)+conjg(spec(1,nn2:2:-1,j3)))
  h2=c2*(spec(1,1,i3)-conjg(spec(1,1,j3)))
  h2a=c2*(spec(1,2:nn2,i3)-conjg(spec(1,nn2:2:-1,j3)))
  spec(1,1,i3)=h1+w*h2
  spec(1,2:nn2,i3)=h1a+w*h2a
  spec(1,1,nn2:2:-1,j3)=conjg(h1-w*h2)
  spec(1,2:nn2,nn2:2:-1,j3)=conjg(h1a-w*h2a)
end do
if (isign == -1) then  ! Case of reverse transform.
call four3(spec,isign)
data(1:nn1:2,:,:)=real(spec)
data(2:nn1:2,:,:)=aimag(spec)
end if
END SUBROUTINE rlft3
```

Referring back to the discussion of parallelism, §22.4, that led to `four1`'s implementation with \(\sqrt{N}\) parallelism, you might wonder whether Fortran 90 provides sufficiently powerful high-level constructs to enable an FFT routine with \(N\)-fold parallelism. The answer is, “It does, but you wouldn’t want to use them!” Access to arbitrary interprocessor communication in Fortran 90 is through the mechanism of the “vector subscript” (one-dimensional array of indices in arbitrary order). When a vector subscript is on the right-hand side of an assignment statement, the operation performed is effectively a “gather”; when it is on the left-hand side, the operation is effectively a “scatter.”

It is quite possible to write the classic FFT algorithm in terms of gather and scatter operations. In fact, we do so now. The problem is efficiency: The computations involved in constructing the vector subscripts for the scatter/gather operations, and the actual scatter/gather operations themselves, tend to swamp the underlying very lean FFT algorithm. The result is very slow, though theoretically perfectly parallelizable, code. Since small-scale parallel (SSP) machines can saturate their processors with \(\sqrt{N}\) parallelism, while massively multiprocessor (MMP) machines inevitably come with architecture-optimized FFT library calls, there is really no niche for these routines, except as pedagogical demonstrations. We give here a one-dimensional routine, and also an arbitrary-dimensional routine modeled on Volume 1’s `fourn`.

Note the complete absence of do-loops of size \(N\); the loops that remain are over \(\log N\) stages, or over the number of dimensions.

SUBROUTINE four1_gather(data,isign)
USE nrtype; USE nrutil, ONLY : arth, assert
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,,:), INTENT(INOUT) :: data
INTEGER(14B), INTENT(IN) :: isign

Replaces a complex array data by its discrete Fourier transform, if isign is input as 1; or replaces data by `size(data)` times its inverse discrete Fourier transform, if isign is input as -1. The data must be an integer power of 2. This routine demonstrates coding the FFT algorithm in high-level Fortran 90 constructs. Generally the result is very efficient, though not as fast or as well-optimized as a hand-coded implementation (the latter is preferable for most applications).
much slower than library routines coded for specific architectures, and also significantly slower than the parallelization-by-rows method used in the routine four1.

```
INTEGER(I4B) :: n,n2,m,mm
REAL(DP) :: theta
COMPLEX(SPC) :: wp
INTEGER(I4B), DIMENSION(size(data)) :: jarr
INTEGER(I4B), DIMENSION(:,), ALLOCATABLE :: jrev
COMPLEX(SPC), DIMENSION(:,), ALLOCATABLE :: wtab,dtemp
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_gather')
allocate(jrev(n))
jarr=arth(0,1,n)
jrev=0
n2=n/2
m=n2
do
  where (iand(jarr,1) /= 0) jrev=jrev+m
  narr=jarr/2
  m=m/2
  if (m == 0) exit
end do
data=data(jrev+1)
deallocate(jrev)
allocate(dtemp(n),wtab(n2))
jarr=arth(0,1,n)
m=1
mm=n2
wtab(1)=(1.0_sp,0.0_sp)
seed the roots-of-unity table.
do
  outer loop executed \log_2 N times.
  where (iand(jarr,m) /= 0)
    The basic idea is to address the correct root-of-unity for each Danielson-Lanczos multiplication by tricky bit manipulations.
    data*data*ww(mm*land(jarr,m-1)+1)
  else where
    data=data+eoshift(dtemp,m)
  end where
  if (m >= n) exit
  mm=mm/2
  theta=PI_D/(isign*m)
  wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2, sin(theta),kind=spc)
  add entries to the table for the next iteration.
  wtab(mm+1:n2:2*mm)=wtab(1:n2-mm:2*mm)*wp+wtab(1:n2-mm:2*mm)
end do
deallocate(dtemp,wtab)
END SUBROUTINE four1_gather
```

SUBROUTINE fourn_gather(data,nn,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,), INTENT(INOUT) :: data
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: nn
INTEGER(I4B), INTENT(IN) :: isign

For data a one-dimensional complex array containing the values (in Fortran normal ordering) of an $M$-dimensional complex array, this routine replaces data by its $M$-dimensional discrete Fourier transform, if isign is input as 1. nn(1:M) is an integer array containing the lengths of each dimension (number of complex values), each of which must be a power of 2. If isign is input as $-1$, data is replaced by its inverse transform times the product of the length of all dimensions. This routine demonstrates coding the multidimensional FFT algorithm in high-level Fortran 90 constructs. Generally the result is very much slower than
library routines coded for specific architectures, and significantly slower than routines four2 and four3 for the two- and three-dimensional cases.

```fortran
INTEGER(I4B), DIMENSION(:,), ALLOCATABLE :: jarr
INTEGER(I4B) :: ndim, idim, ntot, nprev, n, n2, msk0, msk1, msk2, m, mm, mn
REAL(DP) :: theta
COMPLEX(SPC) :: wp
COMPLEX(SPC), DIMENSION(:,), ALLOCATABLE :: wtab, dtemp
call assert(iand(nn,nn-1)==0, &
   'each dimension must be a power of 2 in fourn_gather')
ndim=size(nn)
tot=product(nn)
nprev=1
allocate(jarr(ntot))
do idim=1, ndim
   Loop over the dimensions.
   jarr=arth(0,1,ntot)
   We begin the bit-reversal section of the routine.
   n=nn(idim)
n2=n/2
   msk0=nprev
   msk1=nprev*n2
   msk2=msk0+msk1
   do
      Construct an array of pointers from an index to its bit-reverse.
      if (msk1 <= msk0) exit
      where (iand(jarr,msk0) == 0 .neqv. iand(jarr,msk1) == 0) &
               jarr=ieor(jarr,msk2)
      msk0=msk0*2
      msk1=msk1/2
      msk2=msk0+msk1
   end do
   data=data(jarr+1)
   Move all data to bit-reversed location by a single gather/scatter.
   allocate(dtemp(ntot), wtab(n2))
   We begin the Danielson-Lanczos section of the routine.
   jarr=iand(n-1, arth(0,1,ntot)/nprev)
   m=1
   mm=n2
   mn=m*nprev
   wtab(1)=(1.0_sp,0.0_sp)
   Seed the roots-of-unity table.
   do
      This loop executed \( \log_2 N \) times.
      if (mm==0) exit
      where (iand(jarr,m) /= 0)
         The basic idea is to address the correct root-of-unity for each Danielson-Lanczos multiplication by tricky bit manipulations.
         dtemp=data*wtab(mm+iand(jarr,m-1)+1)
         data=eoshift(data,-mn)-dtemp
         This is half of Danielson-Lanczos.
      else where
         data=data+eoshift(dtemp,mn)
         This is the other half. The referenced elements of dtemp will have been set in the where clause.
      end where
      m=m*2
   end do
   theta=PI_D/isign
   Ready for trigonometry?
   wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
   Add entries to the table for the next iteration.
   wtab(mm+1:n2:mm)=wtab(1:n2-mm:mm)*wp &
                  +wtab(1:n2-mm:mm)
   end do
   deallocate(dtemp,wtab)
   nprev=n*nprev
end do
deallocate(jarr)
END SUBROUTINE fourn_gather
```

call assert(iand(nn,nn-1)==0 ... Once again the vector version of assert is used to test all the dimensions stored in nn simultaneously.
FUNCTION conolv(data, respns, isign)
USE nrtype; USE nrutil, ONLY : assert, nrerror
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
REAL(SP), DIMENSION(:), INTENT(IN) :: respns
INTEGER(I4B), INTENT(IN) :: isign
REAL(SP), DIMENSION(size(data)) :: conolv
Convolves or deconvolves a real data set data (of length N, including any user-supplied zero padding) with a response function respns, stored in wrap-around order in a real array of length \( M \leq N \). \( M \) should be an odd integer, \( N \) a power of 2. Wrap-around order means that the first half of the array respns contains the impulse response function at positive times, while the second half of the array contains the impulse response function at negative times, counting down from the highest element \( \text{respns}(M) \). On input isign is +1 for convolution, -1 for deconvolution. The answer is returned as the function conolv, an array of length \( N \). data has INTENT(INOUT) for consistency with realft, but is actually unchanged.

INTEGER(I4B) :: no2, n, m
COMPLEX(SPC), DIMENSION(size(data)/2) :: tmpd, tmpr
n = size(data)
m = size(respns)
call assert(iand(n, n-1) == 0, 'n must be a power of 2 in conolv')
call assert(mod(m,2) == 1, 'm must be odd in conolv')
conolv(1:m) = respns(:)
Put respns in array of length \( n \).
conolv(n-(m-3)/2:n) = conolv((m+3)/2:m)
Pad with zeros.
no2 = m/2
call realft(data, 1, tmpd)
FFT both arrays.
call realft(conolv, 1, tmpr)
if (isign == 1) then
Multiply FFTs to convolve.
tmpr(1) = cmplx(real(tmpd(1))/no2, &
aimag(tmpd(1))/no2, kind=spc)
tmpr(2:) = tmpd(2:)/no2
else if (isign == -1) then
Divide FFTs to deconvolve.
if (any(abs(tmpr(2:)) == 0.0) .or. real(tmpr(1)) == 0.0 &
or. aimag(tmpr(1)) == 0.0) call nrerror &
(‘deconvolving at response zero in conolv’)
tmpr(1) = cmplx(real(tmpr(1))/no2, &
aimag(tmpr(1))/no2, kind=spc)
tmpr(2:) = tmpr(2:)/no2
else
call nrerror(’no meaning for isign in conolv’)end if
call realft(conolv, -1, tmpr) Inverse transform back to time domain.
END FUNCTION conolv
The intrinsic function `cmplx` returns a quantity of type default complex unless the kind argument is present. It is therefore a good idea always to include this argument. The intrinsic functions `real` and `aimag`, on the other hand, when called with a complex argument, return the same kind as their argument. So it is a good idea not to put in a kind argument for these. (In fact, `aimag` doesn’t allow one.) Don’t confuse these situations, regarding complex variables, with the completely unrelated use of `real` to convert a real or integer variable to a real value of specified kind. In this latter case, kind should be specified.

```fortran
FUNCTION correl(data1, data2)
USE nrtype; USE nrutil, ONLY : assert, assert_eq
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data1, data2
REAL(SP), DIMENSION(size(data1)) :: correl

Computes the correlation of two real data sets `data1` and `data2` of length `N` (including any user-supplied zero padding). `N` must be an integer power of 2. The answer is returned as the function `correl`, an array of length `N`. The answer is stored in wrap-around order, i.e., correlations at increasingly negative lags are in `correl(1)` on down to `correl(N/2 + 1)`, while correlations at increasingly positive lags are in `correl(N/2)` on up to `correl(N/2 - 1)`. Sign convention of this routine: if `data1` lags `data2`, i.e., is shifted to the right of it, then `correl` will show a peak at positive lags.

COMPLEX(SPC), DIMENSION(size(data1)/2) :: cdat1, cdat2
INTEGER(I4B) :: no2, n

Normalization for inverse FFT.

```
no2 = n/2
```

```
call realft(data1, 1, cdat1)
```

Transform both data vectors.

```
call realft(data2, 1, cdat2)
```

```
cdat1(1) = cmplx(real(cdat1(1))*real(cdat2(1))/no2, &
                aimag(cdat1(1))*aimag(cdat2(1))/no2, kind=spc)
```

Multiply to find FFT of their correlation.

```
cdat1(2:)=cdat1(2:)*conjg(cdat2(2:))/no2
```

Inversetransform givescorrelation.

END FUNCTION correl
```

See just above for why we use the explicit kind type parameter `spc` for `cmplx`, but omit `sp` for `real`.

```fortran
SUBROUTINE spectr(p, k, ovrlap, unit, n_window)
USE nrtype; USE nrutil, ONLY : arth, nrerror
USE nr, ONLY : four1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: p
INTEGER(I4B), INTENT(IN) :: k
LOGICAL(LGT), INTENT(IN) :: ovrlap
LOGICAL(LGT), OPTIONAL, INTENT(IN) :: n_window, unit

Reads data from input unit 9, or if the optional argument unit is present, from that input unit. The output is an array `p` of length `M` that contains the data’s power (mean square amplitude) at frequency 

\[(j-1)/2M \text{ cycles per grid point, for } j = 1, 2, \ldots, M, \text{ based on } (2k+1)\times M \text{ data points if ovrlap is set .true. } \text{ or } 4k\times M \text{ data points if ovrlap is set .false.}. \]

The number of segments of the data is `2*k` in both cases. The routine calls `four1` `k` times, each call with 2 partitions each of `2*M` real data points. If the optional argument `n_window` is present, the routine uses the Bartlett window, the square window,
or the Welch window for \( n_{\text{window}} = 1, 2, 3 \) respectively. If \( n_{\text{window}} \) is not present, the Bartlett window is used.

```fortran
INTEGER(I4B) :: j, joff, joffn, kk, m, m4, m43, m44, mm, iunit, nn_window
REAL(SP) :: den, facm, facp, sumw
REAL(SP), DIMENSION(2*size(p)) :: w
REAL(SP), DIMENSION(4*size(p)) :: w1
REAL(SP), DIMENSION(size(p)) :: w2
COMPLEX(SPC), DIMENSION(2*size(p)) :: cv1
m=size(p)
if (present(n_window)) then
  nn_window=n_window
else
  nn_window=1
end if
if (present(unit)) then
  iunit=unit
else
  iunit=9
end if
m4=m+m
m44=m4+4
m43=m4+3
den=0.0
facm=m
fapc=1.0_sp/m
w1(1:mm)=window(arth(1,1,mm),facm,fapc,nn_window)
sumw=dot_product(w1(1:mm),w1(1:mm))
p(:)=0.0
if (ovrlap) read (iunit,*) (w2(j),j=1,m)
do kk=1,k
  loop over data segments in groups of two.
do joff=-1,0,1
  get two complete segments into workspace.
do joff=1,1,1
    if (ovrlap) then
      w1(joff+2:joff+mm:2)=w2(1:m)
      read (iunit,*) (w2(j),j=1,m)
    end if
  end do
  apply the window to the data.
w1(2:mm+2)=w1(2:mm+2)*w
w1(1:mm+2)=w1(1:mm+2)*w
w1(1:mm)=cmplx(w1(1:mm),w1(2:mm),kind=sp)
call four1(cw1(1:mm),1)
end do
p(:)=p(:)+sumw
den=den+sumw
end do
p(:)=p(:)/(m4*den)
END
```

```
FUNCTION window(j,facm,fapc,nn_window)
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: j
INTEGER(I4B), INTENT(IN) :: nn_window
REAL(SP), INTENT(IN) :: facm,fapc
REAL(SP), DIMENSION(size(j)) :: window
select case(nn_window)
case(1)
  ...
```

Useful factors.

Factors used by the window function.

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. Permission is granted for legitimate users to make one paper copy of this chapter for their own personal use. Further reproduction of any copying of machine-readable files (including this one) to any server computer is strictly prohibited. To order Numerical Recipes books or CD-ROMs, visit our website:

http://www.nr.com or call 1-800-872-7423 (North America only) or send email to directcustserv@cambridge.org (outside North America).
window(j)=(1.0_sp-abs(((j-1)-facm)*facp)) Bartlett window.
case(2)  
  window(j)=1.0  Square window.
case(3)  
  window(j)=(1.0_sp-((j-1)-facm)*facp)**2) Welch window.
case default  
  call nrerror('unimplemented window function in spctrm')
end select
END FUNCTION window
END SUBROUTINE spctrm

The Fortran 90 optional argument feature allows us to make unit 9
the default output unit in this routine, but leave the user the option of
specifying a different output unit by supplying an actual argument for
unit. We also use an optional argument to allow the user the option of overriding
the default selection of the Bartlett window function.

FUNCTION window(j,facm,facp,nn_window)
In Fortran 77 we coded this as a
statement function. Here the internal function is equivalent, but allows full specifi-
cation of the interface and so is preferred.

SUBROUTINE memcof(data,xms,d)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: xms
REAL(SP), DIMENSION(:), INTENT(IN) :: data
REAL(SP), DIMENSION(:), INTENT(OUT) :: d

Given a real vector data of length N, this routine returns M linear prediction coefficients
in a vector d of length M, and returns the mean square discrepancy as xms.

INTEGER(I4B) :: k,m,n
REAL(SP) :: denom,pneum
REAL(SP), DIMENSION(size(data)) :: wk1,wk2,wktmp
REAL(SP), DIMENSION(size(d)) :: wkm
m=size(d)
n=size(data)
xms=dot_product(data,data)/n
wk1(1:n-1)=data(1:n-1)
wk2(1:n-1)=data(2:n)
do k=1,m
  pneurn=dot_product(wk1(1:n-k),wk2(1:n-k))
  denom=dot_product(wk1(1:n-k),wk1(1:n-k)) + &
    dot_product(wk2(1:n-k),wk2(1:n-k))
  d(k)=2.0_sp*pneum/denom
  xms=xms*(1.0_sp-d(k)**2)
  d(k:1:k-1)=wkm(1:k-1)-d(k)*wkm(k-1:1:-1)
end do

The algorithm is recursive, although it is implemented as an iteration. It builds up the
answer for larger and larger values of m until the desired value is reached. At this point
in the algorithm, one could return the vector d and scalar xms for a set of LP coefficients
with k (rather than m) terms.

if (k == m) RETURN

wkm(1:k)=d(1:k)
wktmp(2:n-k)=wk1(2:n-k)
wkm(1:n-k)=wk1(1:n-k)-wkm(1:n-k-1)*wk2(1:n-k-1)
wk2(2:n-k)=wk2(2:n-k)-wkm(1:n-k-1)*wktmp(2:n-k)

end do
call nrerror('never get here in memcof')
END SUBROUTINE memcof
SUBROUTINE fixrts(d)
USE nrtype
USE nr, ONLY : zroots
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: d
Given the LP coefficients d, this routine finds all roots of the characteristic polynomial (13.6.14), reflects any roots that are outside the unit circle back inside, and then returns a modified set of coefficients in d.
INTEGER(I4B) :: i,m
LOGICAL(LGT) :: polish
COMPLEX(SPC), DIMENSION(size(d)+1) :: a
COMPLEX(SPC), DIMENSION(size(d)) :: roots
m=size(d)
a(m+1)=cmplx(1.0_sp,kind=spc) Set up complex coefficients for polynomial root finder.
a(m:1:-1)=cmplx(-d(1:m),kind=spc)
polish=.true.
call zroots(a(1:m+1),roots,polish) Find all the roots.
where (abs(roots) > 1.0) roots=1.0_sp/conjg(roots) Reflect all roots outside the unit circle back inside.
a(1)=-roots(1) Now reconstruct the polynomial coefficients.
d(1:m+1)=cmplx(1.0_sp,kind=spc) 
do i=2,m
a(2:i)=a(1:i-1)-roots(i)*a(2:i) and synthetically multiplying.
a(1)=-roots(i)*a(1)
end do
d(m:1:-1)=-real(a(1:m)) The polynomial coefficients are guaranteed to be real, so we need only return the real part as new LP coefficients.
END SUBROUTINE fixrts

FUNCTION predic(data,d,nfut)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: data,d
INTEGER(I4B), INTENT(IN) :: nfut
REAL(SP), DIMENSION(nfut) :: predic
Given an array data, and given the data’s LP coefficients d in an array of length M, this routine applies equation (13.6.11) to predict the next nfut data points, which it returns in an array as the function value predic. Note that the routine references only the last M values of data, as initial values for the prediction.
INTEGER(I4B) :: j,ndata,m
REAL(SP) :: discrp,sm
REAL(SP), DIMENSION(size(d)) :: reg
m=size(d)
ndata=size(data)
reg(1:m)=data(ndata:ndata+m:-1)
do j=1,nfut
sm=discrp
discrp=discrp*dot_product(d,reg)
reg=eoshift(reg,-1,sm) [If you want to implement circular arrays, you can avoid this shifting of coefficients!]
predic(j)=sm
end do
END FUNCTION predic
FUNCTION evlmem(fdt,d,xms)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: fdt,xms
REAL(SP), DIMENSION(:), INTENT(IN) :: d
REAL(SP) :: evlmem

Given d and xms as returned by memcof, this function returns the power spectrum estimate
\( P(f) \) as a function of \( f \Delta \).

COMPLEX(SPC) :: z,zz
REAL(DP) :: theta

Trigonometric recurrences in double precision.

\[ \theta = 2\pi_f \Delta \]
\[ z = \cos(\theta) + i \sin(\theta) \]
\[ zz = 1 - z \cdot \text{poly}(z,d) \]
\[ \text{evlmem} = \frac{xms}{\text{abs}(zz)^2} \]  
Equation (13.7.4).

END FUNCTION evlmem

zz=...poly(z,d) The poly function in nrutil returns the value of the polynomial with coefficients \( d(:) \) at \( z \). Here a version that takes real coefficients and a complex argument is actually invoked, but all the different versions have been overloaded onto the same name poly.

***

SUBROUTINE period(x,y,ofac,hifac,px,py,jmax,prob)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,nrerror
USE nr, ONLY : avevar
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: jmax
REAL(SP), INTENT(IN) :: ofac,hifac
REAL(SP), INTENT(OUT) :: prob
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), DIMENSION(:), POINTER :: px,py

Input is a set of \( N \) data points with abscissas \( x \) (which need not be equally spaced) and ordinates \( y \), and a desired oversampling factor \( ofac \) (a typical value being 4 or larger). The routine returns pointers to internally allocated arrays \( px \) and \( py \). \( px \) is filled with an increasing sequence of frequencies (not angular frequencies) up to \( hifac \times \text{average} \) Nyquist frequency, and \( py \) is filled with the values of the Lomb normalized periodogram at those frequencies. The length of these arrays is \( 0.5 \cdot ofac \cdot hifac \cdot N \).

The arrays \( x \) and \( y \) are not altered. The routine also returns \( jmax \) such that \( py(jmax) \) is the maximum element in \( py \), and \( prob \), an estimate of the significance of that maximum against the hypothesis of random noise. A small value of \( prob \) indicates that a significant periodic signal is present.

INTEGER(I4B) :: i,n,nout
REAL(SP) :: ave,cwtau,effm,expy,pnow,sumc,sumcy,sums,sumsh,sumsy,swtau,var,wtau,xave,xdif,xmax,xmin
REAL(DP), DIMENSION(size(x)) :: tmp1,tmp2,wi,wpi,wpr,wr
LOGICAL(LGT), SAVE :: init=.true.

\[ n = \text{assert_eq}(\text{size}(x),\text{size}(y),'period') \]
if (init) then
init=.false.
nullify(px,py)
else
if (associated(px)) deallocate(px)
if (associated(py)) deallocate(py)
end if
nout=0.5_sp*ofac*hifac*n
allocate(px(nout),py(nout))
call avevary(:,x,ave)
call avevarx(y,ave,\)
allocate(px(nout),py(nout))
call avevary(:,ave,var)
if (var == 0) call nrerror('zero variance in period')

xmax=maxval(x(:))
xmin=minval(x(:))

Go through data to get the range of abscissas.
xdif = xmax - xmin
xave = 0.5 * (xmax + xmin)
pnow = 1.0 / (xdif * ofac)

Starting frequency.
tmp1(:, :) = TWOPI_D * ((x(:, :) - xave) * pnow)
wpr(:, :) = -2.0_dp * sin(0.5_dp * tmp1(:, :) + 2)
wp := cos(tmp1(:, :))
wi(:, :) = sin(tmp1(:, :))

Initialize values for the trigonometric recurrences at each data point. The
recurrences are done in double precision.

do i = 1, nout
    px(i) := pnow
    sumsh = dot_product(wi, wr)
    vtau = 0.5 * atan2(2.0 * sumsh, sumc)
    swtau = sin(vtau)
cwtau = cos(vtau)
    tmp1(:, :) = wi(:, :) * cwtau - wr(:, :) * swtau
    Then, loop over the data again to get the
    periodogram value.
    tmp2(:, :) = wr(:, :) * cwtau + wi(:, :) * swtau
    sums = dot_product(tmp1, tmp1)
    sumc = dot_product(tmp2, tmp2)
    sums = dot_product(y(:, :) - ave, tmp1)
    sumc = dot_product(y(:, :) - ave, tmp2)
    py(i) = 0.5 * (sumc ** 2 / sumc + sums ** 2 / sums) / var
    pnow = pnow + 1.0 / (ofac * xdif)

    The next frequency.
end do

jmax = imaxloc(py(1:nout))
expy = exp(-py(jmax)) Evaluate statistical significance of the maximum.
effm = 2.0 * nout / ofac
prob = effm * expy
if (prob > 0.01) prob = 1.0 - (1.0 - expy) ** effm
END SUBROUTINE period

This routine shows another example of how to return arrays whose size is
not known in advance (cf. zbrac in Chapter B9). The coding is explained
in the subsection on pointers in §21.5. The size of the output arrays,
nout in the code, is available as size(px).

jmax = imaxloc... See discussion of imaxloc on p. 1017.

SUBROUTINE fasper(x, y, ofac, hifac, px, py, jmax, prob)
USE nrtype; USE nrutil, ONLY : arth, assert_eq, imaxloc, nrerror
USE nr, ONLY : avevar, realft
IMPLICIT NONE
REAL(SP), DIMENSION(:, INTENT(IN)) :: x, y
REAL(SP), INTENT(IN) :: ofac, hifac
INTEGER(1B), INTENT(OUT) :: jmax
REAL(SP), INTENT(OUT) :: prob
REAL(SP), DIMENSION(:, ), POINTER :: px, py
INTEGER(1B), PARAMETER :: MACC=4

Input is a set of N data points with abscissas x (which need not be equally spaced) and
ordinates y, and a desired oversampling factor ofac (a typical value being 4 or larger).
The routine returns pointers to internally allocated arrays px and py. px is filled with
an increasing sequence of frequencies (not angular frequencies) up to hifac times the
"average" Nyquist frequency, and py is filled with the values of the Lomb normalized
periodogram at those frequencies. The length of these arrays is 0.5 * ofac * hifac * N.
The arrays x and y are not altered. The routine also returns jmax such that py(jmax) is
the maximum element in py, and prob, an estimate of the significance of that maximum
against the hypothesis of random noise. A small value of prob indicates that a significant
periodic signal is present.

Parameter: MACC is the number of interpolation points per 1/4 cycle of highest frequency.

```fortran
INTEGER(I4B) :: j,k,n,ndim,nfreq,nfreqt,nout
REAL(SP) :: ave,ck,ckk,cterm,cut,den,df,effm,expy,fac,fndim,hc2wt,&
    hs2wt,hypo,stem,swt,var,xdif,xmax,xmin
REAL(SP), DIMENSION(:), ALLOCATABLE :: wk1,wk2
LOGICAL(LGT), SAVE :: init=.true.
n=assert_eq(size(x),size(y),'fasper')
if (init) then
   init=.false.
   nullify(px,py)
else
   if (associated(px)) deallocate(px)
   if (associated(py)) deallocate(py)
end if
nfreqt=ofac*hifac*n*MACC
nfreq=64
do 
   if (nfreq >= nfreqt) exit
   nfreq=nfreq*2
end do
ndim=2*nfreq
allocate(wk1(ndim),wk2(ndim))
call avevar(y(1:n),ave,var)
if (var == 0.0) call nrerror('zero variance in fasper')
xmax=maxval(x(:))
xmin=minval(x(:))
xdif=xmax-xmin
wk1(1:ndim)=0.0
wk2(1:ndim)=0.0
fac=fndim/(xdif*ofac)
fndim=ndim
do j=1,n
   ck=1.0_sp+mod((x(j)-xmin)*fac,fndim)
   ckk=1.0_sp+mod(2.0_sp*(ck-1.0_sp),fndim)
   call spreadval(y(j)-ave,wk1,ck,MACC)
   call spreadval(1.0_sp,wk2,ckk,MACC)
end do
call realft(wk1(1:ndim),1)
call realft(wk2(1:ndim),1)
df=1.0_sp/(xdif*ofac)
nout=0.5_sp*ofac+hifac*n
allocate(px(nout),py(nout))
k=3
do j=1,nout
   hypo=sqrt(wk2(k)**2+wk2(k+1)**2)
   hc2wt=0.5_sp+wk2(k+hypo)
   hs2wt=0.5_sp+wk2(k+1+hypo)
   cwt=sqrt(0.5_sp+hc2wt)
   swt=sign(sqrt(0.5_sp-hc2wt),hs2wt)
   cterm=(cwt*wk1(k)+swt*wk1(k+1))**2/den
   stem=(cwt*wk1(k+1)-swt*wk1(k))**2/(n-den)
   px(j)=j*df
   py(j)=(cterm+stem)/(2.0_sp*var)
   k=k+2
end do
deallocate(wk1,wk2)
jmax=imaxloc(py(1:nout))
expy=exp(-py(jmax))
prob=effm*expy
if (prob > 0.01_sp) prob=1.0_sp-(1.0_sp-expy)**effm
CONTAINS
```
SUBROUTINE spreadval(y,yy,x,m)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: y,x
REAL(SP), DIMENSION(:), INTENT(INOUT) :: yy
INTEGER(4), INTENT(IN) :: m

Given an array yy of length N, extrapolate (spread) a value y into m actual array elements that best approximate the "fictional" (i.e., possibly noninteger) array element number x.

The weights used are coefficients of the Lagrange interpolating polynomial.

INTEGER(4) :: ihi,ilo,ix,j,nden,n
REAL(SP) :: fac
INTEGER(4), DIMENSION(10) :: nfac = (/ 1,1,2,6,24,120,720,5040,40320,362880 /)

if (m > 10) call nrerror('factorial table too small in spreadval')
n=size(yy)
ix=x
if (x == real(ix,sp)) then
    yy(ix)=yy(ix)+y
else
    ilo=min(max(int(x-0.5_sp*m+1.0_sp),1),n-m+1)
    ihi=ilo+m-1
    nden=nfac(m)
    fac=product(x-arith(ilo,1,m))
    yy(ihi)=yy(ihi)+y*fac/(nden*(x-ihi))
    do j=ihi-1,ilo,-1
        nden=(nden/(j+1-ilo))*(j-ihi)
        yy(j)=yy(j)+y*fac/(nden*(x-j))
    end do
end if
END SUBROUTINE spreadval

This routine shows another example of how to return arrays whose size is not known in advance (cf. zbbrac in Chapter B9). The coding is explained in the subsection on pointers in §21.5. The size of the output arrays, nout in the code, is available as size(px).

IMAXLOC... See discussion of imaxloc on p. 1017.

if (x == real(ix,sp)) then Without the explicit kind type parameter sp, real returns a value of type default real for an integer argument. This prevents automatic conversion of the routine from single to double precision. Here all you have to do is redefine sp in nrtype to get double precision.

* * *

SUBROUTINE dftcor(w,delta,a,b,endpts,corre,corim,corfac)
USE nrtype, USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: w,delta,a,b
REAL(SP), INTENT(OUT) :: corre,corim,corfac
REAL(SP), DIMENSION(:), INTENT(IN) :: endpts

For an integral approximated by a discrete Fourier transform, this routine computes the correction factor that multiplies the DFT and the endpoint correction to be added. Input is the angular frequency w, step size delta, lower and upper limits of the integral a and b, while the array endpts of length 8 contains the first 4 and last 4 function values. The
correction factor \( W(\theta) \) is returned as corfac, while the real and imaginary parts of the endpoint correction are returned as corre and corim.

```fortran
REAL(SP) :: a0i,a0r,a1i,a1r,a2i,a2r,a3i,a3r,arg,c1,cr,s1,ssr,t,
    t2,t4,t6
REAL(DP) :: cth,ctth,spth2,sth,sth4i,stth,th,th2,th4,&
    tmth2,tth4i
th=w*delta
call assert(a < b, th >= 0.0, th <= PI_D, 'dftcor args')
if (abs(th) < 5.0e-2_dp) then
  Use series.
  t=th
t2=t*t
t4=t2*t2
t6=t4*t2
corfac=1.0_sp-(11.0_sp/720.0_sp)*t4+(23.0_sp/15120.0_sp)*t6
  a0r=(-2.0_sp/3.0_sp)+t2/45.0_sp+(103.0_sp/15120.0_sp)*t4-&
    (169.0_sp/226800.0_sp)*t6
  a1r=(7.0_sp/24.0_sp)-(7.0_sp/180.0_sp)*t2+(5.0_sp/3456.0_sp)*t4-&
    (7.0_sp/259200.0_sp)*t6
  a2r=(-1.0_sp/6.0_sp)+t2/45.0_sp-(5.0_sp/6048.0_sp)*t4+64800.0_sp
  a3r=(1.0_sp/24.0_sp)-t2/180.0_sp+(5.0_sp/24192.0_sp)*t4-t6/259200.0_sp
  a0i=t*(2.0_sp/45.0_sp*(2.0_sp/105.0_sp)*t4-&
    (8.0_sp/2835.0_sp)*t4+(86.0_sp/467775.0_sp)*t6)
  a1i=t*(7.0_sp/72.0_sp-t2/168.0_sp+(11.0_sp/72576.0_sp)*t4-&
    (13.0_sp/5987520.0_sp)*t6)
  a2i=t*(7.0_sp/90.0_sp+t2/210.0_sp-(11.0_sp/90720.0_sp)*t4-&
    (13.0_sp/7484400.0_sp)*t6)
  a3i=t*(7.0_sp/360.0_sp+t2/840.0_sp+(11.0_sp/362880.0_sp)*t4-&
    (13.0_sp/29937600.0_sp)*t6)
else
  Use trigonometric formulas in double precision.
  cth=cos(th)
  sth=sin(th)
  ctth=cth**2-sth**2
  stth=2.0_dp*sth*cth
  th2=th*th
  th4=th2*th2
  tmth2=3.0_dp-th2
  spth2=6.0_dp+th2
  sth4i=1.0_sp/(6.0_dp*th4)
  tth4i=2.0_dp*stth4
  corfac=tth4i*spth2*(3.0_sp-4.0_dp*ctth+ctth)
  a0r=sth4i*(-42.0_dp+5.0_dp*th2+spth2*(8.0_dp*cth-ctth))
  a1r=sth4i*(14.0_dp*tmth2-7.0_dp*ctth)
  a2r=sth4i*(30.0_dp*th-5.0_dp*spth2*sth)
  a3r=sth4i*(2.0_dp*tmth2-spth2*cth)
end if
c1=s0*endpts(1)+a1r*endpts(2)+a2r*endpts(3)+a3r*endpts(4)
s1=s0*endpts(1)+a1i*endpts(2)+a2i*endpts(3)+a3i*endpts(4)
cr=s0*endpts(8)+a1r*endpts(7)+a2r*endpts(6)+a3r*endpts(5)
sr=-s0*endpts(8)-a1i*endpts(7)-a2i*endpts(6)-a3i*endpts(5)
arg=(a-b)/2
  c=cos(arg)
  s=sin(arg)
corre=c+cr-s+sr
  corr=el+c+cr+c+sr
END SUBROUTINE dftcor
```

SUBROUTINE dftint(func,a,b,w,cosint,sinint)
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : dftcor, polint, realft
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b,w
REAL(SP), INTENT(OUT) :: cosint,sinint
INTERFACE
FUNCTION func(x)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: M=64, NDFT=1024, MPOL=6
Examplesubroutineillustratinghowtousetheroutine

dftcor. The usersuppliesanexter-

nalfunction

cosint

t. The routine then returns

cosint

as

cosint

and

cosint

as

sinint.

Parameters: The values of

M,

NDFT,

and

MPOL

are merely illustrative and should be optim-

izedforyourparticularapplication.

M

isthenumberofsubintervals,

NDFT

isthelengthof

theFFT(a power of 2), and

MPOL

isthedegreeofpolynomialinterpolation usedtoobtain

desired frequency from the FFT.

INTEGER(I4B) :: nn
INTEGER(I4B), SAVE :: init=0
INTEGER(I4B), DIMENSION(MPOL) :: nnmpol
REAL(SP) :: c, cdft, cerr, corfac, corre, endpts, sdft, serr
REAL(SP), SAVE :: delta
REAL(SP), DIMENSION(MPOL) :: cpol, spol, xpol
REAL(SP), DIMENSION(NDFT), SAVE :: data
REAL(SP), DIMENSION(8), SAVE :: endpts
REAL(SP), SAVE :: aold=-1.0e30_sp,bold=-1.0e30_sp
if (init /= 1 .or. a /= aold .or. b /= bold) then
  Do we need to initialize, or
  is only ω changed?
  init=1
  aold=a
  bold=b
  delta=(b-a)/M
  data(1:M+1)=func(a+arth(0,1,M+1)*delta)
  Load the function values into the
  data array.
  data(M+2:NDFT)=0.0
  Zero pad the rest of the data array.
  endpts(1:4)=data(1:4)
  Load the endpoints.
  endpts(5:8)=data(M-2:M+1)
  call realft(data(1:NDFT),1)
  realft returns the unused value corresponding to
  ωN/2 in
  data(2). We actually want
  this element to contain the imaginary part corresponding to
  ω0, which is zero.
  data(2)=0.0
end if

Now interpolate on the DFT result for the desired frequency. If the frequency is an
ωn, i.e.,
the quantity
en
is an integer, then
cdft=data(2*en-1),
sdft=data(2*en),
and you could
omit the interpolation.

en=w*delta*NDFT/TWOPI+1.0_sp
nn=min(max(int(en-0.5_sp*MPOL+1.0_sp),1),NDFT/2-MPOL+1)
nmpol=arth(nn,1,MPOL)
cpol(1:MPOL)=data(2*nmpol(:)-1)
spol(1:MPOL)=data(2*nmpol(:))
xpol(1:MPOL)=nmpol(:)
call point(xpol,cpol,en,cdft,cerr)
call point(spol,spol,en,sdft,serr)
call dftcor(w,delta,a,b,endpts,corre,corim,corfac)
cdf=cdft+corfac*corre
sdft=sdft+corfac*corim
Finally multiply by ∆ and exp(iωa).
c=delta*cos(w*a)
s=delta*sin(w*a)
cosint=c*cdft-s*sdft
Chapter B13. Fourier and Spectral Applications

\[
\sin\text{int} = a \cdot \text{cdft} + c \cdot \text{sdf}\text{t}
\]

END SUBROUTINE dftint

⋆⋆⋆

SUBROUTINE wt1(a,isign,wtstep)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
INTERFACE
SUBROUTINE wtstep(a,isign)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE wtstep
END INTERFACE
One-dimensional discrete wavelet transform. This routine implements the pyramid algorithm, replacing \( a \) by its wavelet transform (for \( \text{isign}=1 \)), or performing the inverse operation (for \( \text{isign}=-1 \)). The length of \( a \) is \( N \), which must be an integer power of 2. The subroutine \text{wtstep}, whose actual name must be supplied in calling this routine, is the underlying wavelet filter. Examples of \text{wtstep} are \text{daub4} and (preceded by \text{pwtset}) \text{pwt}.

\[
\text{INTEGER(I4B)} :: n, nn
n = \text{size}(a)
call assert(\text{iand}(n,n-1)==0, 'n must be a power of 2 in wt1')
if (n < 4) RETURN
if (\text{isign} >= 0) then
    Wavelet transform.
    nn = n
    Start at largest hierarchy,
    do
        if (nn < 4) exit
        call \text{wtstep}(a(1:nn),isign)
        nn = nn/2
        and work towards smallest.
    end do
else
    Inverse wavelet transform.
    nn = 4
    Start at smallest hierarchy,
    do
        if (nn > n) exit
        call \text{wtstep}(a(1:nn),isign)
        nn = nn*2
        and work towards largest.
    end do
end if
END SUBROUTINE wt1

SUBROUTINE daub4(a,isign)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
Applies the Daubechies 4-coefficient wavelet filter to data vector \( a \) (for \( \text{isign}=1 \)) or applies its transpose (for \( \text{isign}=-1 \)). Used hierarchically by routines \text{wt1} and \text{wtn}.

\[
\begin{align*}
\text{REAL(SP), DIMENSION(size(a)) :: } & \text{wksp} \\
\text{REAL(SP), PARAMETER :: C0=0.4829629131445341_sp,} & \\
& C1=0.8365163037378097_sp, C2=0.2241438680420134_sp, & \\
& C3=0.1294095225512604_sp
\end{align*}
\]

\[
\text{INTEGER(I4B)} :: n, nh, nhp, nhm
n = \text{size}(a)
if (n < 4) RETURN
nh = n/2
nhp = nh+1
nhm = nh-1
\]
if (isign > 0) then
  Apply filter.
  wksp(1:nhm) = C0*a(1:n-3:2)+C1*a(2:n-2:2) &
     +C2*a(3:n-1:2)+C3*a(4:n:2)
  wksp(nh)=C0*a(n-1)+C1*a(n)+C2*a(1)+C3*a(2)
  wksp(nhp:n-1) = C3*a(1:n-3:2)-C2*a(2:n-2:2) &
     +C1*a(3:n-1:2)-C0*a(4:n:2)
  wksp(n)=C3*a(n-1)-C2*a(n)+C1*a(1)-C0*a(2)
else
  Apply transpose filter.
  wksp(1)=C2*a(nh)+C1*a(n)+C0*a(1)+C3*a(nhp)
  wksp(2)=C3*a(nh)-C0*a(n)+C1*a(1)-C2*a(nhp)
  wksp(3:n-1:2) = C2*a(1:nhm)+C1*a(nhp:n-1) &
     +C0*a(2:nh)+C3*a(nh+2:n)
  wksp(4:n:2) = C3*a(1:nhm)-C0*a(nhp:n-1) &
     +C1*a(2:nh)-C2*a(nh+2:n)
end if
a(1:n)=wksp(1:n)
END SUBROUTINE daub4

MODULE pwtcom
USE nrtype
INTEGER(I4B), SAVE :: ncof=0,ioff,joff
These module variables communicate the
REAL(SP), DIMENSION(:), ALLOCATABLE, SAVE :: cc,cr
filter to pwt.
END MODULE pwtcom

SUBROUTINE pwtset(n)
USE nrtype; USE nrutil, ONLY : nrerror
USE pwtcom
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
Initializing routine for pwt, here implementing the Daubechies wavelet filters with 4, 12, and 20 coefficients, as selected by the input value n. Further wavelet filters can be included in the obvious manner. This routine must be called (once) before the first use of pwt. (For the case n=4 the specific routine daub4 is considerably faster than pwt.)
REAL(SP) :: sig
REAL(SP), PARAMETER :: &
c4(4)=/(0.4829629131445341_sp, 0.8365163037378079_sp, &
0.224143880420134_sp,-0.1294095225512604_sp ), &
c12(12)=/0.111540743350_sp, 0.494623890398_sp, 0.751133908021_sp, &
0.315250351709_sp,-0.226264693965_sp,-0.12976867567_sp, &
0.097501605587_sp, 0.027522865530_sp,-0.031582039318_sp, &
0.000553842201_sp, 0.004777257511_sp,-0.001077301085_sp )/,
&
c20(20)=/0.02667057901_sp, 0.188176800078_sp, 0.527201188932_sp, &
0.688495039654_sp, 0.261172345661_sp,-0.249846424327_sp, &
-0.195946274377_sp, 0.127369403338_sp, 0.093057364604_sp, &
-0.071394147166_sp,-0.029457536822_sp, 0.033212674059_sp, &
0.003665535657_sp,-0.01073175483_sp, 0.001395351747_sp, &
0.001992405295_sp,-0.000685856895_sp,-0.000116466855_sp, &
0.00093588670_sp,-0.000013264203_sp )/
if (allocated(cc)) deallocate(cc)
if (allocated(cr)) deallocate(cr)
allocate(cc(n),cr(n))
ncof=n
ioff=-n/2
These values center the “support” of the wavelets at each
joff=-n/2
level. Alternatively, the “peaks” of the wavelets can
sig=-1.0
be approximately centered by the choices ioff=-2
select case(n)
and joff=-n/2. Note that daub4 and pwtset with
case(4)

n=4 use different default centerings.
Here we need to have as global variables arrays whose dimensions are known only at run time. At first sight the situation is the same as with the module \texttt{fminln} in \texttt{newt} on p. 1197. If you review the discussion there and in §21.5, you will recall that there are two good ways to implement this: with allocatable arrays ("Method 1") or with pointers ("Method 2"). There is a difference here that makes allocatable arrays simpler. We do not wish to deallocate the arrays on exiting \texttt{pwtset}. On the contrary, the values in \texttt{cc} and \texttt{cr} need to be preserved for use in \texttt{pwt}. Since allocatable arrays are born in the well-defined state of "not currently allocated," we can declare the arrays here as

\begin{verbatim}
REAL(SP), DIMENSION(:,), ALLOCATABLE, SAVE :: cc, cr
\end{verbatim}

and test whether they were used on a previous call with

\begin{verbatim}
if (.not. allocated(cc)) deallocate(cc)
if (.not. allocated(cr)) deallocate(cr)
\end{verbatim}

We are then ready to allocate the new storage:

\begin{verbatim}
allocate(cc(n), cr(n))
\end{verbatim}

With pointers, we would need the additional machinery of nullifying the pointers on the initial call, since pointers are born in an undefined state (see §21.5).

There is an additional important point in this example. The module variables need to be used by a "sibling" routine, \texttt{pwt}. We need to be sure that they do not become undefined when we exit \texttt{pwtset}. We could ensure this by putting a \texttt{USE pwtcom} in the main program that calls both \texttt{pwtset} and \texttt{pwt}, but it's easy to forget to do this. It is preferable to put explicit \texttt{SAVE}s on all the module variables.

\begin{verbatim}
SUBROUTINE pwtset(a, isign)
USE nrtype; USE nrutil, ONLY : arth, nrerror
USE pwtcom
IMPLICIT NONE
REAL(SP), DIMENSION(:,), ALLOCATABLE, SAVE :: cc, cr
INTEGER(I4B), DIMENSION(size(a)/2) :: jf, jr
INTEGER(I4B) :: k, n, nh, nmod

n = size(a)
if (n < 4) RETURN
if (ncof == 0) call nrerror('pwt: must call pwtset before pwt')
nmod = ncof*n
\end{verbatim}

A positive constant equal to zero mod \( n \).
nh=n/2
wksp(:)=0.0
jf=ian(d(n-1, art(h(2*nmod+ioff, 2, nh)))
jr=ian(d(n-1, art(h(2*nmod+ioff, 2, nh)))
if (isign >= 0) then
  wksp(1:n)=wksp(1:n)+cc(k)*a(jf+1)
  wksp(n+1:n)=wksp(n+1:n)+cr(k)*a(jr+1)
else
  wksp(jf+1)=wksp(jf+1)+cc(k)*a(1:nh)
  wksp(jr+1)=wksp(jr+1)+cr(k)*a(nh+1:n)
end if
if (k == ncof) exit
jf=ian(d(n-1,jf+1)
jr=ian(d(n-1,jr+1)
do k=1,ncof
end do
wksp(:)=0.0

SUBROUTINE wtn(a,nn,isign,wtstep)
USE nrtype, USE nrutil, ONLY : arth, assert
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nn
INTEGER(I4B), INTENT(IN) :: isign
INTERFACE
SUBROUTINE wtstep(a,isign)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE wtstep
END INTERFACE

Replaces a by its N-dimensional discrete wavelet transform, if isign is
input as 1. nn is an integer array of length N, containing the length of each
dimension (number of real values), which must all be powers of 2. a is a real
array of length equal to the product of these lengths, in which the data are
stored as in a multidimensional real FORTRAN array. If isign is input as
-1, a is replaced by its inverse wavelet transform. The subroutine wtstep,
whose actual name must be supplied in calling this routine, is the underly-
ing wavelet filter.

Examples of wtstep are daub4 and (preceded by pwtset) pwt.

INTEGER(I4B) :: i1,i2,i3,idim,n,ndim,nnew,nprev,nt,ntot
REAL(SP), DIMENSION(:), ALLOCATABLE :: wksp
call assert(iand(nn,nn-1)==0, 'each dimension must be a power of 2 in wtn')
allocate(wksp(maxval(nn)))
ndim=size(nn)
ntot=product(nn(:))
nprev=1
do idim=1,ndim
  n=nn(idim)
  nnew=n*nprev
  if (n > 4) then
    do i2=0,ntot-1,nnew
      i1=1,nprev
      i3=i1+i2
      wksp(1:n)=a(arth(i3,nprev,n))
      if (isign == 0) then
        nt=n
do
if (nt > n) exit
    call wtstep(wksp(1:nt),isign)
    nt=nt*2
end do
else
    nt=4
    do
        if (nt < 4) exit
        call wtstep(wksp(1:nt),isign)
        nt=nt/2
    end do
    end if
    i3=i1+i2
    a(arth(i3,nprev,n))=wksp(1:n)  
    i3=i3+n*nprev
end do
end do
end if
nprev=nnew
end do
deallocate(wksp)
END SUBROUTINE wtn
Chapter B14. Statistical Description of Data

SUBROUTINE moment(data, ave, adev, sdev, var, skew, curt)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: ave, adev, sdev, var, skew, curt
REAL(SP), DIMENSION(:), INTENT(IN) :: data

Given an array of data, this routine returns its mean ave, average deviation adev, standard deviation sdev, variance var, skewness skew, and kurtosis curt.

INTEGER(I4B) :: n
REAL(SP) :: ep
REAL(SP), DIMENSION(size(data)) :: p, s
n = size(data)
if (n <= 1) call nrerror('moment: n must be at least 2')
ave = sum(data(:))/n
First pass to get the mean.
s(:) = data(:) - ave
Second pass to get the first (absolute), second, third, and fourth moments of the deviation from the mean.
ep = sum(s(:))
adev = sum(abs(s(:)))/n
p(:) = s(:) * s(:)
var = sum(p(:))/n
p(:) = p(:) * s(:)
skew = sum(p(:))
p(:) = p(:) * s(:)
curt = sum(p(:))

var = (var - ep**2/n) / (n-1)
Corrected two-pass formula.
sdev = sqrt(var)
if (var /= 0.0) then
  skew = skew / (n*sdev**3)
curt = curt / (n*var**2) - 3.0_sp
else
  call nrerror('moment: no skew or kurtosis when zero variance')
end if
END SUBROUTINE moment

SUBROUTINE ttest(data1, data2, t, prob)
USE nrtype
USE nr, ONLY : avevar, betai
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1, data2
REAL(SP), INTENT(OUT) :: t, prob

Given the arrays data1 and data2, which need not have the same length, this routine returns Student's t as t, and its significance as prob, small values of prob indicating that
the arrays have significantly different means. The data arrays are assumed to be drawn from populations with the same true variance.

```fortran
INTEGER(I4B) :: n1,n2
REAL(SP) :: ave1,ave2,df,var1,var2
n1=size(data1)
n2=size(data2)
call avevar(data1,ave1,var1)
call avevar(data2,ave2,var2)
df=n1+n2-2
var=((n1-1)*var1+(n2-1)*var2)/df
Degrees of freedom.
t=(ave1-ave2)/sqrt(var*(1.0_sp/n1+1.0_sp/n2))
Pooled variance.
prob=betai(0.5_sp*df,0.5_sp,df/(df+t**2))
See equation (6.4.9).
END SUBROUTINE ttest
```

```fortran
SUBROUTINE avevar(data,ave,var)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data
REAL(SP), INTENT(OUT) :: ave,var
INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(data)) :: s
n=size(data)
ave=sum(data(:))/n
s(:)=data(:)-ave
var=dot_product(s,s)
Corrected two-pass formula (14.1.8).
var=(var-sum(s)**2/n)/(n-1)
END SUBROUTINE avevar
```

```fortran
SUBROUTINE tutest(data1,data2,t,prob)
USE nrtype
USE nr, ONLY : avevar,betai
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: t,prob

Given the arrays data1 and data2, which need not have the same length, this routine returns Student’s t as t, and its significance as prob, small values of prob indicating that the arrays have significantly different means. The data arrays are allowed to be drawn from populations with unequal variances.

```fortran
INTEGER(I4B) :: n1,n2
REAL(SP) :: ave1,ave2,df,var1,var2
n1=size(data1)
n2=size(data2)
call avevar(data1,ave1,var1)
call avevar(data2,ave2,var2)
t=(ave1-ave2)/sqrt(var1/n1+var2/n2)
Corrected two-pass formula (14.1.8).
df=((var1/n1+var2/n2)**2/((var1/n1)**2/(n1-1)+var2/n2)**2/n2)**2/(n2-1))
prob=betai(0.5_sp*df,0.5_sp,df/(df+t**2))
END SUBROUTINE tutest
```
SUBROUTINE tptest(data1, data2, t, prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : avevar, betai
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1, data2
REAL(SP), INTENT(OUT) :: t, prob

Given the paired arrays data1 and data2 of the same length, this routine returns Student's
t for paired data as t, and its significance as prob, small values of prob indicating a
significant difference of means.

INTEGER(I4B) :: n
REAL(SP) :: ave1, ave2, cov, df, sd, var1, var2
n = assert_eq(size(data1), size(data2), 'tptest')
call avevar(data1, ave1, var1)
call avevar(data2, ave2, var2)
cov = dot_product(data1(:) - ave1, data2(:) - ave2)
df = n - 1
sd = sqrt((var1 + var2 - 2.0_sp * cov) / n)
t = (ave1 - ave2) / sd
prob = betai(0.5_sp * df, 0.5_sp, df / (df + t**2))
END SUBROUTINE tptest

* * *

SUBROUTINE ftest(data1, data2, f, prob)
USE nrtype
USE nr, ONLY : avevar, betai
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: f, prob
REAL(SP), DIMENSION(:), INTENT(IN) :: data1, data2

Given the arrays data1 and data2, which need not have the same length, this routine
returns the value of f, and its significance as prob. Small values of prob indicate that the
two arrays have significantly different variances.

INTEGER(I4B) :: n1, n2
REAL(SP) :: ave1, ave2, df1, df2, var1, var2
n1 = size(data1)
n2 = size(data2)
call avevar(data1, ave1, var1)
call avevar(data2, ave2, var2)
if (var1 > var2) then
  Make F the ratio of the larger variance to the smaller one.
  f = var1 / var2
  df1 = n1 - 1
  df2 = n2 - 1
else
  f = var2 / var1
  df1 = n1 - 1
  df2 = n2 - 1
end if
prob = 2.0_sp * betai(0.5_sp * df2, 0.5_sp * df1, df2 / (df2 + df1 * f))
if (prob > 1.0) prob = 2.0_sp - prob
END SUBROUTINE ftest

* * *
SUBROUTINE chsone(bins, ebins, knstrn, df, chsq, prob)
USE nrtype; USE nrutil, ONLY : assert_eq, nrerror
USE nr, ONLY : gammq
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: knstrn
REAL(SP), INTENT(OUT) :: df, chsq, prob
REAL(SP), DIMENSION(:), INTENT(IN) :: bins, ebins

Given the same-size arrays bins containing the observed numbers of events, and ebins containing the expected numbers of events, and given the number of constraints knstrn (normally one), this routine returns (trivially) the number of degrees of freedom df, and (nontrivially) the chi-square chsq and the significance prob. A small value of prob indicates a significant difference between the distributions bins and ebins. Note that bins and ebins are both real arrays, although bins will normally contain integer values.

INTEGER(I4B) :: ndum
ndum=assert_eq(size(bins), size(ebins), 'chsone')
if (any(ebins(:) <= 0.0)) call nrerror('bad expected number in chsone')
df=size(bins)-knstrn
chsq=sum((bins(:)-ebins(:))**2/ebins(:))
prob=gammq(0.5_sp*df, 0.5_sp*chsq)  
Chi-square probability function. See §6.2.
END SUBROUTINE chsone

SUBROUTINE chstwo(bins1, bins2, knstrn, df, chsq, prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : gammq
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: knstrn
REAL(SP), INTENT(OUT) :: df, chsq, prob
REAL(SP), DIMENSION(:), INTENT(IN) :: bins1, bins2

Given the same-size arrays bins1 and bins2, containing two sets of binned data, and given the number of constraints knstrn (normally 1 or 0), this routine returns the number of degrees of freedom df, the chi-square chsq, and the significance prob. A small value of prob indicates a significant difference between the distributions bins1 and bins2. Note that bins1 and bins2 are both real arrays, although they will normally contain integer values.

INTEGER(I4B) :: ndum
LOGICAL(LGT), DIMENSION(size(bins1)) :: nzeromask
ndum=assert_eq(size(bins1), size(bins2), 'chstwo')
nzeromask = bins1(:) /= 0.0 .or. bins2(:) /= 0.0
chsq=sum((bins1(:)-bins2(:))**2/(bins1(:)+bins2(:)), nzeromask)
df=count(nzeromask)-knstrn  
No data means one less degree of freedom.
prob=gammq(0.5_sp*df, 0.5_sp*chsq)  
Chi-square probability function. See §6.2.
END SUBROUTINE chstwo

*   *   *

We use the optional argument mask in sum to select out the elements to be summed over. In this case, at least one of the elements of bins1 or bins2 is not zero for each term in the sum.
SUBROUTINE ksonedata(data,func,d,prob)
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : probks,sort
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: d,prob
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
GIVEN a narray data, and given a user-supplied function of a single variable func which
is a cumulative distribution function ranging from 0 (for smallest values of its argument)
to 1 (for largest values of its argument), this routine returns the K–S statistic d, and
the significance level prob. Small values of prob show that the cumulative distribution function
of data is significantly different from func. The array data is modified by being sorted
into ascending order.
INTEGER(I4B) :: n
REAL(SP) :: en
REAL(SP), DIMENSION(size(data)) :: fvals
REAL(SP), DIMENSION(size(data)+1) :: temp
calldsort(data)
n=size(data)
en=n
fvals(:)=func(data(:))
temp=arth(0,1,n+1)/en
d=maxval(max(abs(temp(1:n)-fvals(:)),
Compute the maximum distance between
the data’s c.d.f. and the user-supplied
function.
en=sqrt(en)
prob=probks((en+0.12_sp+0.11_sp/en)*d)
Compute significance.
END SUBROUTINE ksonedata

Note the difference between max and maxval: max
takes two or more arguments and returns the maximum. If the arguments
are two arrays, it returns an array each of whose elements is the maximum
of the corresponding elements in the two arrays. maxval takes a single array
argument and returns its maximum value.

SUBROUTINE kstwodata(data1,data2,d,prob)
USE nrtype; USE nrutil, ONLY : cumsum
USE nr, ONLY : probks,sort2
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: d,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
Given arrays data1 and data2, which can be of different length, this routine returns the
K–S statistic d, and the significance level prob for the null hypothesis that the data sets
are drawn from the same distribution. Small values of prob show that the cumulative
distribution function of data1 is significantly different from that of data2. The arrays
data1 and data2 are not modified.
INTEGER(I4B) :: n1,n2
REAL(SP) :: en1,en2,en
REAL(SP), DIMENSION(size(data1)+size(data2)) :: dat,org
n1=size(data1)
n2=size(data2)
en1=n1
en2=en2
dat(1:n1)=data1
Copy the two data sets into a single ar-
dat(n1+1:)=data2
ray.
The problem here is how to compute the cumulative distribution function (c.d.f.) corresponding to each set of data, and then find the corresponding KS statistic, without a serial loop over the data. The trick is to define an array that contains 0 when the corresponding element comes from the first data set and 1 when it’s from the second data set. Sort the array of 1’s and 0’s into the same order as the merged data sets. Now tabulate the partial sums of the array. Every time you encounter a 1, the partial sum increases by 1. So if you normalize the partial sums by dividing by the number of elements in the second data set, you have the c.d.f. of the second data set. If you subtract the array of 1’s and 0’s from an array of all 1’s, you get an array where 1 corresponds to an element in the first data set, 0 the second data set. So tabulating its partial sums and normalizing gives the c.d.f. of the first data set. As we’ve seen before, tabulating partial sums can be done with a parallel algorithm (cumsum in nrutil11). The KS statistic is just the maximum absolute difference of the c.d.f.’s, computed in parallel with Fortran 90’s maxval function.

```fortran
FUNCTION probks(alam)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: alam
    REAL(SP) :: probks
    REAL(SP), PARAMETER :: EPS1=0.001_sp,EPS2=1.0e-8_sp
    INTEGER(I4B), PARAMETER :: NITER=100
    Kolmogorov-Smirnov probability function.
    INTEGER(I4B) :: j
    REAL(SP) :: a2,fac,term,termbf
    a2=-2.0_sp*alam**2
    fac=2.0
    probks=0.0
    termbf=0.0
    Previous term in sum.
    do j=1,NITER
        term=fac*exp(a2*j**2)
        probks=probks+term
        if (abs(term) <= EPS1*termbf .or. abs(term) <= EPS2*probks) RETURN
        fac=-fac
        Alternating signs in sum.
        termbf=abs(term)
    end do
    probks=1.0
    Get here only by failing to converge, which implies the function is very close to 1.
END FUNCTION probks
```

```fortran
org(1:n1)=0.0
org(n1+1:)=1.0
Define an array that contains 0 when the corresponding element comes from
data1, 1 from data2.
call sort2(dat,org)
Sort the array of 1’s and 0’s into the order of the merged data sets.
d=maxval(ahs(cumsum(org)/en2-cumsum(1.0_sp-org)/en1))
Now use cumsum to get the c.d.f. corresponding to each set of data.
en=sqrt(en1*en2/(en1+en2))
prob=probks((en*0.12_sp+0.11_sp/en)*d)
Compute significance.
END SUBROUTINE kstwo
```
SUBROUTINE cntab1(nn,chisq,df,prob,cramrv,ccc)
USE nr, ONLY : outerprod
USE nrtype; USE nrutil, ONLY : outerprod, gammq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,:), INTENT(IN) :: nn
REAL(SP), INTENT(OUT) :: chisq,df,prob,cramrv,ccc
REAL(SP), PARAMETER :: TINY=1.0e-30_sp

Given a two-dimensional contingency table in the form of a rectangular integer array nn, this routine returns the chi-square \( chisq \), the number of degrees of freedom \( df \), the significance level \( prob \) (small values indicating a significant association), and two measures of association, Cramer's V (\( cramrv \)), and the contingency coefficient \( C \) (ccc).

INTEGER(I4B) :: nni,nnj
REAL(SP) :: sumn
REAL(SP), DIMENSION(size(nn,1)) :: sumi
REAL(SP), DIMENSION(size(nn,2)) :: sumj
REAL(SP), DIMENSION(size(nn,1),size(nn,2)) :: expctd
sumi(:)=sum(nn(:,:),dim=2) ! Get the row totals.
sumj(:)=sum(nn(:,:),dim=1) ! Get the column totals.
sumn=sum(sumi(:)) ! Get the grand total.
ni=size(sumi(:))-count(sumi(:) == 0.0) ! Eliminate any zero rows by reducing the number of rows.
nj=size(sumj(:))-count(sumj(:) == 0.0) ! Eliminate any zero columns.
df=nni*nnj-nni-nnj+1 ! Corrected number of degrees of freedom.
expctd(:,:)=outerprod(sumi(:),sumj(:))/sumn ! Do the chi-square sum. Here TINY guarantees that any eliminated row or column will not contribute to the sum.
chisq=sum((nn(:,:)-expctd(:,:))**2/(expctd(:,:)+TINY)) ! Chi-square probability function.
prob=gammq(0.5_sp*df,0.5_sp*chisq)
cramrv=sqrt(chisq/(sumn*min(nni-1,nnj-1)))
ccc=sqrt(chisq/(chisq+sumn))
END SUBROUTINE cntab1

SUBROUTINE cntab2(nn,h,hx,hy,hygx,hxgy,uygx,uxgy,uxy)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,:), INTENT(IN) :: nn
REAL(SP), INTENT(OUT) :: h,hx,hy,hygx,hxgy,uygx,uxgy,uxy
REAL(SP), PARAMETER :: TINY=1.0e-30_sp

Given a two-dimensional contingency table in the form of a rectangular integer array nn, where the first index labels the \( x \)-variable and the second index labels the \( y \)-variable, this routine returns the entropy \( h \) of the whole table, the entropy \( hx \) of the \( x \)-distribution, the entropy \( hy \) of the \( y \)-distribution, the entropy \( hygx \) of \( y \) given \( x \), the entropy \( hxgy \) of \( x \) given \( y \), the dependency \( uygx \) of \( y \) on \( x \) (eq. 14.4.15), the dependency \( uxgy \) of \( x \) on \( y \) (eq. 14.4.16), and the symmetrical dependency \( uxy \) (eq. 14.4.17).

REAL(SP) :: sum
REAL(SP), DIMENSION(size(nn,1)) :: sumi
REAL(SP), DIMENSION(size(nn,2)) :: sumj

This is a direct implementation of equation (14.4.2) using outerprod from nrutil.

entropy of the \( x \)-distribution, the entropy \( hy \) of the \( y \)-distribution, the entropy \( hygx \) of \( y \) given \( x \), the entropy \( hxgy \) of \( x \) given \( y \), the dependency \( uygx \) of \( y \) on \( x \) (eq. 14.4.15), the dependency \( uxgy \) of \( x \) on \( y \) (eq. 14.4.16), and the symmetrical dependency \( uxy \) (eq. 14.4.17).

REAL(SP) :: sum
REAL(SP), DIMENSION(size(nn,1)) :: sumi
REAL(SP), DIMENSION(size(nn,2)) :: sumj

This is a direct implementation of equation (14.4.2) using outerprod from nrutil.

entropy of the \( x \)-distribution, the entropy \( hy \) of the \( y \)-distribution, the entropy \( hygx \) of \( y \) given \( x \), the entropy \( hxgy \) of \( x \) given \( y \), the dependency \( uygx \) of \( y \) on \( x \) (eq. 14.4.15), the dependency \( uxgy \) of \( x \) on \( y \) (eq. 14.4.16), and the symmetrical dependency \( uxy \) (eq. 14.4.17).

REAL(SP) :: sum
REAL(SP), DIMENSION(size(nn,1)) :: sumi
REAL(SP), DIMENSION(size(nn,2)) :: sumj

This is a direct implementation of equation (14.4.2) using outerprod from nrutil.

entropy of the \( x \)-distribution, the entropy \( hy \) of the \( y \)-distribution, the entropy \( hygx \) of \( y \) given \( x \), the entropy \( hxgy \) of \( x \) given \( y \), the dependency \( uygx \) of \( y \) on \( x \) (eq. 14.4.15), the dependency \( uxgy \) of \( x \) on \( y \) (eq. 14.4.16), and the symmetrical dependency \( uxy \) (eq. 14.4.17).
and of the $y$ distribution.

$$h = \sum (n_{i,j} \log(n_{i,j})/\sum n)$$

Total entropy: loop over both $x$ and $y$.

$$h_{xy} = h - h_x$$

Uses equation (14.4.18).

$$h_{yx} = h - h_y$$

as does this.

$$u_{yx} = (h_y - h_{yx})/(h_y + \text{TINY})$$

Equation (14.4.15).

$$u_{xy} = 2.0_{sp} (h_x - h_{xy})/(h_x + \text{TINY})$$

Equation (14.4.16).

$$u_{xy} = 2.0_{sp} (h_x - h_{xy})/(h_x + \text{TINY})$$

Equation (14.4.17).

END SUBROUTINE cntab2

This code exploits both the dim feature of sum (see discussion after cntab1) and the mask feature to restrict the elements to be summed over.

* * *

SUBROUTINE pearsn(x,y,r,prob,z)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : betai
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: r,prob,z
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y
REAL(SP), PARAMETER :: TINY=1.0e-20_sp

Given two arrays $x$ and $y$ of the same size, this routine computes their correlation coefficient $r$ (returned as $r$), the significance level at which the null hypothesis of zero correlation is disproved ($\text{prob}$ whose small value indicates a significant correlation), and Fisher's $z$ (returned as $z$), whose value can be used in further statistical tests as described above the routine in Volume 1.

Parameter: $\text{TINY}$ will regularize the unusual case of complete correlation.

REAL(SP), DIMENSION(size(x)) :: xt,yt
REAL(SP) :: ax,ay,df,sxx,sxy,syy,t
INTEGER(I4B) :: n
n=assert_eq(size(x),size(y),'pearsn')

ax=sum(x)/n
ay=sum(y)/n
xt(:)=x(:)-ax
Compute the correlation coefficient.

yt(:)=y(:)-ay
sxx=dot_product(xt,xt)
syy=dot_product(yt,yt)
sxy=dot_product(xt,yt)

$$r = sxy/(\sqrt{sxx syy} + \text{TINY})$$

Fisher's $z$ transformation.

$$df = n - 2$$

Equation (14.5.5).

$$t = r \sqrt{df/((1.0_{sp}-r)^2+(1.0_{sp}+r)^2))}$$

Student's $t$ probability.

$$\text{prob} = \text{betai}(0.5_{sp} df,0.5_{sp} df/df+t**2)$$

For large $n$, this easier computation of $\text{prob}$, using the short routine erfcc, would give approximately the same value.

END SUBROUTINE pearsn

* * *

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. Permission is granted for internet use to make one paper copy of this module for personal use on portable devices. Further reproduction for any purpose whatsoever is forbidden without written permission from the publisher. Foreword and other material not claimed as your own are copyright by Cambridge University Press.

SUBROUTINE spear(data1, data2, d, zd, probd, rs, probrs)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : betai, erfcc, sort2
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1, data2
REAL(SP), INTENT(OUT) :: d, zd, probd, rs, probrs

Given two data arrays of the same size, data1 and data2, this routine returns their sum-
squared difference of ranks as $D$, the number of standard deviations by which $D$ deviates
from its null-hypothesis expected value as $zd$, the two-sided significance level of this devia-
tion as probd, Spearman's rank correlation $rs$ as $rs$, and the two-sided significance level of
its deviation from zero as probrs. data1 and data2 are not modified. A small value of either
probd or probrs indicates a significant correlation ($rs$ positive) or anticorrelation
($rs$ negative).

INTEGER(I4B) :: n
REAL(SP) :: aved, df, en, en3n, fac, sf, sg, t, vard
REAL(SP), DIMENSION(size(data1)) :: wksp1, wksp2
n = assert_eq(size(data1), size(data2), 'spear')
wksp1(:) = data1(:)
wksp2(:) = data2(:)
call sort2(wksp1, wksp2)
Sort each of the data arrays, and convert the
entries to ranks. The values sf and sg
return the sums $\sum (f^3_k - f_k)$ and $\sum (g^3_m - g_m)$, respectively.
call crank(wksp1, sf)
call sort2(wksp2, wksp1)
call crank(wksp2, sg)
wksp1(:) = wksp1(:) - wksp2(:)
d = dot_product(wksp1, wksp1)
Sum the squared difference of ranks.
en = n
en3n = en**3 - en
aved = en3n/12.0_sp - sf/en3n
vard = ((en-1.0_sp)*en**2*(en+1.0_sp)**2/36.0_sp)*fac
Expectation value of $D$, and variance of $D$ give
number of standard deviations, and significance.
zd = (d - aved)/sqrt(vard)
probd = erfcc(abs(zd)/SQRT2)
rs = (1.0_sp - (6.0_sp/en3n)*(d + (sf+sg)/12.0_sp))/sqrt(fac)
Rank correlation coeffi-
cient, fac = (1.0_sp + rs)*(1.0_sp - rs)
if (fac > 0.0) then
  t = rs*sqrt((en-2.0_sp)/fac)
  df = en-2.0_sp
  probrs = betai(0.5_sp*df, 0.5_sp, df/(df+t**2))
give its significance.
else
  probrs = 0.0
end if
CONTAINS

SUBROUTINE crank(w, s)
USE nrtype; USE nrutil, ONLY : arth, array_copy
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: s
REAL(SP), DIMENSION(:), INTENT(INOUT) :: w

Given a sorted array w, replaces the elements by their rank, including midranking of ties,
and returns as a the sum of $j^3 - j$, where $f$ is the number of elements in each tie.

INTEGER(I4B) :: i, n, ndum, nties
INTEGER(I4B), DIMENSION(size(w)) :: tstart, tend, tie, idx
n = size(w)
idx(:) = arth(1, 1, n) Index vector.
tie(:) = merge(1, 0, w == eoshift(w, -1))
Look for ties: Compare each element to the one before. If it's equal, it's part of a tie, and
we put 1 into tie. Otherwise we put 0.
tie(1) = 0 Boundary; the first element must be zero.
w(:) = idx(:) Assign ranks ignoring possible ties.
if (all(tie == 0)) then No ties—we're done.
s = 0.0
RETURN
end if
call array_copy(pack(idx(:), tie(:) < eoshift(tie(:), 1)), tstart, nties, ndum)
Look for $0 \rightarrow 1$ transitions in tie, which mean that the $0$ element is the start of a tie run.

Store index of each transition in tstart. nties is the number of ties found.

tend(1:nties)=pack(idx(:),tie(:)>eoshift(tie(:),1))

Look for $1 \rightarrow 0$ transitions in tie, which mean that the $1$ element is the end of a tie run.

do i=1,nties
   w(tstart(i):tend(i))=(tstart(i)+tend(i))/2.0_sp
   midrank assignments.
   tend(1:nties)=tend(1:nties)-tstart(1:nties)+1
   Now calculate $s$.
   s=sum(tend(1:nties)**3-tend(1:nties))

END SUBROUTINE crank

END SUBROUTINE spear

To understand how the parallel version of crank works, let's consider an example of 9 elements in the array $w$, which is input in sorted order to crank. The elements in our example are given in the second line of the following table:

<table>
<thead>
<tr>
<th>index</th>
<th>1 2 3 4 5 6 7 8 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>data in $w$</td>
<td>0 0 1 1 1 2 3 4 4</td>
</tr>
<tr>
<td>shift right</td>
<td>0 0 0 1 1 1 2 3 4</td>
</tr>
<tr>
<td>compare</td>
<td>1 1 0 1 1 0 0 0 1</td>
</tr>
<tr>
<td>tie array</td>
<td>0 1 0 1 1 0 0 0 1</td>
</tr>
<tr>
<td>shift left</td>
<td>1 0 1 1 0 0 0 1 0</td>
</tr>
<tr>
<td>$0 \rightarrow 1$</td>
<td>1 3 8</td>
</tr>
<tr>
<td>$1 \rightarrow 0$</td>
<td>2 5 9</td>
</tr>
</tbody>
</table>

We look for ties by comparing this array with itself, right shifted by one element ("shift right" in table). We record a 1 for each element that is the same, a 0 for each element that is different ("compare"). A 1 indicates the element is part of a tie with the preceding element, so we always set the first element to 0, even if it was a 1 as in our example. This gives the "tie array." Now wherever the tie array makes a transition $0 \rightarrow 1$ indicates the start of a tie run, while a $1 \rightarrow 0$ transition indicates the end of a tie run. We find these transitions by comparing the tie array to itself left shifted by one ("shift left"). If the tie array element is smaller than the shifted array element, we have a $0 \rightarrow 1$ transition and we record the corresponding index as the start of a tie. Similarly if the tie array element is larger we record the index as the end of a tie. Note that the shifts must be end-off shifts with zeros inserted in the gaps for the boundary conditions to work.

call array_copy(pack(idx(:),tie(:)>eoshift(tie(:),1)),
   tstart,nties,n dum)

The start indices (1, 3, and 8 in our example above) are here packed into the first few elements of tstart. array_copy is a useful routine in nrutil for copying elements from one array to another, when the number of elements to be copied is not known in advance. This line of code is equivalent to

\[
tstart(:)=0
\]
\[
tstart(:)=pack(idx(:), tie(:)< eoshift(tie(:),1),tstart(:))
\]
\[
nties=count(tstart(:) > 0)
\]

The point is that we don't know how many elements pack is going to select. We have to make sure the dimensions of both sides of the pack statement are the same,
so we set the optional third argument of pack to tstart. We then make a separate pass through tstart to count how many elements we copied. Alternatively, we could have used an additional logical array mask and coded this as

\[
\text{mask}(:) = \text{tie}(:) < \text{eoshift}(\text{tie}(:),1)
\]
\[
\text{nties} = \text{count}(\text{mask})
\]
\[
\text{tstart}(1:\text{nties}) = \text{pack}(\text{idx}(:), \text{mask})
\]

But we still need two passes through the mask array. The beauty of the array_copy routine is that \text{nties} is determined from the size of the first argument, without the necessity for a second pass through the array.

\*
\*
\*

SUBROUTINE kend1(data1, data2, tau, z, prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : erfcc
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: tau, z, prob
REAL(SP), DIMENSION(size(data1)) :: data1, data2

Given same-size data arrays data1 and data2, this program returns Kendall's \( \tau \) as tau, its number of standard deviations from zero as z, and its two-sided significance level as prob. Small values of prob indicate a significant correlation (tau positive) or anticorrelation (tau negative).

INTEGER(I4B) :: is, j, n, n1, n2
REAL(SP) :: var
REAL(SP), DIMENSION(size(data1)) :: a1, a2
n = assert_eq(size(data1), size(data2), 'kendl1')
n1 = 0
This will be the argument of one square root in (14.6.8), and this the other.
is = 0
This will be the numerator in (14.6.8).
do j = 1, n - 1
\text{a1}(j+1:n) = \text{data1}(j) - \text{data1}(j+1:n)
\text{a2}(j+1:n) = \text{data2}(j) - \text{data2}(j+1:n)
\text{n1} = \text{n1} + \text{count}(\text{a1}(j+1:n) /= 0.0)
\text{n2} = \text{n2} + \text{count}(\text{a2}(j+1:n) /= 0.0)
\text{is} = \text{is} + \text{count}(\text{a1}(j+1:n) > 0.0 \text{ and } \text{a2}(j+1:n) > 0.0) \&
\text{or. } \text{a1}(j+1:n) < 0.0 \text{ and } \text{a2}(j+1:n) < 0.0) - \text{count}(\text{a1}(j+1:n) > 0.0 \text{ and } \text{a2}(j+1:n) < 0.0) \&
\text{or. } \text{a1}(j+1:n) < 0.0 \text{ and } \text{a2}(j+1:n) > 0.0)
end do
\text{tau} = \text{real}(\text{is}, \text{sp}) / \sqrt{\text{real}(\text{n1}, \text{sp}) * \text{real}(\text{n2}, \text{sp})}
\text{Equation (14.6.8).}
\text{var} = (4.0_{\text{sp}} * \text{n} + 10.0_{\text{sp}}) / (9.0_{\text{sp}} * \text{n} * (\text{n} - 1.0_{\text{sp}}))
\text{Equation (14.6.9).}
\text{z} = \text{tau} / \text{sqrt}(\text{var})
\text{prob} = \text{erfcc}(\text{abs}(\text{z}) / \text{SQRT2})
\text{Significance.}
END SUBROUTINE kend1

SUBROUTINE kend2(tab, tau, z, prob)
USE nrtype; USE nrutil, ONLY : cumsum
USE nr, ONLY : erfcc
IMPLICIT NONE
REAL(SP), DIMENSION(:, :, :), INTENT(IN) :: tab
REAL(SP), INTENT(OUT) :: tau, z, prob

Given a two-dimensional table tab such that \text{tab}(k, l) contains the number of events falling in bin \( k \) of one variable and bin \( l \) of another, this program returns Kendall's \( \tau \) as tau, its number of standard deviations from zero as z, and its two-sided significance level as prob. Small values of prob indicate a significant correlation (tau positive) or anticorrelation (tau
negative) between the two variables. Although \texttt{tab} is a real array, it will normally contain integral values.

```fortran
REAL(SP), DIMENSION(size(tab,1),size(tab,2)) :: cum, cumt
INTEGER(I4B) :: i, j, ii, jj
REAL(SP) :: sc, sd, en1, en2, points, var
ii=size(tab,1)
jj=size(tab,2)
do i=1,ii
   cumt(i,jj:1:-1)=cumsum(tab(i,jj:1:-1))
end do
en2=sum(tab(1:ii,1:jj-1)*cumt(1:ii,2:jj))
do j=1,jj
   cum(ii:1:-1,j)=cumsum(cumt(ii:1:-1,j))
end do
points=cum(1,1)
sc=sum(tab(1:ii-1,1:jj-1)*cum(2:ii,2:jj))
do j=1,jj
   cum(ii:1:-1,j)=cumsum(cumt(ii:1:-1,j))
end do
sd=sum(tab(2:ii,1:jj-1)*cum(1:ii-1,2:jj))
do j=1,jj
   cum(ii:1:-1,j)=cumsum(cumt(ii:1:-1,j))
end do
en1=sum(tab(1:ii-1,1:jj)*cumt(2:ii,1:jj))
tau=(sc-sd)/sqrt((en1+sc+sd)*(en2+sc+sd))
var=(4.0_sp*points+10.0_sp)/(9.0_sp*points*(points-1.0_sp))
z=tau/sqrt(var)
prob=erfcc(abs(z)/SQRT2)
END SUBROUTINE kendl2
```

The underlying algorithm in \texttt{kendl2} might seem to require looping over all pairs of cells in the two-dimensional table \texttt{tab}. Actually, however, clever use of the \texttt{cumsum} utility function reduces this to a simple loop over all the cells; moreover this “loop” parallelizes into a simple parallel product and call to the \texttt{sum} intrinsic. The basic idea is shown in the following table:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(d)</td>
<td>(d)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(t)</td>
<td>(y)</td>
<td>(y)</td>
</tr>
<tr>
<td></td>
<td>(x)</td>
<td>(c)</td>
<td>(c)</td>
</tr>
<tr>
<td></td>
<td>(x)</td>
<td>(c)</td>
<td>(c)</td>
</tr>
<tr>
<td></td>
<td>(x)</td>
<td>(c)</td>
<td>(c)</td>
</tr>
</tbody>
</table>

Relative to the cell marked \(t\) (which we use to denote the numerical value it contains), the cells marked \(d\) contribute to the “discordant” tally in Volume 1’s equation (14.6.8),
while the cells marked $c$ contribute to the “concordant” tally. Likewise, the cells marked $x$ and $y$ contribute, respectively, to the “extra-$x$” and “extra-$y$” tallies. What about the cells left blank? Since we want to count pairs of cells only once, without duplication, these cells will be counted, relative to the location shown as $t$, when $t$ itself moves into the blank-cell area.

Symbolically we have

\[
\begin{align*}
\text{concordant} & = \sum_n t_n \left( \sum \text{lower right} c_m \right) \\
\text{discordant} & = \sum_n t_n \left( \sum \text{upper right} d_m \right) \\
\text{extra-}x & = \sum_n t_n \left( \sum \text{below} x_m \right) \\
\text{extra-}y & = \sum_n t_n \left( \sum \text{to the right} y_m \right)
\end{align*}
\]

(B14.1)

Here $n$ varies over all the positions in the table, while the limits of the inner sums are relative to the position of $n$. (The letters $t_n$, $c_m$, $d_m$, $x_m$, $y_m$ all represent the value in a cell; we use different letters only to make the relation with the above table clear.) Now the final trick is to recognize that the inner sums, over cells to the lower-or-upper-right, below, and to the right can be done in parallel by cumulative sums (cumsum) sweeping to the right and up. The routine does these in a nonintuitive order merely to be able to reuse maximally the scratch spaces $\text{cum}$ and $\text{cumt}$.

* * *

SUBROUTINE ks2dis(x1,y1,quadvl,d1,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : pearsn,probks,quadct
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1,y1
REAL(SP), INTENT(OUT) :: d1,prob
INTERFACE
SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENT(IN) :: x,y
REAL(SP), DIMENT(OUT) :: fa,fb,fc,fd
END SUBROUTINE quadvl
END INTERFACE

Two-dimensional Kolmogorov-Smirnov test of one sample against a model. Given the $x$- and $y$-coordinates of a set of data points in arrays $x1$ and $y1$ of the same length, and given a user-supplied function $\text{quadvl}$ that exemplifies the model, this routine returns the two-dimensional K-S statistic as $d1$, and its significance level as $\text{prob}$. Small values of $\text{prob}$ show that the sample is significantly different from the model. Note that the test is slightly distribution-dependent, so $\text{prob}$ is only an estimate.

INTEGER(14B) :: j,n1
REAL(SP) :: dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,rr,sqen
n1=assert_eq(size(x1),size(y1),'ks2dis')
d1=0.0
do j=1,n1
    Loop over the data points.
    call quadct(x1(j),y1(j),x1,y1,fa,fb,fc,fd)
    call quadvl(x1(j),y1(j),ga,gb,gc,gd)
    di=max(di,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
    For both the sample and the model, the distribution is integrated in each of four quadrants, and the maximum difference is saved.
end do

call pearsn(x1,y1,r1,dum,dumm)
    Get the linear correlation coefficient r1.
sqen=sqrt(real(n1,sp))
    sqen=sqrt(real(n1,sp))
rr=sqrt(1.0_sp-r1**2)
    Estimate the probability using the K-S probability function probks.
prob=probks(di*sqen/(1.0_sp+rr*(0.25_sp-0.75_sp/sqen)))
END SUBROUTINE ks2d1s

SUBROUTINE quadct(x,y,xx,yy,fa,fb,fc,fd)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y
REAL(SP), DIMENSION(:), INTENT(IN) :: xx,yy
REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
    Given an origin (x,y), and an array of points with coordinates xx and yy, count how many of them are in each quadrant around the origin, and return the normalized fractions. Quadrants are labeled alphabetically, counterclockwise from the upper right. Used by ks2d1s and ks2d2s.
INTEGER(I4B) :: na,nb,nc,nd,nn
REAL(SP) :: ff
nn=assert_eq(size(xx),size(yy),'quadct')
na=count(yy(:) > y .and. xx(:) > x)
nb=count(yy(:) > y .and. xx(:) <= x)
nc=count(yy(:) <= y .and. xx(:) <= x)
nd=nn-na-nb-nc
ff=1.0_sp/nn
fa=ff*na
fb=ff*nb
fc=ff*nc
fd=ff*nd
END SUBROUTINE quadct

SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y
REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
    This is a sample of a user-supplied routine to be used with ks2d1s. In this case, the model distribution is uniform inside the square \(-1 < x < 1, -1 < y < 1\). In general this routine should return, for any point \((x, y)\), the fraction of the total distribution in each of the four quadrants around that point. The fractions, \(fa, fb, fc,\) and \(fd\), must add up to 1. Quadrants are alphabetical, counterclockwise from the upper right.
REAL(SP) :: qa,qb,qc,qd
qa=min(2.0_sp,max(0.0_sp,1.0_sp-x))
qb=min(2.0_sp,max(0.0_sp,1.0_sp-y))
qc=min(2.0_sp,max(0.0_sp,x+1.0_sp))
qd=min(2.0_sp,max(0.0_sp,y+1.0_sp))
fa=0.25_sp*qa*qb
fb=0.25_sp*qb*qc
fc=0.25_sp*qc*qd
fd=0.25_sp*qd*qa
END SUBROUTINE quadvl
SUBROUTINE ks2d2s(x1,y1,x2,y2,d,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : pearsn,probks,quadct
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1,y1,x2,y2
REAL(SP), INTENT(OUT) :: d,prob
Compute two-dimensional Kolmogorov-Smirnov test on two samples. Input are the x- and
y-coordinates of the first sample in arrays x1 and y1 of the same length, and of the second
sample in arrays x2 and y2 of the same length (possibly different from the length of the first
sample). The routine returns the two-dimensional, two-sample K-S statistic as d, and its
significance level as prob. Small values of prob show that the two samples are significantly
different. Note that the test is slightly distribution-dependent, so prob is only an estimate.
INTEGER(I4B) :: j,n1,n2
REAL(SP) :: d1,d2,dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,r2,rr,sqen
n1=assert_eq(size(x1),size(y1),'ks2d2s: n1')
n2=assert_eq(size(x2),size(y2),'ks2d2s: n2')
d=0.0
do j=1,n1
First, use points in the first sample as origins.
call quadct(x1(j),y1(j),x1,y1,fa,fb,fc,fd)
call quadct(x2(j),y2(j),x1,y1,ga,gb,gc,gd)
d1=max(d1,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
end do

Do = 0.0
do j=1,n2
Then, use points in the second sample as origins.
call quadct(x2(j),y2(j),x1,y1,fa,fb,fc,fd)
call quadct(x1(j),y1(j),x2,y2,ga,gb,gc,gd)
d2=max(d2,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
end do

d=0.5_sp*(d1+d2) Average the K-S statistics.
sqen=sqrt(real(n1,sp)*real(n2,sp)/real(n1+n2,sp))
call pearsn(x1,y1,r1,dum,dumm) Get the linear correlation coefficient for each sample.
call pearsn(x2,y2,r2,dum,dumm) ple
rr=sqrt(1.0_sp-0.5_sp*(r1**2+r2**2))
Estimate the probability using the K-S probability function probks.
prob=probks(d*sqen/(1.0_sp+rr*(0.25_sp-0.75_sp/sqen)))
END SUBROUTINE ks2d2s

FUNCTION savgol(nl,nrr,ld,m)
USE nrtype; USE nrutil, ONLY : arth,assert,poly
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
Returns in wrap-around order (N.B.!) consistent with the argument respns in routine
convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data
points used, while nrr is the number of rightward (future) data points, making the total
number of data points used nl+nrr+1. ld is the order of the derivative desired (e.g.,
ld = 0 for smoothed function). m is the order of the smoothing polynomial, also equal to
the highest conserved moment; usual value is m = 2 or m = 4.
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
Returns in wrap-around order (N.B.!) consistent with the argument respns in routine
convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data
points used, while nrr is the number of rightward (future) data points, making the total
number of data points used nl+nrr+1. ld is the order of the derivative desired (e.g.,
ld = 0 for smoothed function). m is the order of the smoothing polynomial, also equal to
the highest conserved moment; usual value is m = 2 or m = 4.
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
Returns in wrap-around order (N.B.!) consistent with the argument respns in routine
convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data
points used, while nrr is the number of rightward (future) data points, making the total
number of data points used nl+nrr+1. ld is the order of the derivative desired (e.g.,
ld = 0 for smoothed function). m is the order of the smoothing polynomial, also equal to
the highest conserved moment; usual value is m = 2 or m = 4.
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
Returns in wrap-around order (N.B.!) consistent with the argument respns in routine
convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data
points used, while nrr is the number of rightward (future) data points, making the total
number of data points used nl+nrr+1. ld is the order of the derivative desired (e.g.,
ld = 0 for smoothed function). m is the order of the smoothing polynomial, also equal to
the highest conserved moment; usual value is m = 2 or m = 4.
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
Returns in wrap-around order (N.B.!) consistent with the argument respns in routine
convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data
points used, while nrr is the number of rightward (future) data points, making the total
number of data points used nl+nrr+1. ld is the order of the derivative desired (e.g.,
ld = 0 for smoothed function). m is the order of the smoothing polynomial, also equal to
the highest conserved moment; usual value is m = 2 or m = 4.
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
Returns in wrap-around order (N.B.!) consistent with the argument respns in routine
convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data
points used, while nrr is the number of rightward (future) data points, making the total
number of data points used nl+nrr+1. ld is the order of the derivative desired (e.g.,
ld = 0 for smoothed function). m is the order of the smoothing polynomial, also equal to
the highest conserved moment; usual value is m = 2 or m = 4.
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
Returns in wrap-around order (N.B.!) consistent with the argument respns in routine
convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data
points used, while nrr is the number of rightward (future) data points, making the total
number of data points used nl+nrr+1. ld is the order of the derivative desired (e.g.,
ld = 0 for smoothed function). m is the order of the smoothing polynomial, also equal to
the highest conserved moment; usual value is m = 2 or m = 4.
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
Returns in wrap-around order (N.B.!) consistent with the argument respns in routine
convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data
points used, while nrr is the number of rightward (future) data points, making the total
number of data points used nl+nrr+1. ld is the order of the derivative desired (e.g.,
ld = 0 for smoothed function). m is the order of the smoothing polynomial, also equal to
the highest conserved moment; usual value is m = 2 or m = 4.
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
Returns in wrap-around order (N.B.!) consistent with the argument respns in routine
convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data
points used, while nrr is the number of rightward (future) data points, making the total
number of data points used nl+nrr+1. ld is the order of the derivative desired (e.g.,
ld = 0 for smoothed function). m is the order of the smoothing polynomial, also equal to
the highest conserved moment; usual value is m = 2 or m = 4.
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
Returns in wrap-around order (N.B.!) consistent with the argument respns in routine
convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data
points used, while nrr is the number of rightward (future) data points, making the total
number of data points used nl+nrr+1. ld is the order of the derivative desired (e.g.,
ld = 0 for smoothed function). m is the order of the smoothing polynomial, also equal to
the highest conserved moment; usual value is m = 2 or m = 4.
if (ipj == 0) sm=sm+1.0_sp
mm=min(ipj,2*n-ipj)
do imj=-mm,mm,2
   a(1+(ipj+imj)/2,1+(ipj-imj)/2)=sm
end do
call ludcmp(a(:,,:),indx(:,),d)         Solve them: LU decomposition.
b(:,)=0.0
b(ld+1)=1.0                  Right-hand-side vector is unit vector, depending
call lubksb(a(:,,:),indx(:,),b(:,))      on which derivative we want.
Backsubstitute, giving one row of the inverse matrix.
savgol(:,)=0.0                      Zero the output array.
irng(:,)=arth(-nl,1,nrr+nl+1)
np=nl+nrr+1
savgol(mod(np-irng(:,),np)+1)=poly(real(irng(:,),sp),b(:,))
Each Savitzky-Golay coefficient is the value of the polynomial in (14.8.6) at the corresponding
integer. The polynomial coefficients are a row of the inverse matrix. The mod function takes
care of the wrap-around order.
END FUNCTION savgol

Here is an example of a loop that cannot be parallelized
in the framework of Fortran 90: We need to access “skew” sections of
the matrix a.
savgol...=poly(real(irng(:,),sp),b(:,))) The poly function in nrutil returns
the value of a polynomial, here the one in equation (14.8.6). We need the
explicit kind type parameter sp in the real function, otherwise it would return
type default real for the integer argument and would not automatically convert to
double precision if desired.
Chapter B15.  Modeling of Data

SUBROUTINE fit(x,y,a,b,siga,sigb,chi2,q,sig)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : gammq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(OUT) :: a,b,siga,sigb,chi2,q
REAL(SP), DIMENSION(:), OPTIONAL, INTENT(IN) :: sig

Given a set of data points in same-size arrays x and y, fit them to a straight line \( y = ax + bx \) by minimizing \( \chi^2 \). sig is an optional array of the same length containing the individual standard deviations. If it is present, then a, b are returned with their respective probable uncertainties \( \sigma_a \) and \( \sigma_b \), the chi-square \( \chi^2 \), and the goodness-of-fit probability \( q \) (that the fit would have \( \chi^2 \) this large or larger). If sig is not present, then q is returned as 1.0 and the normalization of \( \chi^2 \) is to unit standard deviation on all points.

INTEGER(I4B) :: ndata
REAL(SP) :: sigdat,ss,sx,sxoss,sy,st2
REAL(SP), DIMENSION(size(x)), TARGET :: t
REAL(SP), DIMENSION(:), POINTER :: wt
if (present(sig)) then
  ndata=assert_eq(size(x),size(y),size(sig),'fit')
  wt=>t
  Use temporary variable t to store weights.
  wt(:)=1.0_sp/(sig(:)**2)
  Accumulate sums with weights.
  ss=sum(wt(:))
  sx=dot_product(wt,x)
  sy=dot_product(wt,y)
else
  ndata=assert_eq(size(x),size(y),'fit')
  ss=real(size(x),sp)
  Accumulate sums without weights.
  sx=sum(x)
  sy=sum(y)
end if

sxoss=sx/ss

siga=sqrt((1.0_sp+sx*sx/(ss*st2))/ss)

sigb=sqrt(1.0_sp/st2)

if (present(sig)) then
  t(:)=t(:)/sig(:)
  Calculate \( \chi^2 \).
  else
    \( \chi^2 \)=dot_product(t,t)
end if

q=1.0

if (present(sig)) then
  t(:)=t(:)/sig(:)
  \( \chi^2 \)=dot_product(t,t)
else
  \( \chi^2 \)=dot_product(t,t)
end if

chi2=chi2+0.5_sp*size(x)-2.0_sp

if (ndata > 2) q=gammq(0.5_sp*size(x)-2.0_sp,chi2)   Equation (15.2.12).
real(sp), dimension(:), pointer :: wt...wt=>t  When standard deviations are supplied in sig, we need to compute the weights for the least squares fit in a temporary array wt. Later in the routine, we need another temporary array, which we call t to correspond to the variable in equation (15.2.15). It would be confusing to use the same name for both arrays. In Fortran 77 the arrays could share storage with an EQUIVALENCE declaration, but that is a deprecated feature in Fortran 90. We accomplish the same thing by making wt a pointer alias to t.

SUBROUTINE fitexy(x,y,sigx,sigy,a,b,siga,sigb,chi2,q)
USE nrtype; USE nrutil, ONLY : assert_eq,swap
USE nr, ONLY : avevar,brent,fit,gammq,mnbrak,zbrent
USE chixyfit
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sigx,sigy
REAL(SP), INTENT(OUT) :: a,b,siga,sigb,chi2,q
REAL(SP), PARAMETER :: POTN=1.571000_sp,BIG=1.0e30_sp,ACC=1.0e-3_sp
Straight-line fit to input data x and y with errors in both x and y, the respective standard deviations being the input quantities sigx and sigy, x, y, sigx, and sigy are all arrays of the same length. Output quantities are a and b such that y = a + bx minimizes \( \chi^2 \), whose value is returned as chi2. The \( \chi^2 \) probability is returned as q, a small value indicating a poor fit (sometimes indicating underestimated errors). Standard errors on a and b are returned as siga and sigb. These are not meaningful if either (i) the fit is poor, or (ii) b is so large that the data are consistent with a vertical (infinite b) line. If siga and sigb are returned as BIG, then the data are consistent with all values of b.

INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(x)), TARGET :: xx,yy,xx,yy,ww
REAL(SP), DIMENSION(6) :: ang,ch
REAL(SP) :: amx,amn,varx,vary,scale,bmn,bmx,d1,d2,r2,&
            dum1,dum2,dum3,dum4,dum5
n=assert_eq(size(x),size(y),size(sigx),size(sigy),'fitexy')
xxp=>xx  Set up communication with function chixy
yyp=>yy  through global variables in the module chixyfit.
sxp=>sx
syp=>sy
wwp=>ww
call avevar(x,dum1,varx)  Find the x and y variances, and scale the
data.
call avevar(y,dum1,vary)
scale=sqrt(varx/vary)
xx(:)=x(:)
yy(:)=y(:)*scale
sx(:)=sigx(:)*scale
sy(:)=sigy(:)*scale
ww(:)=sqrt(sx(:)**2+sy(:)**2)  Use both x and y weights in first trial fit.
call fit(xx,yy,dum1,b,dum2,dum3,dum4,dum5,ww)  Trial fit for b.
offs=0.0  Construct several angles for reference points.
ang(1)=0.0  Make b an angle.
ang(2)=atan(b)
ang(4)=0.0
ang(5)=ang(2)
ang(6)=potn
do j=4,6
    ch(j)=chixy(ang(j))
end do

call mnbrak(ang(1),ang(2),ang(3),ch(1),ch(2),ch(3),chiy)
Bracket the $\chi^2$ minimum and then locate it with brent.
chi2=brent(ang(1),ang(2),ang(3),chiy,ACC,b)
chi2=chiy(b)
a=aa
q=gamma(0.5_sp*(n-2),0.5_sp*chi2)
r2=1.0_sp/sum(ww(:))
Compute $\chi^2$ probability.

b=gamma(0.5_sp*(n-2),0.5_sp*chi2)

bmx=BIG

offs=chi2+1.0_sp

Do through saved values to bracket the desired roots. Note periodicity in slope angles.

if (ch(j) > offs) then
d1=mod(abs(ang(j)-b),PI)
d2=PI-d1
if (ang(j) < b) call swap(d1,d2)
if (d1 < bmx) bmx=d1
if (d2 < bmn) bmn=d2
end if
end do

if (bmx < BIG) then
Call zbrent to find the roots.
bmx=zbrent(chiy,b,b+bmx,ACC)-b
amx=aa-a
bmn=zbrent(chiy,b,b-bmn,ACC)-b
amn=aa-a

sigb=sqrt(0.5_sp*(bmx**2+bmn**2))/(scale*cos(b)**2)
Error in $a$ has additional piece $r2$.  
siga=sqrt(0.5_sp*(amx**2+amn**2)+r2)/scale
else
sigb=BIG
siga=BIG
end if

a=a/scale
Unscale the answers.
b=tan(b)/scale

END SUBROUTINE fitexy

USE chixyfit
We need to pass arrays and other variables to chixy, but not as arguments. See §21.5 and the discussion of fminln on p. 1197 for two good ways to do this. The pointer construction here is analogous to the one used in fminln.

MODULE chixyfit
USE nrtype; USE nrutil, ONLY : nrerror
REAL(SP), DIMENSION(:), POINTER :: xxp,yyp,sxp,syp,wwp
REAL(SP) :: aa,offs
CONTAINS

FUNCTION chixy(bang)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: bang
REAL(SP) :: chixy
REAL(SP), PARAMETER :: BIG=1.0e30_sp
Captive function of fitexy, returns the value of ($\chi^2 - \text{offs}$) for the slope $b=\tan(bang)$.  
Scaled data and offs are communicated via the module chixyfit.

REAL(SP) :: avex,avey,sumx,sumy,b
if (.not. associated(wwp)) call nrerror("chixy: bad pointers")
b=tan(bang)
wwp(:,)=(b*exp(:))*2+exp(:)**2
where (wwp(:) < 1.0/BIG)
wwp(:,)=BIG
elsewhere
wwp(:,)=1.0_sp/wwp(:)
end where
```fortran
sumw = sum(w*p)
ave = dot_product(w*p,x)*sumw
avey = dot_product(w*p,y)/sumw
a = ave-b*ave
chis = sum(w*p*(y-y-b*x)**2)-offs
END FUNCTION chisxy
END MODULE chisxy

* * *

SUBROUTINE lfit(x,y,sig,a,maska,covar,chisq,funcs)
USE nrtype; USE nrutil, ONLY : assert_eq, diagmult, nrerror
USE nr, ONLY : covsrt, gaussj
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: covar
REAL(SP), INTENT(OUT) :: chisq
INTERFACE
  SUBROUTINE funcs(x,arr)
  USE nrtype
  IMPLICIT NONE
  REAL(SP), INTENT(IN) :: x
  REAL(SP), DIMENSION(:), INTENT(OUT) :: arr
  END SUBROUTINE funcs
END INTERFACE

Given a set of \( N \) data points \( x, y \) with individual standard deviations \( \text{sig} \), all arrays of length \( N \), use \( \chi^2 \) minimization to fit for some or all of the \( M \) coefficients \( a \) of a function that depends linearly on \( a \), \( y = \sum_{i=1}^{M} a_i \times \text{afunc}(x) \). The input logical array \( \text{maska} \) of length \( M \) indicates by true entries those components of \( a \) that should be fitted for, and by false entries those components that should be held fixed at their input values. The program returns values for \( a \), \( \chi^2 = \text{chisq} \), and the \( M \times M \) covariance matrix \( \text{covar} \). (Parameters held fixed will return zero covariances.) The user supplies a subroutine \( \text{funcs}(x,\text{afunc}) \) that returns the \( M \) basis functions evaluated at \( x = x \) in the array \( \text{afunc} \).

INTEGER(I4B) :: i,j,k,l,ma,mfit,n
REAL(SP) :: sig2i,wt,ym
REAL(SP), DIMENSION(size(maska)) :: afunc
REAL(SP), DIMENSION(size(maska),1) :: beta
n = assert_eq(size(x), size(y), size(sig), 'lfit: n')
ma = assert_eq(size(maska), size(a), size(covar,1), size(covar,2), 'lfit: ma')
mfit = count(maska)
Number of parameters to fit for.
if (mfit == 0) call nrerror('lfit: no parameters to be fitted')
covar(1:mfit,1:mfit) = 0.0
Initialize the (symmetric) matrix.
beta(1:mfit,1) = 0.0
do i=1,n
  call funcs(x(i),afunc)
  ym = y(i)
  if (mfit < ma) ym = ym - sum(a(1:ma)*afunc(1:ma), mask=not. maska)
  Subtract off dependences on known pieces of the fitting function.
sig2i = 1.0_sp/sig(i)**2
j = 0
do l=1,ma
  if (maska(l)) then
    j = j + 1
    wt = afunc(l)*sig2i
        k = count(maska(1:l))
        covar(j,1:k) = covar(j,1:k) + wt*pack(afunc(1:l), maska(1:l))
        beta(j,1) = beta(j,1) + ym*wt
  end if
end do
```

end do

call diagmult(covar(1:mfit,1:mfit),0.5_sp)
covar(1:mfit,1:mfit)= &
  Fill in above the diagonal from symmetry.
covar(1:mfit,1:mfit)=transpose(covar(1:mfit,1:mfit))
call gaussj(covar(1:mfit,1:mfit),beta(1:mfit,1:1))
a(1:ma)=unpack(beta(1:ma,1),maska,a(1:ma))
  Matrix solution. Partition solution to appropriate coefficients a.
chisq=0.0  Evaluate $\chi^2$ of the fit.
do i=1,n
call func(x(i),afunc)
  This is the first of several uses of maska in this routine to control
  which elements of an array are to be used. Here we include in the sum
  only elements for which maska is false, i.e., elements corresponding to parameters
  that are not being fitted for.
  covar(j,1:k)=covar(j,1:k)+wt*pack(afunc(1:l),maska(1:l))
  Here maska controls which elements of afunc get packed into the covariance matrix.
  call diagmult(covar(1:mfit,1:mfit),0.5_sp)  See discussion of diagadd
  after hqr on p. 1234.
a(1:ma)=unpack(beta(1:ma,1),maska,a(1:ma))  And here maska controls which
  elements of beta get unpacked into the appropriate slots in a. Where maska is
  false, corresponding elements are selected from the third argument of unpack, here
  a itself. The net effect is that those elements remain unchanged.
  if (mfit < ma) ym=ym-\sum(a(1:ma)*afunc(1:ma), mask=.not. maska)
  This is the first of several uses of maska in this routine to control
  which elements of an array are to be used. Here we include in the sum
  only elements for which maska is false, i.e., elements corresponding to parameters
  that are not being fitted for.
  call covsrt(covar,maska)  Sort covariance matrix to true order of fitting
END SUBROUTINE lfit

*   *   *

SUBROUTINE covsrt(covar,maska)
USE nrtype; USE nrutil, ONLY : assert_eq,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: covar
LOGICAL(LGT), DIMENSION(:,), INTENT(IN) :: maska
  Expand in storage the covariance matrix covar, so as to take into account parameters
  that are being held fixed. (For the latter, return zero covariances.)
INTEGER(14B) :: ma,mfit,j,k
ma=assert_eq(size(covar,1),size(covar,2),size(maska),'covsrt')
mfit=count(maska)
covar(mfit+1:ma,1:ma)=0.0
covar(1:ma,mfit+1:ma)=0.0
k=mfit
do j=ma,1,-1
  if (maska(j)) then
    call swap(covar(1:ma,k),covar(1:ma,j))
    call swap(covar(k,1:ma),covar(j,1:ma))
k=k-1
  end if
end do
END SUBROUTINE covsrt

*   *   *
SUBROUTINE svdfit(x,y,sig,a,v,w,chisq,funcs)
USE nrtype; USE nrutil, ONLY : assert_eq,vabs
USE nr, ONLY : svbksb,svdcmp
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y,sig
REAL(SP), DIMENSION(:,), INTENT(OUT) :: a,w
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: v
REAL(SP), INTENT(OUT) :: chisq
INTERFACE
FUNCTION funcs(x,n)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: funcs
END FUNCTION funcs
END INTERFACE
REAL(SP), PARAMETER :: TOL=1.0e-5_sp
Given a set of \( N \) data points \( x, y \) with individual standard deviations \( \text{sig} \), all arrays of length \( N \), use \( \chi^2 \) minimization to determine the \( M \) coefficients \( a_i \) of a function that depends linearly on \( a, y = \sum_{i=1}^{M} a_i \times \text{func}_i(x) \). Here we solve the fitting equations using singular value decomposition of the \( N \times M \) matrix, as in §2.6. On output, the \( M \times M \) array \( v \) and the vector \( w \) of length \( M \) define part of the singular value decomposition, and can be used to obtain the covariance matrix. The program returns values for the \( M \) fit parameters \( a \), and \( \chi^2 \), \( \text{chisq} \). The user supplies a function \( \text{funcs}(x,\text{func}) \) that returns the \( M \) basis functions evaluated at \( x = X \) in the array \( \text{afunc} \).

Parameter: Above value of TOL is for single precision and variables scaled to order unity.

INTEGER(I4B) :: i,ma,n
REAL(SP), DIMENSION(size(x)) :: b,sigi
REAL(SP), DIMENSION(size(x),size(a)) :: u,usav
n=assert_eq(size(x),size(y),size(sig),'svdfit: n')
ma=assert_eq(size(a),size(v,1),size(v,2),size(w),'svdfit: ma')
sigi=1.0_sp/sig
 Accumulate coefficients of the fitting matrix in \( u \).
 b=y*sigi
 do i=1,n
 usav(i,:)=funcs(x(i),ma)
 end do
 u=usav*spread(sigi,dim=2,ncopies=ma)
 usav=u
call svdcmp(u,w,v)
 where \( u < \text{TOL} \times \text{maxval}(u) \) \( \rightarrow \text{w}=0 \)
 call svbksb(u,w,b,a)
 chisq=vabs(matmul(usav,a)-b)**2
 Evaluate chi-square.
END SUBROUTINE svdfit

SUBROUTINE svdvar(v,w,cvm)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: v
REAL(SP), DIMENSION(:,), INTENT(OUT) :: w
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: cvm

SUBROUTINE svdvar(v, w, cvm)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: v
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: w
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: cvm

\[ \chi^2 = \text{vabs}(\text{matmul}(\text{usav},a)-b)^2 \]
To evaluate the covariance matrix \( \text{cvm} \) of the fit for \( M \) parameters obtained by \texttt{svdfit}, call this routine with matrices \( v, w \) as returned from \texttt{svdfit}. The dimensions are \( M \) for \( v \) and \( M \times M \) for \( v \) and \( \text{cvm} \).

\[
\text{INTEGER(I4B)} :: \text{ma} \\
\text{REAL(SP), DIMENSION(size(w)) :: wti} \\
\text{ma=assert_eq((/size(v,1),size(v,2),size(w),size(cvm,1),size(cvm,2)/),'svdvar')} \\
\text{where (w /= 0.0)} \\
\text{wti=1.0_sp/(w*w)} \\
\text{elsewhere} \\
\text{wti=0.0} \\
\text{end where} \\
\text{cvm=v*spread(wti,dim=1,ncopies=ma)} \\
\text{cvm=matmul(cvm,transpose(v))} \\
\text{Covariance matrix is given by (15.4.20).} \\
\text{END SUBROUTINE svdvar}
\]

This is the standard Fortran 90 construction for doing different things to a matrix depending on some condition. Here we want to avoid inverting elements of \( w \) that are zero.

\[
\text{cvm=v*spread(wti,dim=1,ncopies=ma)} \\
\text{Each column of \( v \) gets multiplied by the corresponding element of \( wti \). Contrast the construction spread(...dim=2...) in \texttt{svdfit}}.
\]

\[
\text{FUNCTION fpoly(x,n)} \\
\text{USE nrtype; USE nrutil, ONLY : geop} \\
\text{IMPLICIT NONE} \\
\text{REAL(SP), INTENT(IN) :: x} \\
\text{INTEGER(I4B), INTENT(IN) :: n} \\
\text{REAL(SP), DIMENSION(n) :: fpoly} \\
\text{Fitting routine for a polynomial of degree \( n-1 \), returning \( n \) coefficients in \texttt{fpoly}.} \\
\text{fpoly=geop(1.0_sp,x,n)} \\
\text{END FUNCTION fpoly}
\]

\[
\text{FUNCTION fleg(x,nl)} \\
\text{USE nrtype} \\
\text{IMPLICIT NONE} \\
\text{REAL(SP), INTENT(IN) :: x} \\
\text{INTEGER(I4B), INTENT(IN) :: nl} \\
\text{REAL(SP), DIMENSION(nl) :: fleg} \\
\text{Fitting routine for an expansion with \( nl \) Legendre polynomials evaluated at \( x \) and returned in the array \texttt{fleg} of length \( nl \). The evaluation uses the recurrence relation as in \S\ 5.5.} \\
\text{INTEGER(I4B) :: j} \\
\text{REAL(SP) :: d,f1,f2,twox} \\
\text{fleg(1)=1.0} \\
\text{fleg(2)=x} \\
\text{if (nl > 2) then} \\
\text{twox=2.0_sp*x} \\
\text{f2=x} \\
\text{d=1.0} \\
\text{do j=3,nl} \\
\text{f1=d} \\
\text{f2=f2+twox} \\
\text{d=d+1.0_sp} \\
\text{end where} \\
\text{fleg=nl}
\]
\[ f_{\text{leg}}(j) = \frac{(f_2 \cdot f_{\text{leg}}(j-1) - f_1 \cdot f_{\text{leg}}(j-2))}{d} \]

end do

end if

END FUNCTION f_{\text{leg}}

* * *

SUBROUTINE mrqmin(x,y,sig,a,maska,covar,alpha,chisq,funcs,alamda)
USE nrtype; USE nrutil, ONLY : assert_eq, diagmult
USE nr, ONLY : covsrt, gaussj
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: covar,alpha
REAL(SP), INTENT(OUT) :: chisq
REAL(SP), INTENT(INOUT) :: alamda
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska

INTERFACE
SUBROUTINE funcs(x,a,yfit,dyda)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x,a
REAL(SP), DIMENSION(:), INTENT(OUT) :: yfit
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dyda
END SUBROUTINE funcs
END INTERFACE

Levenberg-Marquardt method, attempting to reduce the value $\chi^2$ of a fit between a set of $N$ data points $x,y$ with individual standard deviations sig, and a nonlinear function dependent on $M$ coefficients $a$. The input logical array maska of length $M$ indicates by true entries those components of $a$ that should be fitted for, and by false entries those components that should be held fixed at their input values. The program returns current best-fit values for the parameters $a$, and $\chi^2 = chisq$. The $M \times M$ arrays covar and alpha are used as working space during most iterations. Supply a subroutine funcs(x,a,yfit,dyda) that evaluates the fitting function $yfit$, and its derivatives $dyda$ with respect to the fitting parameters $a$ at $x$. On the first call provide an initial guess for the parameters $a$, and set alamda<0 for initialization (which then sets alamda=.001). If a step succeeds chi$^2$ becomes smaller and alamda decreases by a factor of 10. If a step fails alamda grows by a factor of 10. You must call this routine repeatedly until convergence is achieved. Then, make one final call with alamda=0, so that covar returns the covariance matrix, and alpha the curvature matrix. (Parameters held fixed will return zero covariances.)

INTEGER(I4B) :: ma,ndata
INTEGER(I4B), SAVE :: mfit
call mrqmin_private
CONTAINS
SUBROUTINE mrqmin_private
REAL(SP), SAVE :: ochisq
REAL(SP), DIMENSION(:,), ALLOCATABLE, SAVE :: atry,beta
REAL(SP), DIMENSION(:,,:), ALLOCATABLE, SAVE :: da
ndata=assert_eq(size(x),size(y),size(sig),'mrqmin: ndata')
ma=assert_eq((/size(a),size(maska),size(covar,1),size(covar,2),&
size(alpha,1),size(alpha,2)/),'mrqmin: ma')
mfit=count(maska)
if (alamda < 0.0) then
  Initialization.
  allocate(atry(ma),beta(ma),da(ma,1))
  alamda=.001_sp
  call mrqcof(a,alpha,beta)
  ochisq=chisq
  atry=a
end if

covar(1:mfit,1:mfit)=alpha(1:mfit,1:mfit)
call diagmult(covar(1:mfit,1:mfit),1.0_sp+alamda)
Alter linearized fitting matrix, by augmenting diagonal elements.

da(1:mfit,1)=beta(1:mfit)
call gaussj(covar(1:mfit,1:mfit),da(1:mfit,1:1))  
Matrix solution.

if (alamda == 0.0) then
  Once converged, evaluate covariance matrix.
call covsrt(covar,maska)
call covsrt(alpha,maska)
deallocate(atry,beta,da)
  Spread out alpha to its full size too.
  RETURN
end if

atry=a+unpack(da(1:mfit,1),maska,0.0_sp)
Did the trial succeed?
call mrqcof(atry,covar,da(1:mfit,1))
if (chisq < ochisq) then
  Success, accept the new solution.
  ochisq=chisq
  alpha(1:mfit,1:mfit)=covar(1:mfit,1:mfit)
  beta(1:mfit)=da(1:mfit,1)
  a=atry
else
  Failure, increase alamda and return.
  ochisq=alamda=10.0_sp*alamda
end if

END SUBROUTINE mrqmin_private

SUBROUTINE mrqcof(a,alpha,beta)
REAL(SP), DIMENSION(:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: beta
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: alpha

Used by mrqmin to evaluate the linearized fitting matrix alpha, and vector beta as in (15.5.8), and calculate $\chi^2$.

INTEGER(I4B) :: j,k,l,m
REAL(SP), DIMENSION(size(x),size(a)) :: dyda
REAL(SP), DIMENSION(size(x)) :: dy,sig2i,wt,ymod

call funcs(x,a,ymod,dyda)
Loop over all the data.
sig2i=1.0_sp/(sig**2)
dy=y-ymod
j=0
do l=1,ma
  if (maska(l)) then
    j=j+1
    wt=dyda(:,l)*sig2i
    k=0
do m=1,l
      if (maska(m)) then
        k=k+1
        alpha(j,k)=dot_product(wt,dyda(:,m))
        alpha(k,j)=alpha(j,k)
      end if
      k=k+1
    end do
    beta(j)=dot_product(dy,wt)
  end if
end do
chisq=dot_product(dy**2,sig2i)
Find $\chi^2$.
END SUBROUTINE mrqcof

END SUBROUTINE mrqmin

The organization of this routine is similar to that of amoeba, discussed on p. 1209. We want to keep the argument list of mrqcof to a minimum, but we want to make clear what global variables it accesses, and protect mrqmin_private's name space.

REAL(SP), DIMENSION(:,), ALLOCATABLE, SAVE :: atry,beta  
These arrays, as well as da, are allocated with the correct dimensions on the first call to mrqmin.
They need to retain their values between calls, so they are declared with the SAVE attribute. They get deallocated only on the final call when \( \text{alamda}=0 \).

\[
\text{call diagmult(...) \quad See discussion of diagadd after hqr on p. 1234.}
\]

\[
\text{atry=unpack(da(1:mfit,1),maska,0.0_sp) \quad maska controls which elements of a get incremented by da and which by 0.}
\]

⋆⋆⋆

SUBROUTINE fgauss(x,a,y,dyda)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,a
REAL(SP), DIMENSION(:), INTENT(OUT) :: y
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: dyda

\[ y(x;a) \] is the sum of \( N/3 \) Gaussians (15.5.16). Here \( N \) is the length of the vectors \( x, y \) and \( a \), while \( dyda \) is an \( N \times N \) matrix. The amplitude, center, and width of the Gaussians are stored in consecutive locations of \( a \):

\[
a(i) = B_k, \quad a(i+1) = E_k, \quad a(i+2) = G_k, \quad k = 1, \ldots, N/3.
\]

INTEGER(I4B) :: i,na,nx
REAL(SP), DIMENSION(size(x)) :: arg,ex,fac

\[
x=\text{assert_eq(size(x),size(y),size(dyda,1),’fgauss: nx’)}
\]

\[
a=\text{assert_eq(size(a),size(dyda,2),’fgauss: na’)}
\]

\[
y(:)=0.0
\]

\[
do i=1,na-1,3
\]

\[
\begin{align*}
\text{arg}(:)&=(x(:)-a(i+1))/a(i+2) \\
\text{ex}(:)&=\exp(-\text{arg}(:)**2) \\
\text{fac}(:)&=a(i)*\text{ex}(:)*2.0_{\text{sp}}*\text{arg}(:) \\
\text{y}(:)&=y(:)+a(i)*\text{ex}(:) \\
\text{dyda}(:,i)&=\text{ex}(:) \\
\text{dyda}(:,i+1)&=\text{fac}(:)/a(i+2) \\
\text{dyda}(:,i+2)&=\text{fac}(:)*\text{arg}(:)/a(i+2)
\end{align*}
\]

end do
END SUBROUTINE fgauss

⋆⋆⋆

SUBROUTINE medfit(x,y,a,b,abdev)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : select
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(OUT) :: a,b,abdev

Fits \( y=a+bx \) by the criterion of least absolute deviations. The same-size arrays \( x \) and \( y \) are the input experimental points. The fitted parameters \( a \) and \( b \) are output, along with \( abdev \), which is the mean absolute deviation (in \( y \)) of the experimental points from the fitted line.

INTEGER(I4B) :: ndata
REAL(SP) :: aa

\[
\text{call medfit_private}
\]
CONTAINS

SUBROUTINE medfit_private
IMPLICIT NONE
REAL(SP) :: b1,b2,bb,chi2sq,del,f1,f2,sigb,sx,sxx,sxy,sy
REAL(SP), DIMENSION(size(x)) :: tmp
ndata=\text{assert_eq(size(x),size(y),’medfit’)}

\[
sx=\text{sum}(x)
sy=\text{sum}(y)
sxy=\text{dot_product}(x,y)
\]

As a first guess for \( a \) and \( b \), we will find the least squares fitting line.
sxx=dot_product(x,x)

Least squares solutions.
del=ndata*sxx-sx**2

aa=(sxx*sy-sx*sxy)/del

bb=(ndata*sxy-sx*sy)/del

tmp(:)=y(:)-(aa+bb*x(:))

chisq=dot_product(tmp,tmp)

The standard deviation will give some idea of how
big an iteration step to take.
sigb=sqrt(chisq/del)

b1=bb

bb=bb+sign(3.0_sp*sigb,f1)

Guess bracket as 3-σ away, in the downhill direc-
tion known from f1.

f2=rofunc(b2)

if (b2 == b1) then
    a=aa
    b=bb
    RETURN
endif

do

Bracketing.

if (f1*f2 <= 0.0) exit
    bb=b2+1.6_sp*(b2-b1)
    b1=b2
    f1=f2
    b2=bb
    f2=rofunc(b2)
end do

sigb=0.01_sp*sigb

Refine until error a negligible number of standard
deviations.

if (abs(b2-b1) <= sigb) exit
    bb=b1+0.5_sp*(b2-b1)
    Bisection.
    if (bb == b1 .or. bb == b2) exit
    f=rofunc(bb)
    if (f*f1 >= 0.0) then
        f1=f
        b1=bb
    else
        f2=f
        b2=bb
    end if
end do

end if

a=aa
b=bb
abdev=abdev/ndata

FUNCTION rofunc(b)

Evaluates the right-hand side of equation (15.7.16) for a given value of b.

INTEGER(I4B) :: j

REAL(SP), DIMENSION(size(x)) :: arr,d

arr(:)=y(:)-b*x(:)

d(:)=y(:)-b*x(:)

if (mod(ndata,2) == 0) then
    j=ndata/2
    aa=0.5_sp*(select(j,arr)+select(j+1,arr))
else
    aa=select((ndata+1)/2,arr)
end if

where (y(:) /= 0.0) d(:)=d(:)/abs(y(:))

rofunc=sum(x(:)*sign(1.0_sp,d(:)), mask=(abs(d(:)) > EPS )

END SUBROUTINE medfit_private
The organization of this routine is similar to that of amoeba discussed on p. 1209. We want to keep the argument list of rofunc to a minimum, but we want to make clear what global variables it accesses and protect medfit_private's name space. In the Fortran 77 version, we kept the only argument as b by passing the global variables in a common block. This required us to make copies of the arrays x and y. An alternative Fortran 90 implementation would be to use a module with pointers to the arguments of medfit like x and y that need to be passed to rofunc. We think the medfit_private construction is simpler.
Chapter B16. Integration of Ordinary Differential Equations

SUBROUTINE rk4(y,dydx,x,h,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: y,dydx
REAL(SP), INTENT(IN) :: x,h
REAL(SP), DIMENSION(:,), INTENT(OUT) :: yout
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
INTEGER(I4B) :: ndum
REAL(SP) :: h6,hh,xh
REAL(SP), DIMENSION(size(y)) :: dym,dyt,yt
ndum=assert_eq(size(y),size(dydx),size(yout),'rk4')
hh=h*0.5_sp
h6=h/6.0_sp
xh=x+hh
yt=y+hh*dydx
First step.
call derivs(xh,yt,dyt)
Second step.
yt=y+hh*dyt
Third step.
yt=y+h*dym
dym=dyt+dym
Fourth step.
call derivs(x+h,yt,dym)
yout=y+h6*(dydx+dym*2.0_sp*dym)
Accumulate increments with proper weights.
END SUBROUTINE rk4

*   *   *

MODULE rkdumb_path
Storage of results.
USE nrtype
REAL(SP), DIMENSION(:,), ALLOCATABLE:: xx
REAL(SP), DIMENSION(:,), ALLOCATABLE :: y
END MODULE rkdumb_path
SUBROUTINE rkdumb(vstart,x1,x2,nstep,derivs)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : rk4
USE rkdumb_path
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: vstart
REAL(SP), INTENT(IN) :: x1,x2
INTEGER(I4B), INTENT(IN) :: nstep
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE

Starting from \( N \) initial values \( v_{\text{start}} \) known at \( x_1 \), use fourth-order Runge-Kutta to advance \( n_{\text{step}} \) equal increments to \( x_2 \). The user-supplied subroutine \( \text{derivs}(x,y,dydx) \) evaluates derivatives. Results are stored in the module variables \( xx \) and \( y \).

INTEGER(I4B) :: k
REAL(SP) :: h,x
REAL(SP), DIMENSION(size(vstart)) :: dv,v
v(:)=vstart(:)  
Load starting values.
if (allocated(xx)) deallocate(xx)  
Clear out old stored variables if necessary.
if (allocated(y)) deallocate(y)
allocate(xx(nstep+1))  
Allocate storage for saved values.
allocate(y(size(vstart),nstep+1))
y(:,1)=v(:)
xx(1)=x1  
x=x1
h=(x2-x1)/nstep  
Take \( n_{\text{step}} \) steps.
do k=1,nstep
    call derivs(x,v,dv)
    call rk4(v,dv,x,h,v,derivs)
    if (x+h == x) call nrerror('stepsize not significant in rkdumb')  
    x=x+h  
    xx(k+1)=x  
    Store intermediate steps.
y(:,k+1)=v(:)
end do
END SUBROUTINE rkdumb

This routine needs straightforward communication of arrays with the calling program. The dimension of the arrays is not known in advance, and if the routine is called a second time we need to throw away the old array information. The Fortran 90 construction for this is to declare allocatable arrays in a module, and then test them at the beginning of the routine with if (allocated...).
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE

Fifth order Runge-Kutta step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector $y$ and its derivative $dydx$ at the starting value of the independent variable $x$. Also input are the stepsize to be attempted $htry$, the required accuracy $eps$, and the vector $yscal$ against which the error is scaled. $y$, $dydx$, and $yscal$ are all of the same length. On output, $y$ and $x$ are replaced by their new values, $hdid$ is the stepsize that was actually accomplished, and $hnext$ is the estimated next stepsize. $derivs$ is the user-supplied subroutine that computes the right-hand-side derivatives.

INTEGER(I4B) :: ndum
REAL(SP) :: errmax,h,htemp,xnew
REAL(SP), DIMENSION(size(y)) :: yerr,ytemp
REAL(SP), PARAMETER :: SAFETY=0.9_sp,PGROW=-0.2_sp,PSHRNK=-0.25_sp,&
ERRCON=1.89e-4

The value $ERRCON$ equals $(5/SAFETY)^{(1/PGROW)}$, see use below.

ndum=assert_eq(size(y),size(dydx),size(yscal),'rkqs')
h=htry Set stepsize to the initial trial value.

do
  call rkck(y,dydx,x,h,yout,yerr,derivs) Take a step.
  errmax=maxval(abs(yerr(:)/yscal(:)))/eps Evaluate accuracy.
  if (errmax <= 1.0) exit Step succeeded.
  htemp=SAFETY*h*(errmax**PSHRNK) Truncation error too large, reduce stepsize.
  h=sign(max(abs(htemp),0.1_sp*abs(h)),h) No more than a factor of 10.
  xnew=x+h
  if (xnew == x) call nrerror('stepsize underflow in rkqs')
  end do

if (errmax > ERRCON) then Compute size of next step.
  hnext=SAFETY*h*(errmax**PGROW)
else
  hnext=5.0_sp*h
end if

hdid=h
x=x+h
y(:)=yout(:)
END SUBROUTINE rkqs

* * *

SUBROUTINE rkck(y,dydx,x,h,yout,yerr,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
REAL(SP), INTENT(IN) :: x,h
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout,yerr
END SUBROUTINE rkck

INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE

Given values for $N$ variables $y$ and their derivatives $dydx$ known at $x$, use the fifth order Cash-Karp Runge-Kutta method to advance the solution over an interval $h$ and return...
the incremented variables as yout. Also return an estimate of the local truncation error in yout using the embedded fourth order method. The user supplies the subroutine derivs(x,y,dydx), which returns derivatives dydx at x.

```
INTEGER(I4B) :: ndum
REAL(SP), DIMENSION(size(y)) :: ak2,ak3,ak4,ak5,ak6,ytemp
REAL(SP), PARAMETER :: A2=0.2_sp,A3=0.3_sp,A4=0.6_sp,A5=1.0_sp,&
A6=0.875_sp,B21=0.2_sp,B31=3.0_sp/40.0_sp,B32=9.0_sp/40.0_sp,&
B41=0.3_sp,B42=0.9_sp,B43=1.2_sp,B51=11.0_sp/54.0_sp,&
B52=2.5_sp,B53=70.0_sp/27.0_sp,B54=35.0_sp/27.0_sp,&
B61=1631.0_sp/B5296.0_sp,B62=175.0_sp/512.0_sp,&
B63=575.0_sp/13824.0_sp,B64=44275.0_sp/110592.0_sp,&
B65=253.0_sp/4096.0_sp,C1=37.0_sp/378.0_sp,&
C3=250.0_sp/621.0_sp,C4=125.0_sp/594.0_sp,&
C6=512.0_sp/1771.0_sp,DC1=C1-2825.0_sp/27648.0_sp,&
DC3=C3-18575.0_sp/48384.0_sp,DC4=C4-13525.0_sp/55296.0_sp,&
DC5=-277.0_sp/14336.0_sp,C6=6.25_sp

ndum=assert_eq(size(y),size(dydx),size(yout),size(yerr),'rkck')
```

```
ytemp=y+B21*h*dydx
First step.
call derivs(x+A2*h,ytemp,ak2)
Second step.
ytemp=y+h*(B31*dydx+B32*ak2)
Third step.
call derivs(x+A3*h,ytemp,ak3)
ytemp=y+h*(B41*dydx+B42*ak2+B43*ak3)
Fourth step.
call derivs(x+A4*h,ytemp,ak4)
ytemp=y+h*(B51*dydx+B52*ak2+B53*ak3+B54*ak4)
Fifth step.
call derivs(x+A5*h,ytemp,ak5)
ytemp=y+h*(B61*dydx+B62*ak2+B63*ak3+B64*ak4+B65*ak5)
Sixth step.
call derivs(x+A6*h,ytemp,ak6)
```

```
yout=y+h*(C1*dydx+C3*ak3+C4*ak4+C6*ak6)
Accumulate increments with proper weights.
yerr=h*(DC1*dydx+DC3*ak3+DC4*ak4+DC6*ak6)
Estimate error as difference between fourth and fifth order methods.
END SUBROUTINE rkck
```

```
* * *
```

```
MODULE ode_path
USE nrtype
INTEGER(I4B) :: nok,nbad,kount
LOGICAL(LGT), SAVE :: save_steps=.false.
REAL(SP) :: dxsav
REAL(SP), DIMENSION(:,), POINTER :: xp
REAL(SP), DIMENSION(:,,:), POINTER :: yp
END MODULE ode_path
```

```
SUBROUTINE odeint(ystart,x1,x2,eps,h1,hmin,derivs,rkqs)
USE nrtype; USE nrutil, ONLY : nrerror,reallocate
USE ode_path
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: ystart
REAL(SP), INTENT(IN) :: x1,x2,eps,h1,hmin
INTERFACE:
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs
SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: y
```

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. ... or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).
REAL(SP), DIMENSION(:,), INTENT(IN) :: dydx, yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry, eps
REAL(SP), INTENT(INOUT) :: hdid, hnext

INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE rkqs

REAL(SP), PARAMETER :: TINY = 1.0e-30_sp
INTEGER(I4B), PARAMETER :: MAXSTP = 10000

Runge-Kutta driver with adaptive stepsize control. Integrate the array of starting values ystart from x1 to x2 with accuracy eps, storing intermediate results in the module variables in odeint. h1 should be set as a guessed first step size, hmin as the minimum allowed stepsize (can be zero). On output ystart is replaced by values at the end of the integration interval. derivs is the user-supplied subroutine for calculating the right-hand-side derivative, while rkqs is the name of the stepper routine to be used.

INTEGER(I4B) :: nstp
REAL(SP) :: h, hdid, hnext, x, xsav
REAL(SP), DIMENSION(size(ystart)) :: dydx, y, yscal
x = x1
h = sign(h1, x2 - x1)
nok = 0
nbad = 0
kount = 0
y(:) = ystart(:)
nullify(xp, yp)
Pointers nullified here, but memory not deallocated. If odeint is called multiple times, calling program should deallocate xp and yp between calls.
if (save_steps) then
  xsav = x - 2.0_sp * dxsav
  allocate(xp(256))
  allocate(yp(size(ystart), size(xp)))
end if

do nstp = 1, MAXSTP
  Take at most MAXSTP steps.
call derivs(x, y, dydx)
yscal(:) = abs(y(:)) + abs(h * dydx(:)) + TINY
  Scaling used to monitor accuracy. This general purpose choice can be modified if need be.
  if (save_steps .and. (abs(x - xsav) > abs(dxsav))) & Store intermediate results.
    call save_a_step
call rkqs(y, dydx, x, h, eps, yscal, hdid, hnext, derivs)
  if (hdid == h) then
    nok = nok + 1
  else
    nbad = nbad + 1
  end if
  if ((ds2 * (x2 - x1)) > 0.0) h = x2 - x
  If stepsize can overshoot, decrease.
call rkqs(y, dydx, x, h, eps, yscal, hdid, hnext, derivs)
  if (hdid == h) then
    nok = nok + 1
  else
    nbad = nbad + 1
  end if
  if ((ds2 * (x2 - x1)) >= 0.0) then
    Are we done?
ystart(:) = y(:)
    if (save_steps) call save_a_step
    Save final step.
    RETURN
    Normal exit.
  end if
  if (abs(hnext) < hmin) &
    call nrerror('stepsize smaller than minimum in odeint')
  h = hnext
end do
call nrerror('too many steps in odeint')
CONTAINS

SUBROUTINE save_a_step
  kount=kount+1
  if (kount > size(xp)) then
    xp=>reallocate(xp,2*size(xp))
    yp=>reallocate(yp,size(yp,1),size(xp))
  end if
  xp(kount)=x
  yp(:,kount)=y(:,)
xsav=x
END SUBROUTINE save_a_step
END SUBROUTINE odeint

MODULE ode_path
  The situation here is similar to rkdumb_path, except we don’t know at run time how much storage to allocate. We may need to use reallocate from nrutil to increase the storage. The solution is pointers to arrays, with a nullify to be sure the pointer status is well-defined at the beginning of the routine.

SUBROUTINE save_a_step
  An internal subprogram with no arguments is like a macro in C: you could imagine just copying its code wherever it is called in the parent routine.

* * *

SUBROUTINE mmid(y,dydx,xs,htot,nstep,yout,derivs)
USE nrtype, USE nrutil, ONLY : assert_eq, swap
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), INTENT(IN) :: xs,htot
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
INTERFACE
  SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
  END SUBROUTINE derivs
END INTERFACE
Modified midpoint step. Dependent variable vector y and its derivative vector dydx are input at xs. Also input is htot, the total step to be taken, and nstep, the number of substeps to be used. The output is returned as yout, which need not be a distinct array from y; if it is distinct, however, then y and dydx are returned undamaged. y, dydx, and yout must all have the same length.

INTEGER(I4B) :: n,ndum
REAL(SP) :: h,h2,x
REAL(SP), DIMENSION(size(y)) :: ym,yn
ndum=assert_eq(size(y),size(dydx),size(yout),'mmid')
h=htot/nstep  Stepsize this trip.
y=y+h*dydx  First step.
x=x+h
call derivs(x,yn,yout)  Will use yout for temporary storage of derivatives.
h2=2.0_sp*h
do n=2,nstep  General step.
call swap(ym,yn)
y=yn+h2*yout
SUBROUTINE bsstep(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,cumsum,iminloc,nrerror,& outerdiff,outerprod,upper_triangle
USE nr, ONLY : mmid,pzextr
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
INTEGER(I4B), PARAMETER :: IMAX=9, KMAXX=IMAX-1
REAL(SP), PARAMETER :: SAFE1=0.25_sp,SAFE2=0.7_sp,REDMAX=1.0e-5_sp,& REDMIN=0.7_sp,TINY=1.0e-30_sp,SCALMX=0.1_sp
Bulirsch-Stoer step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector $y$ and its derivative $dydx$ at the starting value of the independent variable $x$. Also input are the stepsize to be attempted $htry$, the required accuracy $\varepsilon$, and the vector $yscal$ against which the error is scaled. On output, $y$ and $x$ are replaced by their new values, $hdid$ is the stepsize that was actually accomplished, and $hnext$ is the estimated next stepsize. $derivs$ is the user-supplied subroutine that computes the right-hand-side derivatives $y$, $dydx$, and $yscal$ must all have the same length. Be sure to set $htry$ on successive steps to the value of $hnext$ returned from the previous step, as is the case if the routine is called by $odeint$.
Parameters: $KMAXX$ is the maximum row number used in the extrapolation; $IMAX$ is the next row number; $SAFE1$ and $SAFE2$ are safety factors; $REDMAX$ is the maximum factor used when a stepsize is reduced, $REDMIN$ the minimum; $TINY$ prevents division by zero; $1/SCALMX$ is the maximum factor by which a stepsize can be increased.
INTEGER(I4B) :: k,kn,ndum
INTEGER(I4B), DIMENSION(IMAX) :: nseq = (/ 2,4,6,8,10,12,14,16,18 /)
INTEGER(I4B), SAVE :: kopt,kmax
REAL(SP), DIMENSION(KMAXX,KMAXX), SAVE :: alf
REAL(SP), DIMENSION(IMAX), SAVE :: a
REAL(SP), SAVE :: epsold = -1.0_sp,xnew
REAL(SP) :: eps1,errmax,fact,h,red,scale,wrkmin,xest
REAL(SP), DIMENSION(size(y)) :: yerr,ysav,yseq
LOGICAL(LGT) :: reduct
LOGICAL(LGT), SAVE :: first=.true.
ndum=assert_eq(size(y),size(dydx),size(yscal),'bsstep')
if (eps /= epsold) then 
  A new tolerance, so reinitialize.
  hnext=-1.0e29_sp
  "Impossible" values.
  xnew=-1.0e29_sp
  eps1=SAFE1*eps
  a(:)=cumsum(nseq,1)
  Compute $\alpha(k,q)$:
  where (upper_triangle(KMAXX,KMAXX)) alf=eps1** &
  (outerdiff(a(2:),a(2:))/outerprod(arth( &
3.0_sp,2.0_sp,KMAXX),(a(2:)-a(1)+1.0_sp))
epsold=eps
do kopt=2,KMAXX-1  Determine optimal row number for con-
if (a(kopt+1) > a(kopt)*af(kopt-1,kopt)) exit  vergence.
end do
kmax=kopt
end if
h=htry
ysav(:)=y(:)  Save the starting values.
if (h /= hnext .or. x /= xnew) then  A new stepsize or a new integration: Re-
first=.true.
end if
kopt=kmax
end if
reduct=.false.
main_loop: do
  do k=1,kmax  Evaluate the sequence of modified mid-
    xnew=x+h
    if (xnew == x) call nrerror('step size underflow in bstep')
    call mmid(ysav,dydx,x,h,nseq(k),yseq,derivs)
    xest=(h/nseq(k))**2  Squared, since error series is even.
    call pzextr(k,xest,yseq,y,yerr)  Perform extrapolation.
    if (k /= 1) then  Compute normalized error estimate \( \epsilon(k) \).
      errmax=maxval(abs(yerr(:)/yscal(:)))
      errmax=max(TINY,errmax)/eps  Scale error relative to tolerance.
      km=k-1
      err(km)=(errmax/SAFE1)**(1.0_sp/(2*km+1))
    end if
    if (k /= 1 .and. (k >= kopt-1 .or. first)) then  In order window.
      if (errmax < 1.0) exit main_loop  Converged.
      if (k == kmax .or. k == kopt+1) then  Check for possible step-
        red=SAFE2/err(km)  size reduction.
        exit
      else if (k == kopt) then
        if (af(kopt-1,kopt) < err(km)) then
          red=1.0_sp/err(km)
        end if
      else if (kopt == kmax) then
        if (af(km,kmax-1) < err(km)) then
          red=af(km,kmax-1)*SAFE2/err(km)
        end if
      else if (af(km,kopt) < err(km)) then
        red=af(km,kopt-1)/err(km)
      end if
    end if
  end do
  red=max(min(red,REDMIN),REDMAX)  Reduce steplsize by at least REDMIN and
  h=h*red  at most REDMAX.
  reduct=.true.
end do main_loop  Try again.
x=xnew  Successful step taken.
hdid=h
first=.false.
  kopt=1+imnloc(a(2:km+1)*max(err(1:km),SCALMX))  Compute optimal row for convergence and corresponding stepsize.
scale=max(err(kopt-1),SCALMX)
wrkmin=scale*a(kopt)
  hnext=scale/h
  if (kopt >= k .and. kopt /= kmax .and. not. reduct) then  Check for possible or-
    fact=max(scale/af(kopt-1,kopt),SCALMX)
    if (a(kopt+1)*fact <= wrkmin) then  der increase, but
      hnext=h/fact  not if stepsize was just reduced.
The function \( \text{cumsum} \) in \text{nrintool} with the optional argument \text{seed}=1 gives a direct implementation of equation (16.4.6).

where \((\text{upper_triangle}(KMAXX,KMAXX))\)... The \text{upper_triangle} function in \text{nrintool} returns an upper triangular logical mask. As used here, the mask is true everywhere in the upper triangle of a \( KMAXX \times KMAXX \) matrix, excluding the diagonal. An optional integer argument \text{extra} allows additional diagonals to be set to true. With \text{extra}=1 the upper triangle including the diagonal would be true.

Using a named do-loop provides clear structured code that required goto's in the Fortran 77 version.

See the discussion of \text{imaxloc} on p. 1017.

SUBROUTINE \( p_{\text{extr}}(i_{\text{est}},x_{\text{est}},y_{\text{est}},y_{\text{z}},y_{\text{dy}}) \)

Use polynomial extrapolation to evaluate \( N \) functions at \( x=0 \) by fitting a polynomial to a sequence of estimates with progressively smaller values \( x=x_{\text{est}} \), and corresponding function vectors \( y_{\text{est}} \). This call is number \text{iest} in the sequence of calls. Extrapolated function values are output as \( y_{\text{z}} \), and their estimated error is output as \( y_{\text{dy}} \). \( y_{\text{est}}, y_{\text{z}}, y_{\text{dy}} \) are arrays of length \( N \).

INTEGER(I4B), PARAMETER :: \text{IEST\_MAX}=16

INTEGER(I4B), SAVE :: \text{nvdold}=1

REAL(SP) :: \text{delta,fl,f2}

REAL(SP), DIMENSION(size(yz)) :: \text{d,tmp,q}

REAL(SP), DIMENSION(IEST\_MAX), SAVE :: \text{x}

REAL(SP), DIMENSION(:,,:), ALLOCATABLE, SAVE :: \text{qcol}

\text{nvd}=assert_eq(size(yz),size(yest),size(dy),'\text{p_{extr}}')

if (iest > \text{IEST\_MAX}) call \&
\text{nrerror('p_{extr}: probable misuse, too much extrapolation')}\n
if (nv /= \text{nvdold}) then
    Set up internal storage.
    if (allocated(qcol)) deallocate(qcol)
    allocate(qcol(nv,\text{IEST\_MAX}))
    \text{nvdold}=nv
end if

\text{x(iest)}=x_{\text{est}} \quad \text{Save current independent variable.}

\text{dy(:,)}=y_{\text{est}}(;)

\text{yz(:,)}=y_{\text{est}}(;)

\text{qc}(;,:)=y_{\text{est}}(;)

\text{else}
   \text{d(:,)}=y_{\text{est}}(;)
   do \text{j}=1,\text{iest}-1
      \text{delta}=1.0_sp/(\text{x(iest-j)}-x_{\text{est}})
      \text{f1}=x_{\text{est}}*\text{delta}
      \text{f2}=x(iest-j)*\text{delta}
      \text{q(:,)}=\text{qcol(:,j)}
   end if
END SUBROUTINE \( p_{\text{extr}} \)
SUBROUTINE pzextr

REAL(SP), DIMENSION(:,,:), ALLOCATABLE, SAVE :: qcol
The second dimension of qcol is known at compile time to be IEST_MAX, but the first
dimension is known only at run time, from size(yz). The language
requires us to have all dimensions allocatable if any one of them is.

if (nv /= nvold) then...
This routine generally gets called many times with
iest cycling repeatedly through the values 1, 2, ..., up to some value less than
IFEST_MAX. The number of variables, nv, is fixed during the solution of the problem.
The routine might be called again in solving a different problem with a new value
of nv. This if block ensures that qcol is dimensioned correctly both for the first
and subsequent problems, if any.

SUBROUTINE rzextr(iest,xest,yest,yz,dy)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: iest
REAL(SP), INTENT(IN) :: xest
REAL(SP), DIMENSION(:), INTENT(IN) :: yest
REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy

Exact substitute for pzextr, but uses diagonal rational function extrapolation instead of
polynomial extrapolation.

INTEGER(I4B), PARAMETER :: IEST_MAX=16
INTEGER(I4B) :: k,nv
INTEGER(I4B), SAVE :: nvold=-1
REAL(SP), DIMENSION(size(yz)) :: yy,v,c,b,b1,ddy
REAL(SP), DIMENSION(:,,:), ALLOCATABLE, SAVE :: d
REAL(SP), DIMENSION(IEST_MAX), SAVE :: fx,x
nv=assert_eq(size(yy),size(dy),size(yest),'rzextr')
if (iest > IEST_MAX) call &
nrerror('rzextr: probable misuse, too much extrapolation')
if (nv /= nvold) then
    if (allocated(d)) deallocate(d)
    allocate(d(nv,IFEST_MAX))
    nvold=nv
end if
Save current independent variable.
if (iest == 1) then
    yz=yest
d(:,1)=yest
    dy=yest
else
    fx(2:iest)=x(iest-1:1:-1)/xest
    yy=yest
    Evaluate next diagonal in tableau.
v=d(:,1)
c=yy
    d(:,1)=yy
do k=2,iest
        bl=fx(k)*v
        b=bl-c
        where (b /= 0.0)
b = (c - v) / b
ddy = c * b
c = b * 1
elsewhere
    ddy = v
end where
if (k /= iest)
    v = d(1:nv, k)
d(d(1:nv, k), k) = ddy
    yy = yy + ddy
end do
dy = ddy
yz = yy
end if
END SUBROUTINE rzextr

⋆⋆⋆

SUBROUTINE stoerm(y, d2y, x, htot, nstep, yout, derivs)

USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,), INTENT(IN) :: y, d2y
REAL(SP), INTENT(IN) :: x, htot
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), DIMENSION(:,), INTENT(OUT) :: yout
 INTERFACE
    SUBROUTINE derivs(x, y, dydx)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:,), INTENT(IN) :: y
        REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
END INTERFACE

Stoermer's rule for integrating $y'' = f(x, y)$ for a system of $n$ equations. On input $y$ contains $y$ in its first $n$ elements and $y'$ in its second $n$ elements, all evaluated at $x$. $d2y$ contains the right-hand-side function $f$ (also evaluated at $x$) in its first $n$ elements. Its second $n$ elements are not referenced. Also input is $htot$, the total step to be taken, and $nstep$, the number of substeps to be used. The output is returned as $yout$, with the same storage arrangement as $y$. $derivs$ is the user-supplied subroutine that calculates $f$.

INTEGER(I4B) :: neqn, neqn1, nn, nv
REAL(SP) :: h, h2, halfh, x
REAL(SP), DIMENSION(size(y)) :: ytemp
nv = assert_eq(size(y), size(d2y), size(yout), 'stoerm')
nv = nv / 2
neqn = nv / 2
Number of equations.
nv = nv + 1
h = htot / nstep
First step.
ytemp(neqn1:nv) = y(neqn1:nv) + halfh * d2y(1:neqn)
ytemp(1:neqn) = y(1:neqn) + ytemp(neqn1:nv)
x = x + h
Call derivs(x, ytemp, yout)
Use yout for temporary storage of derivatives.
h2 = h * h
General step.
ytemp(neqn1:nv) = ytemp(neqn1:nv) + h * yout(1:neqn)
ytemp(1:neqn) = ytemp(1:neqn) + ytemp(neqn1:nv)
x = x + h
Call derivs(x, ytemp, yout)
end do
yout(neqn1:nv) = ytemp(neqn1:nv) / h + halfh * yout(1:neqn)
Last step.
yout(1:neqn) = ytemp(1:neqn)
END SUBROUTINE stoerm

⋆⋆⋆
SUBROUTINE stiff(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq, diagadd, nrerror
USE nr, ONLY : lubksb, ludcmp
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx, yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry, eps
REAL(SP), INTENT(OUT) :: hdid, hnext
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
SUBROUTINE jacobn(x,y,dfdx,dfdy)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
END SUBROUTINE jacobn
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXTRY=40
REAL(SP), PARAMETER :: SAFETY=0.9_sp, GROW=1.5_sp, PGROW=-0.25_sp,
& SHRNK=0.5_sp, PSHRNK=-1.0_sp/3.0_sp, ERRCON=0.1296_sp,
& GAM=1.0_sp/2.0_sp, A21=2.0_sp, A2X=1.0_sp, A31=48.0_sp/25.0_sp, A3X=3.0_sp/5.0_sp,
& A32=6.0_sp/25.0_sp, C21=-8.0_sp, C31=372.0_sp/25.0_sp, C32=12.0_sp/5.0_sp,
& C41=-112.0_sp/125.0_sp, C42=-54.0_sp/125.0_sp, C43=-2.0_sp/5.0_sp,
& B1=19.0_sp/9.0_sp, B2=1.0_sp/2.0_sp, B3=25.0_sp/108.0_sp, B4=1.0_sp, E1=17.0_sp/54.0_sp,
& E2=7.0_sp/36.0_sp, E3=0.0_sp, E4=125.0_sp/108.0_sp, C2X=-3.0_sp/2.0_sp, C3X=121.0_sp/50.0_sp,
& C4X=29.0_sp/250.0_sp, A21=1.0_sp, A31=3.0_sp/5.0_sp
Fourth order Rosenbrock step for integrating stiff ODEs, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector y and its derivative dydx at the starting value of the independent variable x. Also input are the stepsize to be attempted htry, the required accuracy eps, and the vector yscal against which the error is scaled. On output, y and x are replaced by their new values, hdid is the stepsize that was actually accomplished, and hnext is the estimated next stepsize.

Parameters: GROW and SHRNK are the largest and smallest factors by which stepsize can change in one step; ERRCON=(GROW/SAFETY)**(1/PGROW) and handles the case when errmax ≃ 0.

INTEGER(I4B) :: ndum=assert_eq(size(y),size(dydx),size(yscal),'stiff')
ndum=assert_eq(size(y),size(dydx),size(yscal), 'stiff')
ndum=assert_eq(y, dydx, yscal, 'stiff')
xsav=x
ysav::=y(:)
call jacobn(xsav, ysav, dfdx, dfdy)
   The user must supply this subroutine to return the n x n matrix dfdy and the vector dfdx.
   h=htry
   Set stepsize to the initial trial value.
do jtry=1, MAXTRY

Set up the matrix $I - \gamma hf'$. 

LU decomposition of the matrix.

Set up right-hand side for $g_1$.

Solve for $g_1$.

Compute intermediate values of $y$ and $x$.

Compute dydx at the intermediate values.

Set up right-hand side for $g_2$.

Solve for $g_2$.

Compute intermediate values of $y$ and $x$.

Compute dydx at the intermediate values.

Set up right-hand side for $g_3$.

Solve for $g_3$.

Get fourth order estimate of $y$ and error estimate.

Evaluate accuracy.

Truncation error too large, reduce step size.

Go back and retry step.

END SUBROUTINE stiff

SUBROUTINE jacobn(x,y,dfdx,dfdy)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdx
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
SUBROUTINE used to obtain jacobian corresponding to example in (16.6.27).  
routine for Jacobi matrix corresponding to example in equations (16.6.27).

dfdx(:)=0.0

dfdy(1,1)=-0.013_sp-1000.0_sp*y(3)
dfdy(1,2)=0.0

dfdy(1,3)=-1000.0_sp*y(1)
dfdy(2,1)=0.0

dfdy(2,2)=-2500.0_sp*y(3)
dfdy(2,3)=-2500.0_sp*y(2)
dfdy(3,1)=-0.013_sp-1000.0_sp*y(3)
dfdy(3,2)=-2500.0_sp*y(2)
dfdy(3,3)=-1000.0_sp*y(1)-2500.0_sp*y(2)
END SUBROUTINE jacobn

SUBROUTINE diagadd(a,b)
call diagadd(a,1.0_sp/(GAM*h))
call ludcmp(a,indx,d)
g1=dydx+h*C1X*dfdx
call lubksb(a,indx,g1)
y=ysav+A1*g1+x=xsav+A2X*h
call derivs(x,y,dytmp)
g2=dytmp+h*C2X*dfdx+C21*g1/h
call lubksb(a,indx,g2)
y=ysav+A31*g1+32*g2+x=xsav+A3X*h
call derivs(x,y,dytmp)
g3=dytmp+h*C3X*dfdx+(C31*g1+C32*g2)/h
call lubksb(a,indx,g3)
y=ysav+E1*g1+E2*g2+E3*g3+E4*g4+g4
er=ysav+e1*g1+e2*g2+e3*g3+e4*g4+g4
x=xsav+h
if (x == xsav) call &
    nrerror('stepsize not significant in stiff')
errmax=maxval(abs(err)/yscal))/eps  Evaluate accuracy.
if (errmax <= 1.0) then  Step succeeded. Compute size of next step
    hnext=max(abs(hnext),SHRNK*abs(h)),h)
end if
else Truncation error too large. Reduce stepsize.
    hnext=SAFETY*h*errmax**PSHRNK
    h=sign(max(abs(hnext),SHRNK*abs(h)),h)
end if
call nrerror('exceeded MAXTRY in stiff')
END SUBROUTINE stiff

call diagadd(...) See discussion of diagadd after hqr on p. 1234.

SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx

Routine for right-hand side of example in equations (16.6.27).

dydx(1)=-0.013_sp*y(1)-1000.0_sp*y(1)*y(3)
dydx(2)=-2500.0_sp*y(2)*y(3)
dydx(3)=-0.013_sp*y(1)-1000.0_sp*y(1)*y(3)-2500.0_sp*y(2)*y(3)
END SUBROUTINE derivs

### SIMPR

SUBROUTINE simpr(y,dydx,dfdx,dfdy,xs,htot,nstep,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,diagadd
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xs,htot
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx,dfdx
REAL(SP), DIMENSION(:,:), INTENT(IN) :: dfdy
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout

INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE

Performs one step of semi-implicit midpoint rule. Input are the dependent variable
\( y \), its derivative \( dydx \), the derivative of the right-hand side with respect to \( x \), \( dfdx \), which are all vectors of length \( N \), and the \( N \times N \) Jacobian \( dfdy \) at \( xs \). Also input are \( htot \), the total step to be taken, and \( nstep \), the number of substeps to be used. The output is returned as
\( yout \), a vectro of length \( N \). \( derivs \) is the user-supplied subroutine that calculates \( dydx \).

INTEGER(I4B) :: ndum,nn
INTEGER(I4B), DIMENSION(size(y)) :: indx
REAL(SP) :: d,h,x
REAL(SP), DIMENSION(size(y)) :: del,ytemp
REAL(SP), DIMENSION(size(y),size(y)) :: a
ndum=assert_eq((/size(y),size(dydx),size(dfdx),size(dfdy,1),&
size(dfdy,2),size(yout)/),'simpr')
h=htot/nstep
a(:,:,)=h*dfdy(:,:,)
call diagadd(a,1.0_sp)
call ludcmp(a,indx,d)
yout=h*(dydx+h*dfdx)
call lubksb(a,indx,yout)
del=yout
ytemp=yout-del
x=x+h
call derivs(x,ytemp,yout)
do nn=2,nstep
  yout=h*yout-del
call lubksb(a,indx,yout)
del=del+2.0_sp*yout
  ytemp=yout-del
  x=x+h
call derivs(x,ytemp,yout)

END SUBROUTINE simpr
SUBROUTINE stifbs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,cumsum,iminloc,nrerror,&
outerdiff,outerprod,upper_triangle
USE nr, ONLY : simpr,pzextr
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(INOUT) :: y
REAL(SP), INTENT(OUT) :: hdid,hnext
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
SUBROUTINE jacobn(x,y,dfdx,dfdy)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: dfdy
END SUBROUTINE jacobn
END INTERFACE
INTEGER(I4B), PARAMETER :: IMAX=8, KMAXX=IMAX-1
REAL(SP), PARAMETER :: SAFE1=0.25_sp,SAFE2=0.7_sp,REDMAX=1.0e-5_sp,&
REDMIN=0.7_sp,TINY=1.0e-30_sp,SCALMX=0.1_sp
Semi-implicit extrapolation step for integrating stiff ODEs, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector $y$ and its derivative $dydx$ at the starting value of the independent variable $x$. Also input are the stepsize to be attempted $htry$, the required accuracy $eps$, and the vector $yscal$ against which the error is scaled. On output, $y$ and $x$ are replaced by their new values, $hdid$ is the stepsize that was actually accomplished, and $hnext$ is the estimated next stepsize. derivs is a user-supplied subroutine that computes the derivatives of the right-hand side with respect to $x$, while jacobn (a fixed name) is a user-supplied subroutine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of $y$, $dydx$, and $yscal$ must all have the same length. Be sure to set $htry$ on successive steps to the value of $hnext$ returned from the previous step, as is the case if the routine is called by odeint.
INTEGER(I4B) :: k,km,ndum
INTEGER(I4B), DIMENSION(IMAX) :: nseq = (/ 2,6,10,14,22,34,50,70 /)
Sequence is different from bsstep.
INTEGER(I4B), SAVE :: kopt,kmax,nvold=-1
REAL(SP), DIMENSION(KMAXX,KMAXX), SAVE :: alf
REAL(SP), DIMENSION(KMAXX) :: err
REAL(SP), DIMENSION(IMAX), SAVE :: a
REAL(SP), SAVE :: epsold = -1.0
REAL(SP) :: eps1,errmax,fact,h,red,scale,wrkmin,xest

end do
yout=h*yout-del
Set up right-hand side for last step.
call lubksb(a,indx,yout)
yout=ytemp+yout
Take last step.
END SUBROUTINE simpr
REAL(SP), SAVE :: xnew
REAL(SP) :: dfdx, yerr, ysav, yseq
REAL(SP) :: dfdy
LOGICAL(LGT) :: reduct
LOGICAL(LGT), SAVE :: first=.true.
ndum=assert_eq(size(y), size(dydx), size(yscal), 'stifbs')
if (eps /= epsold .or. nvold /= size(y)) then
    Reinitialize also if number of variables has changed.
end if
hnext=-1.0e29_sp
xnew=-1.0e29_sp
eps1=SAFE1*eps
a(:)=cumsum(nseq,1)
where (upper_triangle(KMAXX, KMAXX)) alf=eps1** &
(outerdiff(a(2:),a(2:))/outerprod(arth( &
3.0_sp,2.0_sp,KMAXX),(a(2:)-a(1)+1.0_sp)))
epsold=eps
nvold=size(y)
Save number of variables.
a(:)=cumsum(nseq,1+nvold)
Add cost of Jacobian evaluations to work coefficients.
do kopt=2,KMAXX-1
    if (a(kopt+1) > a(kopt)*alf(kopt-1,kopt)) exit
end do
kmax=kopt
end if
h=htry
ysav(:)=y(:)
call jacobn(x,y,dfdx,dfdy)
Evaluate Jacobian.
if (h /= hnext .or. x /= xnew) then
    first=.true.
kopt=kmax
end if
reduct=.false.
main_loop: do
    do k=1,kmax
        xnew=x+h
        if (xnew == x) call nrerror('step size underflow in stifbs')
call simp(yav,dydx,dfdx,dfdy,x,h,nseq(k),yseq,derivs)
The rest of the routine is identical to bsstep.
call pzextr(k,xest,yseq,y,yerr)
if (k /= 1 .and. (k >= kopt-1 .or. first)) then
    errmax=maxval(abs(yerr(:)/yscal(:)))
    errmax=max(TINY, errmax)/eps
    km=k-1
    err(km)=(errmax/SAFE1)**(1.0_sp/(2*km+1))
end if
if (k /= 1 .and. (k >= kopt-1 .or. first)) then
    if (errmax < 1.0) exit main_loop
if (k == kmax .or. k == kopt+1) then
    red=SAFE2/err(km)
    exit
else if (k == kopt) then
    if (alf(kopt-1,kopt) < err(km)) then
        red=1.0_sp/err(km)
        exit
    end if
else if (kopt == kmax) then
    if (alf(km,kmax-1) < err(km)) then
        red=alf(km,kmax-1)*SAFE2/err(km)
        exit
    end if
else if (alf(km,kopt) < err(km)) then
    red=alf(km,kopt-1)/err(km)
    exit
end if
end if
This routine is very similar to `bsstep`, and the same remarks about Fortran 90 constructions on p. 1305 apply here.
Chapter B17. Two Point Boundary Value Problems

! FUNCTION shoot(v) is named "funcv" for use with "newt"
FUNCTION funcv(v)
USE nrtype
USE nr, ONLY : odeint,rkqs
USE sshoot_caller, ONLY : nvar,x1,x2; USE ode_path, ONLY : xp,yp
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: v
REAL(SP), DIMENSION(size(v)) :: funcv
REAL(SP), PARAMETER :: EPS=1.0e-6_sp

Routine for use with newt to solve a two point boundary value problem for $N$ coupled ODEs by shooting from $x_1$ to $x_2$. Initial values for the ODEs at $x_1$ are generated from the $n_2$ input coefficients $v$, using the user-supplied routine load. The routine integrates the ODEs to $x_2$ using the Runge-Kutta method with tolerance EPS, initial stepsize $h_1$, and minimum stepsize $h_{min}$. At $x_2$ it calls the user-supplied subroutine score to evaluate the $n_2$ functions funcv that ought to be zero to satisfy the boundary conditions at $x_2$. The functions funcv are returned on output. newt uses a globally convergent Newton's method to adjust the values of $v$ until the functions funcv are zero. The user-supplied subroutine derivs(x,y,dydx) supplies derivative information to the ODE integrator (see Chapter 16). The module sshoot_caller receives its values from the main program so that funcv can have the syntax required by newt. Set nvar = $N$ in the main program.

REAL(SP) :: h1,hmin
REAL(SP), DIMENSION(nvar) :: y

INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
END SUBROUTINE derivs

SUBROUTINE load(x1,v,y)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1
REAL(SP), DIMENSION(:), INTENT(IN) :: v
REAL(SP), DIMENSION(:), INTENT(OUT) :: y
END SUBROUTINE load

SUBROUTINE score(x2,y,f)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x2
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: f
END SUBROUTINE score

END INTERFACE

h1=(x2-x1)/100.0_sp
FUNCTION shootf(v) is named "funcv" for use with "newt"

USE nr, ONLY : odeint, rkqs
USE sphfpt_caller, ONLY : x1, x2, xf, nn2; USE ode_path, ONLY : xp, yp

IMPLICIT NONE

REAL(SP), DIMENSION(:,), INTENT(IN) :: v
REAL(SP), DIMENSION(size(v)) :: funcv

REAL(SP), PARAMETER :: EPS = 1.0e-6_sp

Routine for use with newt to solve a two point boundary value problem for \( N \) coupled ODEs by shooting from \( x_1 \) and \( x_2 \) to a fitting point \( x_f \). Initial values for the ODEs at \( x_1 \) (\( x_2 \)) are generated from the \( n_2 \) (\( n_1 \)) coefficients \( V_1 \) (\( V_2 \)), using the user-supplied routine load1 (load2). The coefficients \( V_1 \) and \( V_2 \) should be stored in a single array \( v \) of length \( N \) in the main program, and referenced by pointers as \( v_1 \Rightarrow v(1:n_2) \), \( v_2 \Rightarrow v(n_2+1:N) \). Here \( N = n_1 + n_2 \). The routine integrates the ODEs to \( x_f \) using the Runge-Kutta method with tolerance \( EPS \), initial stepsize \( h_1 \), and minimum stepsize \( h_{min} \). If \( x_f \) it calls the user-supplied subroutine score to evaluate the \( N \) functions \( f_1 \) and \( f_2 \) that ought to match at \( x_f \). The differences \( funcv \) are returned on output. newt uses a globally convergent Newton's method to adjust the values of \( v \) until the functions \( funcv \) are zero. The user-supplied subroutine derivs(x,y,dydx) supplies derivative information to the ODE integrator (see Chapter 16). The module sphfpt_caller receives its values from the main program so that \( funcv \) can have the syntax required by \( newt \).

Set \( nn2 = n_2 \) in the main program.

REAL(SP) :: h1, hmin
REAL(SP), DIMENSION(size(v)) :: f1, f2, y

INTERFACE

SUBROUTINE derivs(x,y,dydx)
USE nr
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs

SUBROUTINE load1(x1,v1,y)
USE nr
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1
REAL(SP), DIMENSION(:,), INTENT(IN) :: v1
REAL(SP), DIMENSION(:,), INTENT(OUT) :: y
END SUBROUTINE load1

SUBROUTINE load2(x2,v2,y)
USE nr
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x2
REAL(SP), DIMENSION(:,), INTENT(IN) :: v2
REAL(SP), DIMENSION(:,), INTENT(OUT) :: y
END SUBROUTINE load2

SUBROUTINE score(x2,y,f)
USE nr
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x2
REAL(SP), DIMENSION(:,), INTENT(IN) :: y

!
REAL(SP), DIMENSION(:), INTENT(OUT) :: f
END SUBROUTINE score

END INTERFACE

h1=(x2-x1)/100.0_sp
hmin=0.0

if (associated(xp)) deallocate(xp,yp)  Prevent memory leak if
Path from x1 to xf with best trial values V1.
call load1(x1,v,y)

if (associated(xp)) deallocate(xp,yp)
Path from x2 to xf with best trial values V2.
call load2(x2,v(nn2+1:),y)

funcv(:)=f1(:)-f2(:)
END FUNCTION funcv

SUBROUTINE solvde(itmax,conv,slowc,scalv,indexv,nb,y)

Driver routine for solution of two point boundary value problems with N equations by
relaxation. itmax is the maximum number of iterations. conv is the convergence criterion
(see text). slowc controls the fraction of corrections actually used after each iteration.
scalv, a vector of length N, contains typical sizes for each dependent variable, used to
weight errors. indexv, also of length N, lists the column ordering of variables used to
construct the matrix s of derivatives. (The nb boundary conditions at the first mesh point
must contain some dependence on the first nb variables listed in indexv.) There are a total
of M mesh points. y is the N x M array that contains the initial guess for all the dependent
variables at each mesh point. On each iteration, it is updated by the calculated correction.

INTEGER(I4B), DIMENSION(size(scalv)) :: kmax
REAL(SP) :: err,fac
REAL(SP), DIMENSION(size(scalv)) :: ermax
REAL(SP), DIMENSION(size(scalv),size(scalv)-nb+1,size(y,2)+1) :: c
m=assert_eq(size(scalv),size(indexv),size(y,1),'solvde: ne')

Set up row and column markers.
k1=1
k2=m
nvars=nvars
j1=1
j2=nb
j3=nb+1
j4=ne
j5=j4+1
j6=j4+2
j7=j4+3
j8=j4+4
j9=j8+1
ic1=1
ic2=ne-nb
ic3=ic2+1
ic4=ne
ic5=1
jc1=ic3

Primary iteration loop.
k=1

Boundary conditions at first point.
call difeq(k,k1,k2,j9,ic3,ic4,indexv,s,y)  
call pinsv(ic3,ic4,j9,jk1,jc1,k1,c,s)  
do k=k1+1,k2  
Finite difference equations at all point 
kp=k-1  
pairs.  
call difeq(k1,k2,j9,ic1,ic4,indexv,s,y)  
call red(ic1,ic4,j1,j2,j3,j4,j9,jc1,jcf,kp,c,s)  
call pinsv(ic1,ic4,j3,j9,jc1,k1,c,s)  
end do  
k=k2+1  
Final boundary conditions.  
call difeq(k1,k2,j9,ic1,ic2,indexv,s,y)  
call red(ic1,ic2,j5,j6,j7,j8,j9,ic3,jcf,kp,c,s)  
call pinsv(ic1,ic2,j7,j9,jcf,kp+1,c,s)  
call bksub(ne,nb,jcf,k1,k2,c)  
Backsubstitution.  
do j=1,ne  
Convergence check, accumulate average 
jv=indexv(j)  
error.  
km=imaxloc(abs(c(jv,1,k1:k2)))+k1-1  
Find point with largest error, for each dependent variable.  
ermax(j)=c(jv,km)  
km(j)=km  
end do  
ermax(:)=ermax(:)/scalv(:)  
Weighting for each dependent variable.  
err=sum(sum(abs(c(indexv(:),1,k1:k2)),dim=2)/scalv(:))/nvars  
fac=slowc/max(slowc,err)  
Reduce correction applied when error is large.  
y(:,k1:k2)=y(:,k1:k2)-fac*c(indexv(:),1,k1:k2)  
Apply corrections.  
write(*,'(1x,i4,2f12.6)') it,err,fac  
Summary of corrections for this step. Point with largest error for each variable can be  
monitored by writing out km and ermax.  
if (err < conv) RETURN  
end do  
call nrerror('itmax exceeded in solvde')  
Convergence failed.

CONTAINS

SUBROUTINE bksub(ne,nb,jf,k1,k2,c)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ne,nb,jf,k1,k2
REAL(SP), DIMENSION(:,:,:), INTENT(INOUT) :: c

Backsubstitution, used internally by solvde.

INTEGER(I4B) :: im,k,nbf
nbf=ne-nb
im=1
do k=k2,k1,-1
  Use recurrence relations to eliminate remaining dependences.
  if (k == k1) im=nbf+1  
    Special handling of first point.
  c(im:ne,jf,k)=c(im:ne,jf,k)-matmul(c(im:ne,1:nbf,k),c(1:nbf,jf,k+1))  
end do  
c(1:nb,1,k1:k2)=c(1+nb:nbf+nb,jf,k1:k2)  
Reorder corrections to be in column 1.  
c(1+nb:nbf+nb,1,k1:k2)=c(1:nbf,jf,k1+1:k2+1)  
END SUBROUTINE bksub

SUBROUTINE pinsv(ic1,ic2,je1,je2,jsc1,k1,c,s)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ic1,ic2,je1,je2,jsc1,k1
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: c
REAL(SP) :: s

Diagonalize the square subsection of the s matrix, and store the recursion coefficients in  
c; used internally by solvde.

INTEGER(I4B) :: icoff,ipiv,jeoff,je2,je,jp,jpv,jpv,js
INTEGER(I4B), DIMENSION(ic2) :: indexr
REAL(SP) :: big,piv,pivinv
REAL(SP), DIMENSION(ic2) :: psc1  
je2=je1+1  
je1=je2+1
psc1(ic1:ic2)=maxval(abs(s(ic1:ic2,je1:je2)),dim=2)  
Implicit pivoting, as in §2.1.
if (any(pscl(ie1:ie2) == 0.0)) &
call nrerror(‘singular matrix, row all 0 in pinvs’)
pscl(ie1:ie2)=1.0_sp-pscl(ie1:ie2)
indxr(ie1:ie2)=0
do id=ie1,ie2
  piv=0.0
  do i=ie1,ie2
    Find pivot element.
    if (indxr(i) == 0) then
      jp=imaxloc(abs(s(i,je1:je2)))+je1-1
      big=abs(s(i,jp))
      if (big*pscl(i) > piv) then
        ipiv=i
        jpiv=jp
        piv=big*pscl(i)
      end if
    end if
  end do
  if (s(ipiv,jpiv) == 0.0) call nrerror(‘singular matrix in pinvs’)
  indxr(ipiv)=jpiv
  In place reduction. Save column ordering.
  pivinv=1.0_sp/s(ipiv,jpiv)
  s(ipiv,je1:jsf)=s(ipiv,je1:jsf)*pivinv
  Normalize pivot row.
  s(ipiv,jpiv)=1.0
  do i=ie1,ie2
    Reduce nonpivot elements in column.
    if (indxr(i) /= jpiv .and. s(i,jpiv) /= 0.0) then
      s(i,je1:jsf)=s(i,je1:jsf)-s(i,jpiv)*s(ipiv,je1:jsf)
      s(i,jpiv)=0.0
    end if
  end do
end do
jcoff=jc1-jsf
Sort and store unreduced coefficients.
icoff=ie1-je1
c(indxr(ie1:ie2)+icoff,jsf+jcoff:jsf+jcoff,k)=s(ie1:ie2,js1:jsf)
END SUBROUTINE pinvs

SUBROUTINE red(iz1,iz2,jz1,jz2,jm1,jm2,jmf,ic1,jc1,jcf,kc,c,s)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: iz1,iz2,jz1,jz2,jm1,jm2,jmf,ic1,jc1,jcf,kc
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: s
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: c

Reduce columns jz1-jz2 of the s matrix, using previous results as stored in the c matrix.
Only columns jm1-jm2,jmf are affected by the prior results. red is used internally by solvde.

INTEGER(I4B) :: ic,l,loff
loff=jc1-jm1
ic=ic1
do j=jz1,jz2
  Loop over columns to be zeroed.
do l=jm1,jm2
  Loop over columns altered.
    s(iz1:iz2,l)=s(iz1:iz2,l)-s(iz1:iz2,j)*c(ic,l+loff,kc)
    Loop over rows.
  end do
  s(iz1:iz2,jmf)=s(iz1:iz2,jmf)-s(iz1:iz2,j)*c(ic,jcf,kc)
  Plus final element.
  ic=ic+1
end do
END SUBROUTINE red

END SUBROUTINE solvde

km=imaxloc... See discussion of imaxloc on p. 1017.

*   *   *
MODULE sfroid_data
Communicates with difeq.
USE nrtype
USE nrutil, ONLY : arth
USE nr, ONLY : plgndr,solvde
USE sfroid_data
IMPLICIT NONE
INTEGER(4), PARAMETER :: M=41
INTEGER(4) :: mm,n
REAL(SP) :: anorm,c2,h
REAL(SP), DIMENSION(M) :: x
END MODULE sfroid_data

PROGRAM sfroid
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : plgndr,solvde
USE sfroid_data
IMPLICIT NONE
INTEGER(4), PARAMETER :: NE=3,NB=1
Sample program using solvde. Computes eigenvalues of spheroidal harmonics $S_{mn}(x;c)$ for $m \geq 0$ and $n \geq m$. In the program, $m$ is $mm$, $c^2$ is $c2$, and $\gamma$ of equation (17.4.20) is $anorm$.
INTEGER(4) :: itmax
INTEGER(4), DIMENSION(NE) :: indexv
REAL(SP) :: conv,slowc
REAL(SP), DIMENSION(M) :: deriv,fac1,fac2
REAL(SP), DIMENSION(NE) :: scalv
REAL(SP), DIMENSION(NE,M) :: y
itmax=100
conv=5.0e-6_sp
slowc=1.0
h=1.0_sp/(M-1)
c2=0.0
write(*,*) 'ENTER M,N'
read(*,*) mm,n
indexv(1:3)=merge( (/ 1, 2, 3 /), (/ 2, 1, 3 /), (mod(n+mm,2) == 1) )
No interchanges necessary if $n+mm$ is odd; otherwise interchange $y_1$ and $y_2$.
anorm=1.0
Compute $\gamma$.
if (mm /= 0) then
  anorm=(-0.5_sp)**mm*product(&
  arth(n+1,1,mm)*arth(real(n,sp),-1.0_sp,mm)/arth(1,1,mm))
end if
x(1:M-1)=arth(0,1,M-1)*h
fac1(1:M-1)=1.0_sp-x(1:M-1)**2
Compute initial guess.
fac2(1:M-1)=fac1(1:M-1)**(-mm/2.0_sp)
y(1,1:M-1)=plgndr(n,mm,x(1:M-1))*fac2(1:M-1) $P^n_{mm}$ from §6.8.
deriv(1:M-1)=-(n-mm+1)*plgndr(n+1,mm,x(1:M-1))-(n+1)*&
x(1:M-1)*plgndr(n,mm,x(1:M-1))/fac1(1:M-1)
Derivative of $P^n_{mm}$ from a recurrence relation.
y(2,1:M-1)=mm*x(1:M-1)y(1,1:M-1)/fac1(1:M-1)+deriv(1:M-1)*fac2(1:M-1)
y(3,1:M-1)=n*(n+1)-mm*(mm+1)
x(M)=1.0
Initial guess at $x = 1$ done separately.
y(1,M)=anorm
y(3,M)=n*(n+1)-mm*(mm+1)
y(2,M)=y(3,M)-c2)*y(1,M)/(2.0_sp*y(3,M))
scalv(1:3)=/( abs(anorm), max(abs(anorm),y(2,M)), max(1.0_sp,y(3,M)) )
do
write ('*',*) 'ENTER C**2 OR 999 TO END'
read ('*',*) c2
if (c2 /= 999.0) exit
call solvde(itmax,conv,slowc,scalv,indexv,NB,y)
write ('*',*) ' M = ',mm,' N = ',n,' C**2 = ',c2,' LAMBDA = ',y(3,1)+mm*(mm+1)
end do
Go back for another value of $c^2$.
END PROGRAM sfroid
MODULE sfroid_data

This module functions just like a common block to communicate variables with difeq. The advantage of a module is that it allows complete specification of the variables.

anorm=(-0.5_sp)**mm*product(...)

This statement computes equation (17.4.20) by direct multiplication.

* * *

SUBROUTINE difeq(k,k1,k2,jsf,is1,isf,indexv,s,y)
USE ntype
USE sfroid_data
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: is1,isf,jsf,k,k1,k2
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: indexv
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: s
REAL(SP), DIMENSION(:,), INTENT(IN) :: y

Returns matrix s(i,j) for solvde.
REAL(SP) :: temp,temp2
INTEGER(I4B), DIMENSION(3) :: indexv3

indexv3(1:3)=3+indexv(1:3)

if (k == k1) then
  Boundary condition at first point.
  if (mod(n+mm,2) == 1) then
    s(3,indexv3(1:3))= (/ 1.0_sp, 0.0_sp, 0.0_sp /)
    s(3,jsf)=y(1,1)
  else
    s(3,indexv3(1:3))= (/ 0.0_sp, 1.0_sp, 0.0_sp /)
    s(3,jsf)=y(2,1)
  end if
else if (k > k2) then
  Boundary conditions at last point.
  s(1,indexv3(1:3))= (/ -(y(3,M)-c2)/(2.0_sp*(mm+1.0_sp)),&
                  1.0_sp,-y(1,M)/(2.0_sp*(mm+1.0_sp)) /)
  s(1,jsf)=y(2,M)-(y(3,M)-c2)*y(1,M)/(2.0_sp*(mm+1.0_sp))
  s(2,indexv3(1:3))=(/ 1.0_sp, 0.0_sp, 0.0_sp /)
  s(2,jsf)=y(1,M)-anorm
  s(3,indexv3(1:3))=(/ 0.0_sp, 0.0_sp, -1.0_sp /)
  s(3,jsf)=y(3,M)
else
  Interior point.
  s(1,indexv(1:3))=(/ -1.0_sp, -0.5_sp*h, 0.0_sp /)
  s(1,indexv3(1:3))=(/ -1.0_sp, -0.5_sp*h, 0.0_sp /)
  temp=h/(1.0_sp-(x(k)+x(k-1))**2*0.25_sp)
  temp2=0.5_sp*(y(3,k)+y(3,k-1))-c2*0.25_sp*(x(k)+x(k-1))**2
  s(2,indexv(1:3))=(/ temp*temp2*0.5_sp,&
                   -1.0_sp-0.5_sp*temp*(mm+1.0_sp)*(x(k)+x(k-1)),&
                   0.25_sp*temp*(y(1,k)+y(1,k-1)) /)
  s(2,indexv3(1:3))=(/ temp*temp2*0.5_sp*&
                  (y(1,k)+y(1,k-1)) * (x(k)+x(k-1)) /&
                  0.25_sp*temp*(y(1,k)+y(1,k-1)) /)
  s(2,indexv3(1:3))=(/ temp*temp2*0.5_sp*&
                  (y(1,k)+y(1,k-1)) * (x(k)+x(k-1)) /&
                  0.25_sp*temp*(y(1,k)+y(1,k-1)) /)
  s(3,indexv(1:3))=(/ 0.0_sp, 0.0_sp, 1.0_sp /)
  s(3,jsf)=y(3,M)-y(3,k-1)
end if

END SUBROUTINE difeq

* * *
MODULE sphoot_data
Communicates with load, score, and deriv.
USE nrtype
INTEGER(4B) :: m,n
REAL(SP) :: c2,dx,gamma
END MODULE sphoot_data

MODULE sphoot_caller
Communicates with shoot.
USE nrtype
INTEGER(4B) :: nvar
REAL(SP) :: x1,x2
END MODULE sphoot_caller

PROGRAM sphoot
Sample program using shoot. Computes eigenvalues of spheroidal harmonics $S_{mn}(x; c)$ for $m \geq 0$ and $n \geq m$. Be sure that routine funcv for newt is provided by shoot (§17.1).
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : newt
USE sphoot_data
USE sphoot_caller
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: NV=3,N2=1
REAL(SP), DIMENSION(N2) :: v
LOGICAL(LGT) :: check
nvar=NV
Number of equations.
dx=1.0e-4_sp
Avoid evaluating derivatives exactly at $x = -1$.
do
write(*,*) 'input m,n,c-squared (999 to end)'
read(*,*) m,n,c2
if (c2 == 999.0) exit
if ((n < m) .or. (m < 0)) cycle
gamma=(-0.5_sp)**m*product(& Compute $\gamma$ of equation (17.4.20).
arth(n+1,1,m)*arth(real(n,sp),-1.0_sp,m)/arth(1,1,m))
v(1)=n*(n+1)-m*(m+1)+c2/2.0_sp
Initial guess for eigenvalue.
x1=-1.0_sp+dx
Set range of integration.
x2=0.0
call newt(v,check) Find $v$ that zeros function $f$ in score.
if (check) then
write(*,*)'shoot failed; bad initial guess' exit
else
write(*,'(1x,t6,a)') 'mu(m,n)' write(*,'(1x,f12.6)') v(1)
end if
end do
END PROGRAM sphoot

SUBROUTINE load(x1,v,y)
Supplies starting values for integration at $x = -1 + dx$.
USE nrtype
USE sphoot_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1
REAL(SP), DIMENSION(:), INTENT(IN) :: v
REAL(SP), DIMENSION(:), INTENT(OUT) :: y

y1=merge(gamma,-gamma, mod(n-m,2) == 0 )
y(3)=v(1)
y(2)=-(y(3)-c2)*y1/(2*(m+1))
y(1)=y1+y(2)*dx
END SUBROUTINE load
SUBROUTINE score(x2,y,f)
USE nrtype
USE sphoot_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x2
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: f

Tests whether boundary condition at \( x = 0 \) is satisfied.
\[ f(1) = \text{merge}(y(2), y(1), \text{mod}(n-m, 2) == 0) \]
END SUBROUTINE score

MODULE sphoot_data...MODULE sphoot_caller
These modules function just like common blocks to communicate variables from sphoot to the various subsidiary routines. The advantage of a module is that it allows complete specification of the variables.

SUBROUTINE derivs(x,y,dydx)
USE nrtype
USE sphoot_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
Evaluates derivatives for \textit{odeint}.
\[ dydx(1) = y(2) \]
\[ dydx(2) = (2.0 \times x \times (m+1.0) \times y(2) - (y(3) - c2 \times x \times x) \times y(1))/(1.0 - x \times x) \]
\[ dydx(3) = 0.0 \]
END SUBROUTINE derivs

* * *

MODULE sphfpt_data
Communicates with load1, load2, score, and derivs.
USE nrtype
INTEGER(I4B) :: m,n
REAL(SP) :: c2,dx,gamma
END MODULE sphfpt_data

MODULE sphfpt_caller
Communicates with shootf.
USE nrtype
INTEGER(I4B) :: nn2
REAL(SP) :: xi,x2,xf
END MODULE sphfpt_caller
PROGRAM sphfpt

Sample program using shootf. Computes eigenvalues of spheroidal harmonics \( S_{mn}(x; c) \)
for \( m \geq 0 \) and \( n \geq m \). Be sure that routine funcv for newt is provided by shootf (§17.2).

The routine derivs is the same as for sphoot.

USE nrtypes; USE nrtutil, ONLY : arth
USE nr, ONLY : newt
USE sphfpt_data
IMPLICIT NONE

INTEGER(I4B), PARAMETER :: N1=2,N2=1,NTOT=N1+N2
REAL(SP), PARAMETER :: DXX=1.0e-4_sp
REAL(SP), DIMENSION(:), POINTER :: v1,v2
REAL(SP), DIMENSION(NTOT), TARGET :: v
LOGICAL(LGT) :: check

v1=>v(1:N2)
v2=>v(N2+1:NTOT)
nn2=N2
dx=DXX

Avoid evaluating derivatives exactly at \( x = \pm 1 \).

do
  write(*,*) 'input m,n,c-squared (999 to end)',
  read(*,*) m,n,c2
  if (c2 == 999.0) exit
  if ((n < m) .or. (m < 0)) cycle
  gamma=(-0.5_sp)**m*product(&
    Compute \( \gamma \) of equation (17.4.20).
    arth(n+1,1,m)*(arth(real(n,sp),-1.0_sp,m)/arth(1,1,m)))
  v1(1)=n*(n+1)-m*(m+1)+c2/2.0_sp
    Initial guess for eigenvalue and function value.
  v2(2)=v1(1)
  v2(1)=gamma*(1.0_sp-(v2(2)-c2)*dx/(2*(m+1))
  x1=-1.0_sp+dx
  x2=1.0_sp-dx
  xf=0.0
  call newt(v,check)
    Find \( v \) that zeros function \( f \) in score.
  if (check) then
    write(*,*) 'shootf failed; bad initial guess'
    exit
  else
    write(*,'(1x,t6,a)') 'mu(m,n)'
    write(*,'(1x,f12.6)') v1(1)
  end if
end do

END PROGRAM sphfpt

SUBROUTINE load1(x1,v1,y)

USESphfpt_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1
REAL(SP), DIMENSION(:,), INTENT(IN) :: v1
REAL(SP), DIMENSION(:,), INTENT(OUT) :: y

Supplies starting values for integration at \( x = \pm 1 + dx \).

REAL(SP) :: y1

y(3)=v1(1)
y1=merge(gamma,-gamma,mod(n-m,2) == 0)
y(2)=-(y(3)-c2)*y1/(2*(m+1))
y(1)y1+y(2)*dx

END SUBROUTINE load1
SUBROUTINE load2(x2,v2,y)
USE nrtype
USE sphfpt_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x2
REAL(SP), DIMENSION(:), INTENT(IN) :: v2
REAL(SP), DIMENSION(:), INTENT(OUT) :: y
   Supplies starting values for integration at \( x = 1 - dx \).
y(3)=v2(2)
y(1)=v2(1)
y(2)=(y(3)-c2)*y(1)/(2*(m+1))
END SUBROUTINE load2

SUBROUTINE score(xf,y,f)
USE nrtype
USE sphfpt_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xf
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: f
   Tests whether solutions match at fitting point \( x = 0 \).
f(1:3)=y(1:3)
END SUBROUTINE score

MODULE sphfpt_data
...MODULE sphfpt_caller
   These modules function just like common blocks to communicate variables from sphfpt to the various subsidiary routines. The advantage of a module is that it allows complete specification of the variables.
Chapter B18. Integral Equations and Inverse Theory

SUBROUTINE fred2(a,b,t,f,w,g,ak)
USE nrtype; USE nrutil, ONLY : assert_eq,unit_matrix
USE nr, ONLY : gauleg,lubksb,ludcmp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(OUT) :: t,f,w
INTERFACE
FUNCTION g(t)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: t
REAL(SP), DIMENSION(size(t)) :: g
END FUNCTION g
FUNCTION ak(t,s)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: t,s
REAL(SP), DIMENSION(size(t),size(s)) :: ak
END FUNCTION ak
END INTERFACE
Solves a linear Fredholm equation of the second kind by \( N \)-point Gaussian quadrature. On input, \( a \) and \( b \) are the limits of integration. \( g \) and \( ak \) are user-supplied external functions. \( g \) returns \( g(t) \) as a vector of length \( N \) for a vector of \( N \) arguments, while \( ak \) returns \( \lambda K(t, s) \) as an \( N \times N \) matrix. The routine returns arrays \( t \) and \( f \) of length \( N \) containing the abscissas \( t_i \) of the Gaussian quadrature and the solution \( f \) at these abscissas. Also returned is the array \( w \) of length \( N \) of Gaussian weights for use with the Nyström interpolation routine \( fred2 \).

INTEGER(I4B) :: n
INTEGER(I4B), DIMENSION(size(f)) :: indx
REAL(SP) :: d
REAL(SP), DIMENSION(size(f),size(f)) :: omk
n=assert_eq(size(f),size(t),size(w),'fred2')
call gauleg(a,b,t,w)
call unit_matrix(omk)
call unit_matrix(omk)
omk=omk-ak(t,t)*spread(w,dim=1,ncopies=n)
f=g(t)
call ludcmp(omk,indx,d)
call lubksb(omk,indx,f)
END SUBROUTINE fred2

The unit_matrix routine in nrutil does exactly what its name suggests.
omk=omk-ak(t,t)*spread(w,dim=1,n copies=n)  By now this idiom should be second nature: the first column of \(ak\) gets multiplied by the first element of \(w\), and so on.

*     *     *

FUNCTION fredin(x,a,b,t,f,w,g,ak)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:,:), INTENT(IN) :: x,t,f,w
REAL(SP), DIMENSION(size(x)) :: fredin
INTERFACE
   FUNCTION g(t)
   USE nrtype
   IMPLICIT NONE
   REAL(SP), DIMENSION(:), INTENT(IN) :: t
   REAL(SP), DIMENSION(size(t)) :: g
   END FUNCTION g
   FUNCTION ak(t,s)
   USE nrtype
   IMPLICIT NONE
   REAL(SP), DIMENSION(:), INTENT(IN) :: t,s
   REAL(SP), DIMENSION(size(t),size(s)) :: ak
   END FUNCTION ak
END INTERFACE
Input are arrays \(t\) and \(w\) of length \(N\) containing the abscissas and weights of the \(N\)-point Gaussian quadrature, and the solution array \(f\) of length \(N\) from \texttt{fred2}. The function \texttt{fredin} returns the array of values of \(f\) at an array of points \(x\) using the Nystrom interpolation formula. On input, \(a\) and \(b\) are the limits of integration. \(g\) and \(ak\) are user-supplied external functions. \(g\) returns \(g(t)\) as a vector of length \(N\) for a vector of \(N\) arguments, while \(ak\) returns \(\lambda K(t,s)\) as an \(N \times N\) matrix.

INTEGER(I4B) :: n
n=assert_eq(size(f),size(t),size(w),'fredin')
fredin=g(x)+matmul(ak(x,t),w*f)
END FUNCTION fredin

Fredin=g(x)+matmul...  Fortran 90 allows very concise coding here, which also happens to be much closer to the mathematical formulation than the loops required in Fortran 77.

*     *     *

SUBROUTINE voltra(t0,h,t,f,g,ak)
USE nrtype; USE nrutil, ONLY : array_copy,assert_eq,unit_matrix
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: t0,h
REAL(SP), DIMENSION(:,:,), INTENT(INOUT) :: t
REAL(SP), DIMENSION(:,:,), INTENT(INOUT) :: f
INTERFACE
   FUNCTION g(t)
   USE nrtype
   IMPLICIT NONE
   REAL(SP), INTENT(IN) :: t
   REAL(SP), DIMENSION(:,), POINTER :: g
   END FUNCTION g
   FUNCTION ak(t,s)
   USE nrtype
   IMPLICIT NONE
   REAL(SP), DIMENSION(:,), POINTER :: g
   END FUNCTION ak(t,s)
FUNCTION g(t)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: t
REAL(SP), DIMENSION(:,:), POINTER :: g
END FUNCTION g

The routine `voltra` requires an argument that is a function returning a vector, but we don’t know the dimension of the vector at compile time. The solution is to make the function return a `pointer` to the vector. This is not the same thing as a pointer to a function, which is not allowed in Fortran 90. When you use the pointer in the routine, Fortran 90 figures out from the context that you want the vector of values, so the code remains highly readable. Similarly, the argument `ak` is a function returning a pointer to a matrix.

The coding of the user-supplied functions `g` and `ak` deserves some comment: functions returning pointers to arrays are potential memory leaks if the arrays are allocated dynamically in the functions. Here the user knows in advance the dimension of the problem, and so there is no need to use dynamical allocation in the functions. For example, in a two-dimensional problem, you can code `g` as follows:

```
FUNCTION g(t)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: t
REAL(SP), DIMENSION(:), POINTER :: g
REAL(SP), DIMENSION(2), TARGET, SAVE :: gg

  g=>gg
  g(1)=...
  g(2)=...
END FUNCTION g
```
and similarly for \( ak \).
Suppose, however, we coded \( g \) with dynamical allocation:

\[
\text{FUNCTION } g(t) \\
\text{USE nrtype} \\
\text{IMPLICIT NONE} \\
\text{REAL(SP), INTENT(IN) :: t} \\
\text{REAL(SP), DIMENSION(:), POINTER :: g} \\
\text{allocate(g(2))} \\
g(1)=... \\
g(2)=... \\
\text{END FUNCTION } g
\]

Now \( g \) never gets deallocated; each time we call the function fresh memory gets consumed. If you have a problem that really does require dynamical allocation in a pointer function, you have to be sure to deallocate the pointer in the calling routine. In \texttt{voltra}, for example, we would declare pointers \texttt{gtemp} and \texttt{aktemp}.

Then instead of writing simply

\[
b=g(t(i))+...
\]

we would write

\[
gtemp=>g(t(i)) \\
b=gtemp+... \\
\text{deallocate(gtemp)}
\]

and similarly for each pointer function invocation.

\[
\text{call array_copy(g(t(1)),f(:,1),ncop,nerr)}
\]

The purpose of using \texttt{array_copy} from \texttt{nrutil} is that we can check that \( f \) and \( g \) have consistent dimensions with a call to \texttt{assert_eq}.

\[
\star \quad \star \quad \star
\]

\[
\text{FUNCTION } \text{wwghts}(n,h,kermom) \\
\text{USE nrtype; USE nrutil, ONLY : geop} \\
\text{IMPLICIT NONE} \\
\text{INTEGER(I4B), INTENT(IN) :: n} \\
\text{REAL(SP), INTENT(IN) :: h} \\
\text{REAL(SP), DIMENSION(n) :: wwghts} \\
\text{INTERFACE} \\
\text{FUNCTION } kermom(y,m) \\
\text{USE nrtype} \\
\text{IMPLICIT NONE} \\
\text{REAL(DP), INTENT(IN) :: y} \\
\text{INTEGER(I4B), INTENT(IN) :: m} \\
\text{REAL(DP), DIMENSION(m) :: kermom} \\
\text{END FUNCTION } kermom \\
\text{END INTERFACE} \\
\text{END FUNCTION } \text{wwghts}
\]

\[
\text{INTEGER(I4B) :: j} \\
\text{REAL(DP), DIMENSION(4) :: wold,wnew,w} \\
\text{hh=hh} \\
h=1.0_dp/hh \\
wghts(1:n)=0.0 \\
wold(1:4)=kermom(0.0_dp,4)
\]

"Double precision on internal calculations even though the interface is in single precision."
"Zero all the weights so we can sum into them."
"Evaluate indefinite integrals at lower end."
if (n >= 4) then
  Use highest available order.
  b=0.0
  For another problem, you might change this lower
  limit.
  do j=1,n-3
    c=j-1
    a=b
    b=a+hh
    if (j == n-3) b=(n-1)*hh
    This is called $k$ in equation (18.3.5).
    do j=1,n-3
      c=j-1
      a=b
      Set upper and lower limits for this step.
      b=a+hh
      wnew(1:4)=kermom(b,4)
      Equation (18.3.4).
      w(1:4)= (/ wnew(1:4)-wold(1:4), geop(1.0_dp,hi,4) /)
      Equation (18.3.5).
      wwghts(j:j+3)=wwghts(j:j+3)+(/&
        ((c+1.0_dp)*(c+2.0_dp)*(c+3.0_dp)*w(1))&
        -3.0_dp*(c+c+1.0_dp)*(c+2.0_dp)*w(2)&
        +3.0_dp*(c+c+1.0_dp)*(c+3.0_dp)*w(3)&
        -w(4))/6.0_dp,&
        (-c*(c+2.0_dp)*(c+3.0_dp)*w(1))&
        -(2.0_dp+c*(6.0_dp+c+3.0_dp))*w(2)&
        +3.0_dp*(c+2.0_dp)*w(3)-w(4))/6.0_dp /)
    wold(1:4)=wnew(1:4)
    Reset lower limits for moments.
  end do
else if (n == 3) then
  Lower-order cases; not recommended.
  wnew(1:3)=kermom(hh+hh,3)
  w(1:3)= (/ wnew(1)-wold(1), hi*(wnew(2)-wold(2)), &
    hi**2*(wnew(3)-wold(3)) /)
  wwghts(1)=wnew(1)-wold(1)-wwghts(2)
  wwghts(2)=hi*(wnew(2)-wold(2))
  wwghts(3)=hi*(wnew(3)-wold(3))
  return
  end if
END FUNCTION wwghts

⋆⋆⋆

MODULE kermom_info
USE nrtype
REAL(DP) :: kermom_x
END MODULE kermom_info

FUNCTION kermom(y,m)
USE nrtype
USE kermom_info
IMPLICIT NONE
REAL(DP), INTENT(IN) :: y
INTEGER(I4B), INTENT(IN) :: m
REAL(DP), DIMENSION(m) :: kermom

Returns in kermom(1:m) the first m indefinite-integral moments of one row of the singular
part of the kernel. (For this example, m is hard-wired to be 4.) The input variable y labels
the column, while kermom_x (in the module kermom_info) is the row.

REAL(DP) :: x,d,df,clog,x2,x3,x4
x=kermom_x

FUNCTION kermom(y,m)
USE nrtype
USE kermom_info
IMPLICIT NONE
REAL(DP), INTENT(IN) :: y
INTEGER(I4B), INTENT(IN) :: m
REAL(DP), DIMENSION(m) :: kermom

Returns in kermom(1:m) the first m indefinite-integral moments of one row of the singular
part of the kernel. (For this example, m is hard-wired to be 4.) The input variable y labels
the column, while kermom_x (in the module kermom_info) is the row.

REAL(DP) :: x,d,df,clog,x2,x3,x4
x=kermom_x

We can take x as the lower limit of integration. Thus, we
if (y >= x) then
  return the moment integrals either purely to the left or
  d=x
  df=2.0_dp*sqrt(df)*d
  kermom(1:4) = (/ df/3.0_dp, df*(x/3.0_dp+d/5.0_dp), &
df*((x/3.0_dp + 0.4_dp*d)*x + d**2/7.0_dp),&
df*((x/3.0_dp + 0.6_dp*d)*x + 3.0_dp*d**2/7.0_dp)*x& + d**3/9.0_dp) /

else
x2=x**2
x3=x2*x
x4=x2*x2
d=x-y
clog=log(d)
kermom(1:4) = (/ d*(clog-1.0_dp),&
-0.25_dp*(3.0_dp*x+y-2.0_dp*clog*(x+y))*d,&
(-11.0_dp*x3+y*(6.0_dp*x2+y*(3.0_dp*x+2.0_dp*y))&
+6.0_dp*clog*(x3-y**3))/18.0_dp,&
(-25.0_dp*x4+y*(12.0_dp*x3+y*(6.0_dp*x2+y*&
(4.0_dp*x**3+y**3)))+12.0_dp*clog(x4-y**4))/48.0_dp /)
end if
END FUNCTION kermom

MODULE kermom_info
This module functions just like a common block to
share the variable kermom_x with the routine quadmx.

*   *   *

SUBROUTINE quadmx(a)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,diagadd,outerprod
USE nr, ONLY : wwghts,kermom
USE kermom_info
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: a
Constructs in the \( N \times N \) array \( a \) the quadrature matrix for an example Fredholm equation of
the second kind. The nonsingular part of the kernel is computed within this routine, while
the quadrature weights that integrate the singular part of the kernel are obtained via calls
to \( \text{wwghts} \). An external routine \( \text{kermom} \), which supplies indefinite-integral moments of the
singular part of the kernel, is passed to \( \text{wwghts} \).
INTEGER(I4B) :: j,n
REAL(SP) :: h,x
REAL(SP), DIMENSION(size(a,1)) :: wt
n=assert_eq(size(a,1),size(a,2),'quadmx')
h=PI/(n-1)
do j=1,n
x=(j-1)*h
kermom_x=x
Put \( x \) in the module kermom_info for use by kermom.
wt(:)=wwghts(n,h,kermom)
Part of nonsingular kernel.
a(j,:)=wt(:)
Put together all the pieces of the kernel.
end do
wt(:)=cos(arth(0.1,n)*h)
a(:,:)=a(:,:)*outerprod(wt(:),wt(:))
call diagadd(a,1.0_sp)
Since equation of the second kind, there is diagonal
piece independent of \( h \).
END SUBROUTINE quadmx

call diagadd... See discussion of diagadd after hqr on p. 1234.

*   *   *
PROGRAM fredex
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : quadmx,ludcmp,lubksb
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: N=40
INTEGER(I4B) :: j
INTEGER(I4B), DIMENSION(N) :: indx
REAL(SP) :: d
REAL(SP), DIMENSION(N) :: g,x
REAL(SP), DIMENSION(N,N) :: a

This sample program shows how to solve a Fredholm equation of the second kind using
the product Nyström method and a quadrature rule especially constructed for a particular,
singular, kernel.

Parameter: N is the size of the grid.
call quadmx(a) Make the quadrature matrix; all the action is here.
call ludcmp(a,indx,d) Decompose the matrix.
x(:)=arth(0,1,n)*PI/(n-1) Construct the right-hand side, here sin x.
g(:)=sin(x(:)) Backsubstitute.
call lubksb(a,indx,g) Write out the solution.
do j=1,n
   write (*,*) j,x(j),g(j)
end do
write (*,*) 'normal completion'
END PROGRAM fredex
Chapter B19. Partial Differential Equations

SUBROUTINE sor(a,b,c,d,e,f,u,rjac)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(DP), DIMENSION(:,,:), INTENT(IN) :: a,b,c,d,e,f
REAL(DP), DIMENSION(:,,:), INTENT(INOUT) :: u
REAL(DP), INTENT(IN) :: rjac
INTEGER(I4B), PARAMETER :: MAXITS=1000
REAL(DP), PARAMETER :: EPS=1.0e-5_dp
Successive overrelaxation solution of equation (19.5.25) with Chebyshev acceleration. a, b, c, d, e, and f are input as the coefficients of the equation, each dimensioned to the grid size $J \times J$. u is input as the initial guess to the solution, usually zero, and returns with the final value. rjac is input as the spectral radius of the Jacobi iteration, or an estimate of it. Double precision is a good idea for $J$ bigger than about 25.

REAL(DP), DIMENSION(size(a,1),size(a,1)) :: resid
INTEGER(I4B) :: jmax,jm1,jm2,jm3,n
REAL(DP) :: anorm,anormf,omega
jmax=assert_eq((/size(a,1),size(a,2),size(b,1),size(b,2), &
size(c,1),size(c,2),size(d,1),size(d,2),size(e,1), &
size(e,2),size(f,1),size(f,2),size(u,1),size(u,2)/),'sor')
jmax=jmax-1
jm1=jmax-1
jm2=jmax-2
jm3=jmax-3
anormf=sum(abs(f(2:jm1,2:jm1)))

Compute initial norm of residual and terminate iteration when norm has been reduced by a factor EPS. This computation assumes initial u is zero.

omega=1.0_dp

Do n=1,MAXITS

First do the even-even and odd-odd squares of the grid, i.e., the red squares of the checkerboard:

resid(2:jm1:2,2:jm1:2)=a(2:jm1:2,2:jm1:2)*u(3:jm1:2,3:jm1:2)+&
b(2:jm1:2,2:jm1:2)*u(1:jm2:2,2:jm1:2)+&
c(2:jm1:2,2:jm1:2)*u(2:jm1:2,3:jm1:2)+&
d(2:jm1:2,2:jm1:2)*u(2:jm1:2,1:jm2:2)+&
e(2:jm1:2,2:jm1:2)*u(2:jm1:2,2:jm1:2)-f(2:jm1:2,2:jm1:2)
u(2:jm1:2,2:jm1:2)=u(2:jm1:2,2:jm1:2)-omega*resid(2:jm1:2,2:jm1:2)/e(2:jm1:2,2:jm1:2)
resid(3:jm2:2,3:jm2:2)=a(3:jm2:2,3:jm2:2)*u(4:jm1:2,3:jm2:2)+&
b(3:jm2:2,3:jm2:2)*u(2:jm3:2,3:jm2:2)+&
c(3:jm2:2,3:jm2:2)*u(3:jm2:2,4:jm1:2)+&
d(3:jm2:2,3:jm2:2)*u(3:jm2:2,2:jm3:2)+&

omega=1.0_dp

Now do odd-odd and odd-even squares of the grid, i.e., the black squares of the checkerboard:

resid(3:jm2:2,2:jm1:2)=a(3:jm2:2,2:jm1:2)*u(4:jm1:2,2:jm1:2)+&
b(3:jm2:2,2:jm1:2)*u(2:jm3:2,2:jm1:2)+&
c(3:jm2:2,2:jm1:2)*u(3:jm2:2,4:jm1:2)+&
d(3:jm2:2,2:jm1:2)*u(3:jm2:2,2:jm3:2)+&
e(3:jm2:2,2:jm1:2)*u(3:jm2:2,3:jm1:2)-f(3:jm2:2,2:jm1:2)
u(3:jm2:2,2:jm1:2)=u(3:jm2:2,2:jm1:2)-omega*resid(3:jm2:2,2:jm1:2)/e(3:jm2:2,2:jm1:2)

End SUBROUTINE sor
Red-black iterative schemes like the one used in sor are easily parallelizable. Updating the red grid points requires information only from the black grid points, so they can all be updated independently. Similarly the black grid points can all be updated independently. Since nearest neighbors are involved in the updating, communication costs can be kept to a minimum.

There are several possibilities for coding the red-black iteration in a data parallel way using only Fortran 90 and no parallel language extensions.

One way is to define an \( N \times N \) logical mask \( \text{red} \) that is true on the red grid points and false on the black. Then each iteration consists of an update governed by a \( \text{where} (\text{red}) \ldots \text{end where} \) block and a \( \text{where}(.\text{not. red}) \ldots \text{end where} \) block. We have chosen a more direct coding that avoids the need for storage of the array \( \text{red} \). The red update corresponds to the even-even and odd-odd grid points, the black to the even-odd and odd-even points. We can code each of these four cases directly with array sections, as in the routine above.

The array section notation used in sor is rather dense and hard to read. We could use pointer aliases to try to simplify things, but since each array section is different, we end up merely giving names to each term that was there all along. Pointer aliases do help if we code sor using a logical mask. Since there may be machines on which this version is faster, and since it is of some pedagogic interest, we give the alternative code:

```fortran
SUBROUTINE sor_mask(a,b,c,d,e,f,u,rjac)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(DP), DIMENSION(:,,:), TARGET, INTENT(IN) :: a,b,c,d,e,f
REAL(DP), DIMENSION(:,,:), TARGET, INTENT(INOUT) :: u
REAL(DP), INTENT(IN) :: rjac
INTEGER(I4B), PARAMETER :: MAXITS=1000
REAL(DP), PARAMETER :: EPS=1.0e-5_dp
REAL(DP), DIMENSION(:,,:), ALLOCATABLE :: resid
REAL(DP), DIMENSION(:,,:), POINTER :: u_int,u_down,u_up,u_left,&
   u_right,a_int,b_int,c_int,d_int,e_int,f_int
INTEGER(I4B) :: jmax,jm1,jm2,jm3,n
REAL(DP) anorm,anormf,omega
LOGICAL, DIMENSION(:,,:), ALLOCATABLE :: red
jmax=assert_eq(/(size(a,1),size(a,2),size(b,1),size(b,2), &
size(c,1),size(c,2),size(d,1),size(d,2),size(e,1), &
size(e,2),size(f,1),size(f,2),size(u,1),size(u,2)/,'sor')

jm1=jmax-1
jm2=jmax-2
ja3=jmax-3
allocate(resid(jm2,jm2),red(jm2,jm2)) Interior is (jmax - 2) x (jmax - 2).
red=.false.
red(1:jm2,1:jm2)=.true.
red(2:jm3,2:jm3)=.true.
u_int=>u(2:jm1,2:jm1)
u_down=>u(3:jmax,2:jm1)
u_up=>u(1:jm2,2:jm1)
u_left=>u(2:jm1,1:jm2)
u_right=>u(2:jm1,3:jmax)
a_int=>a(2:jm1,2:jm1)
b_int=>b(2:jm1,2:jm1)
c_int=>c(2:jm1,2:jm1)
d_int=>d(2:jm1,2:jm1)
e_int=>e(2:jm1,2:jm1)
f_int=>f(2:jm1,2:jm1)
anormf=sum(abs(f_int))
omega=1.0

do n=1,MAXITS
where(red)
   resid=a_int*u_down+b_int*u_up+c_int*u_right+&
       d_int*u_left+e_int*u_int-f_int
   u_int=u_int-omega*resid/e_int
end where

omega=merge(1.0_dp/(1.0_dp-0.5_dp*rjac**2), &
             1.0_dp/(1.0_dp-0.25_dp*rjac**2*omega), n == 1)
where(.not.red)
   resid=a_int*u_down+b_int*u_up+c_int*u_right+&
       d_int*u_left+e_int*u_int-f_int
   u_int=u_int-omega*resid/e_int
end where

omega=1.0_dp/(1.0_dp-0.25_dp*rjac**2*omega)
anorm=sum(abs(resid))
if(anorm < EPS*anormf)exit
end do
deallocate(resid,red)
if (n > MAXITS) call nrerror('MAXITS exceeded in sor')

END SUBROUTINE sor_mask

* * *

SUBROUTINE mglin(u,ncycle)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : interp,rstrct,slvsml
IMPLICIT NONE
REAL(dp), DIMENSION(:,,:), INTENT(INOUT) :: u
INTEGER(INTEGER), INTENT(INOUT) :: ncycle
INTEGER(INTEGER), BINDING(:) :: j,jcycle,n,ng,ngrid,nn
TYPE ptr2d
   REAL(dp), POINTER :: a(:,:)
END TYPE ptr2d
TYPE(ptr2d), ALLOCATABLE :: rho(:)

Full Multigrid Algorithm for solution of linear elliptic equation, here the model problem (19.0.6). On input u contains the right-hand side \( \rho \) in an \( N \times N \) array, while on output it returns the solution. The dimension \( N \) is related to the number of grid levels used in the solution, \( N = 2**ng+1 \). ncycle is the number of V-cycles to be used at each level.

Define a type so we can have an array of pointers to arrays of grid variables.

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. Further reproduction of this file is strictly prohibited. To order Numerical Recipes books or CD-ROMs, visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).
REAL(DP), DIMENSION(:,::), POINTER :: uj,uj_1
n=assert_eq(size(u,1),size(u,2),'mglin')
ng=nint(log(n-1.0)/log(2.0))
if (n /= 2-ng+1) call nrerror('n-1 must be a power of 2 in mglin')
allocate(rho(ng))
nn=ng
ngrid=ng
allocate(rho(ngrid)%a(nn,nn))
do uj(ng)%a=u
allocate(rho(ngrid)%a(nn,nn))
 rho(ngrid)%a=rstrct(rho(ngrid+1)%a)
end do
allocate(uj(nn,nn))
call slvsml(uj,rho(1)%a)
do j=2,ng
 allocate(uj(nn,nn))
 uj_1=uj
 allocate(uj(nn,nn))
 uj=interp(uj_1)
deallocate(uj_1)
do jcycle=1,ncycle
 call mg(j,uj,rho(j)%a)
edo
u=uj
 end do
allocate(rho(ng))
do j=1,ng
 deallocate(rho)%a
end do
allocate(rho)
 end do
CONTAINS
RECURSIVE SUBROUTINE mg(j,u,rhs)
!...
The Fortran 90 version of mglin (and of mgfas below) is quite different from the Fortran 77 version, although the algorithm is identical. First, we use a recursive implementation. This makes the code much more transparent. It also makes the memory management much better: we simply define the new arrays res and v as automatic arrays of the appropriate dimension on each recursive call to a coarser level. And a third benefit is that it is trivial to change the code to increase the number of multigrid iterations done at level $j-1$ by each iteration at level $j$, i.e., to set the quantity $\gamma$ in §19.6 to a value greater than one. (Recall that $\gamma = 1$ as chosen in mglin gives V-cycles, $\gamma = 2$ gives W-cycles.) Simply enclose the recursive call in a do-loop:

```fortran
  do i=1,merge(gamma,1,j /= 2)
    call mg(j-1,v,res)
  end do
```

The merge expression ensures that there is no more than one call to the coarsest level, where the problem is solved exactly.

A second improvement in the Fortran 90 version is to make the procedures resid, interp, and rstrct functions instead of subroutines. This allows us to code the algorithm exactly as written mathematically.

**Type ptr2d...** The right-hand-side quantity $\rho$ is supplied initially on the finest grid in the argument $u$. It has to be defined on the coarser grids by restriction, and then supplied as the right-hand side to mg in the nested iteration loop. This loop starts at the coarsest level and progresses up to the finest level. We thus need a data structure to store $\rho$ on all the grid levels. A convenient way to implement this in Fortran 90 is to define a type ptr2d, a pointer to a two-dimensional array $a$ that represents a grid. (In three dimensions, $a$ would of course be three-dimensional.) We then declare the variable $\rho$ as an allocatable array of type ptr2d:

```fortran
  TYPE(ptr2d), ALLOCATABLE :: rho(:)
```

Next we allocate storage for $\rho$ on each level. The number of levels or grids, ng, is known only at run time:

```fortran
  allocate(rho(ng))
```

Then we allocate storage as needed on particular sized grids. For example,

```fortran
  allocate(rho(ngrid)%a(nn,nn))
```

allocates an $nn \times nn$ grid for $\rho$ on grid number $ngrid$.

The various subsidiary routines of mglin such as rstrct and interp are written to accept two-dimensional arrays as arguments. With the data structure we’ve employed, using these routines is simple. For example,

```fortran
  rho(ngrid)%a=rstrct(rho(ngrid+1)%a)
```

will restrict $\rho$ from the grid $ngrid+1$ to the grid $ngrid$. The statement is even more readable if we mentally ignore the $%a$ that is tagged onto each variable. (If
we actually did omit %a in the code, the compiler would think we meant the array of type ptr2d instead of the grid array.)

Note that while Fortran 90 does not allow you to declare an array of pointers directly, you can achieve the same effect by declaring your own type, as we have done with ptr2d in this example.

FUNCTION rstrct(uf)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,), INTENT(IN) :: uf
REAL(DP), DIMENSION((size(uf,1)+1)/2,(size(uf,1)+1)/2) :: rstrct
!
Half-weighting restriction. If \( N_c \) is the coarse-grid dimension, the fine-grid solution is input in the \((2N_c-1) \times (2N_c-1)\) array \( uf \), the coarse-grid solution is returned in the \( N_c \times N_c \) array \( rstrct \).
!
INTEGER(I4B) :: nc,nf
nf=assert_eq(size(uf,1),size(uf,2),'rstrct')
nc=(nf+1)/2
rstrct(2:nc-1,2:nc-1)=0.5_dp*uf(3:nf-2:2,3:nf-2:2)+0.125_dp*(
uf(4:nf-1:2,3:nf-2:2)+uf(2:nf-3:2,3:nf-2:2)+
uf(3:nf-2:2,4:nf-1:2)+uf(3:nf-2:2,2:nf-3:2))
INTERIOR points.
!
END FUNCTION rstrct

FUNCTION interp(uc)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,), INTENT(IN) :: uc
REAL(DP), DIMENSION(2*size(uc,1)-1,2*size(uc,1)-1) :: interp
!
Coarse-to-fine prolongation by bilinear interpolation. If \( N_f \) is the fine-grid dimension and \( N_c \) the coarse-grid dimension, then \( N_f = 2N_c - 1 \). The coarse-grid solution is input as \( uc \), the fine-grid solution is returned in \( interp \).
!
INTEGER(I4B) :: nc,nf
nc=assert_eq(size(uc,1),size(uc,2),'interp')
nf=2*nc-1
interp(1:nf,1:nf)=uc(1:nc,1:nc)
Do elements that are copies.
interp(2:nf-1:2,1:nf)=0.5_dp*(interp(3:nf:2,1:nf)+
interp(1:nf-2:2,1:nf))
Do odd-numbered columns, interpolating vertically.
interp(1:nf,2:nf-1:2)=0.5_dp*(interp(1:nf,3:nf:2)+interp(1:nf,1:nf-2:2))
Do even-numbered columns, interpolating horizontally.
END FUNCTION interp

SUBROUTINE slvsm1(u,rhs)
USE nrtype
IMPLICIT NONE
REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
!
Solution of the model problem on the coarsest grid, where \( h = 1/2 \). The right-hand side is input in \( rhs(1:3,1:3) \) and the solution is returned in \( u(1:3,1:3) \).
!
REAL(DP) :: h
u=0.0
h=0.5_dp
u(2,2)=h*rhs(2,2)/4.0_dp
END SUBROUTINE slvsm1
SUBROUTINE relax(u,rhs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,:,:), INTENT(IN) :: rhs

Red-black Gauss-Seidel relaxation for model problem. The current value of the solution \( u \) is updated, using the right-hand-side function \( rhs \). \( u \) and \( rhs \) are square arrays of the same odd dimension.

INTEGER(I4B) :: n
REAL(DP) :: h,h2
n=assert_eq(size(u,1),size(u,2),size(rhs,1),size(rhs,2),'relax')
h=1.0_dp/(n-1)
h2=h*h

First do the even-even and odd-odd squares of the grid, i.e., the red squares of the checkerboard:
\[
\begin{align*}
    u(2:n-1:2,2:n-1:2) &= 0.25_dp*(u(3:n,2:n-1:2)+u(1:n-2:2,2:n-1:2)+u(2:n-1:2,3:n)+u(2:n-1:2,1:n-2:2)-h2*rhs(2:n-1:2,2:n-1:2)) \\
    u(3:n-2:2,3:n-2:2) &= 0.25_dp*(u(4:n-1:2,3:n-2:2)+u(2:n-3:2,3:n-2:2)+u(3:n-2:2,4:n-1:2)+u(3:n-2:2,2:n-3:2)-h2*rhs(3:n-2:2,3:n-2:2))
\end{align*}
\]

Now do even-odd and odd-even squares of the grid, i.e., the black squares of the checkerboard:
\[
\begin{align*}
    u(3:n-2:2,2:n-1:2) &= 0.25_dp*(u(4:n-1:2,2:n-1:2)+u(2:n-3:2,2:n-1:2)+u(3:n-2:2,3:n)+u(3:n-2:2,1:n-2:2)-h2*rhs(3:n-2:2,2:n-1:2)) \\
    u(2:n-1:2,3:n-2:2) &= 0.25_dp*(u(3:n,3:n-2:2)+u(1:n-2:2,3:n-2:2)+u(2:n-1:2,4:n-1:2)+u(2:n-1:2,2:n-3:2)-h2*rhs(2:n-1:2,3:n-2:2))
\end{align*}
\]
END SUBROUTINE relax

FUNCTION resid(u,rhs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: u,rhs
REAL(DP), DIMENSION(size(u,1),size(u,1)) :: resid

INTEGER(I4B) :: n
REAL(DP) :: h,h2i
n=assert_eq((/size(u,1),size(u,2),size(rhs,1),size(rhs,2)/),'resid')
n=size(u,1)
h=1.0_dp/(n-1)
h2i=1.0_dp/(h*h)

resid(2:n-1,2:n-1)=-h2i*(u(3:n,2:n-1)+u(1:n-2,2:n-1)+u(2:n-1,3:n)+u(2:n-1,1:n-2)-4.0_dp*u(2:n-1,2:n-1))+rhs(2:n-1,2:n-1)

resid(1:n,1)=0.0
resid(1:n,n)=0.0
resid(1,1:n)=0.0
resid(n,1:n)=0.0
END FUNCTION resid

\[ \star \star \star \]
SUBROUTINE mgfas(u,maxcyc)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : interp,lop,rstrct,slvsm2
IMPLICIT NONE
REAL(DP), DIMENSION(:,,:), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: maxcyc

Full Multigrid Algorithm for FAS solution of nonlinear elliptic equation, here equation (19.6.44). On input \( u \) contains the right-hand side \( \rho \) in an \( N \times N \) array, while on output it returns the solution. The dimension \( N \) is related to the number of grid levels used in the solution, \( ng \) below, by \( N = 2^{ng+1} \). \( maxcyc \) is the maximum number of V-cycles to be used at each level.

INTEGER(I4B) :: j,jcycle,n,ng,ngrid,nn
REAL(DP) :: res,trerr

TYPE ptr2d
  D e f i n e a  t y p e s o w e c a n h a v e a n a r r a y o f
  p o i n t e r s t o a r r a y s o f g r i d v a r i a b l e s .
END TYPE ptr2d

TYPE(ptr2d), ALLOCATABLE :: rho(:)
REAL(DP), DIMENSION(:,,:), POINTER :: uj,uj_1
n=assert_eq(size(u,1),size(u,2),'mgfas')
ng=nint(log(n-1.0)/log(2.0))
if (n /= 2**ng+1) call nrerror('n-1 must be a power of 2 in mgfas')
allocation(rho(ng))
n=nn-1
rho(ng)%a=u
and fill it with \( \rho \) from the fine grid.
do
  nn=nn/2+1
  ngrid=ngrid-1
  allocate(rho(ngrid)%a(nn,nn))
rho(ngrid)%a=rstrct(rho(ngrid+1)%a)
end do
nn=3
allocate(uj(nn,nn))
call slvsm2(uj,rho(1)%a)
do j=2,ng
  Initial solution on coarsest grid.
  allocate(uj(j,nn,nn))
  call rstrct(uj(j,1)%a)
  interpolate from grid \( j-1 \) to next finer grid
  do jcycle=1,maxcyc
    V-cycle loop.
      call mg(j,uj,trerr=trerr)
      res=sqrt(sum((lop(uj)-rho(j)%a)**2))/nn
      Form residual \( \|d_h\| \).
      if (res < trerr) exit
    end do
  end do
  deallocate(uj)
do j=1,ng
  deallocate(rho)
end do
CONTAINS

RECURSIVE SUBROUTINE mg(j,u,rhs,trerr)
USE nrtype
USE nr, ONLY : interp,lop,relax2,rstrct,slvsm2
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: j
REAL(DP), DIMENSION(:,,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,,:), INTENT(IN), OPTIONAL :: rhs
REAL(DP), INTENT(OUT), OPTIONAL :: trerr
INTEGER(I4B), PARAMETER :: NPRE=1, NPOST=1
REAL(DP), PARAMETER :: ALPHA=0.33_dp

Recursive multigrid iteration. On input, \( j \) is the current level and \( u \) is the current value of the solution. For the first call on a given level, the right-hand side is zero, and the optional argument \( \text{rhs} \) is not present. Subsequent recursive calls supply a nonzero \( \text{rhs} \) as in equation (19.6.33). On output \( u \) contains the improved solution at the current level. When the first call on a given level is made, the relative truncation error \( \tau \) is returned in the optional argument \( \text{trerr} \).

Parameters: \( \text{NPRE} \) and \( \text{NPOST} \) are the number of relaxation sweeps before and after the coarse-grid correction is computed; \( \text{ALPHA} \) relates the estimated truncation error to the norm of the residual.

INTEGER(I4B) :: jpost, jpre
REAL(DP), DIMENSION((size(u,1)+1)/2,(size(u,1)+1)/2) :: v, ut, tau

if (j == 1) then
  Bottom of V: Solve on coarsest grid.
call slvsms2(u, rhs+rho(j)%a)
else
  On downward stroke of the V.
  Pre-smoothing.
do jpre=1, NPRE
    if (present(rhs)) then
      call relax2(u, rhs+rho(j)%a)
    else
      call relax2(u, rho(j)%a)
    end if
  end do
  ut = rstrct(u)
  v = ut
  if (present(rhs)) then
    Form \( \tilde{\tau}_h + f_H = \mathcal{L}_H(\mathcal{R}\tilde{u}_h) - \mathcal{R}\mathcal{L}_H(\tilde{u}_h) + f_H \)
    else
    tau = lop(ut) - rstrct(lop(u))
    end if
  end do
  \( \text{trerr} = \text{ALPHA} \times \text{sqrt}(\text{sum}(\tau^2))/\text{size}(\tau,1) \)
  Estimate truncation error \( \tau \).
call mg(j-1, v, tau)
u = u + interp(v-ut)
do jpost=1, NPOST
    if (present(rhs)) then
      call relax2(u, rhs+rho(j)%a)
    else
      call relax2(u, rho(j)%a)
    end if
  end do
end if
END SUBROUTINE mg
END SUBROUTINE mgfas

See the discussion after mglin on p. 1336 for the changes made in the Fortran 90 versions of the multigrid routines from the Fortran 77 versions.

TYPE ptr2d...

Recursive SUBROUTINE mg(j, u, rhs, trerr)
Recall that mgfas solves the problem \( Lu = 0 \), but that nonzero right-hand sides appear during the solution. We implement this by having \( \text{rhs} \) be an optional argument to mg. On the first call at a given level \( j \), the right-hand side is zero and so you just omit it from the calling sequence. On the other hand, the truncation error \( \text{trerr} \) is computed only on the first call at a given level, so it is also an optional argument that does get supplied on the first call:

\[
call \text{mg}(j, u, \text{trerr} = \text{trerr})
\]

The second and subsequent calls at a given level supply \( \text{rhs} = \tau \) but omit \( \text{trerr} \):
call mg(j-1,v,tau)

Note that we can omit the keyword rhs from this call because the variable tau appears in the correct order of arguments. However, in the other call above, the keyword trerr must be supplied because rhs has been omitted.

The example equation that is solved in mgfas, equation (19.6.44), is almost linear, and the code is set up so that $\rho$ is supplied as part of the right-hand side instead of pulling it over to the left-hand side. The variable rho is visible to mg by host association. Note also that the function lop does not include rho, but that the statement

$$tau=lo{p}(ut)-rstrct(lo{p}(u))$$

is nevertheless correct, since rho would cancel out if it were included in lop. This feature is also true in the Fortran 77 code.

SUBROUTINE relax2(u,rhs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,,:), INTENT(IN) :: rhs
INTEGER(I4B) :: n
REAL(DP) :: foh2,h,h2i
REAL(DP) :: res(size(u,1),size(u,1))
n=assert_eq(size(u,1),size(u,2),size(rhs,1),size(rhs,2),'relax2')
h=1.0_dp/(n-1)
h2i=1.0_dp/(h*h)
foh2=-4.0_dp*h2i
First do the even-even and odd-odd squares of the grid, i.e., the red squares of the checkerboard:
$$res(2:n-1:2,2:n-1:2)=h2i*(u(3:n:2,2:n-1:2)+u(1:n-2:2,2:n-1:2)+u(2:n-1:2,1:n-2:2)+u(2:n-1:2,2:n-2:2)-4.0_dp*u(2:n-1:2,2:n-1:2))$$
$$u(2:n-1:2,2:n-1:2)=u(2:n-1:2,2:n-1:2)-res(2:n-1:2,2:n-1:2)/(foh2+2.0_dp*u(2:n-1:2,2:n-1:2))$$
$$res(3:n-2:2,3:n-2:2)=h2i*(u(4:n-1:2,3:n-2:2)+u(2:n-3:2,3:n-2:2)+u(3:n-2:2,4:n-1:2)+u(3:n-2:2,2:n-3:2)-4.0_dp*u(3:n-2:2,3:n-2:2))$$
Now do even-odd and odd-even squares of the grid, i.e., the black squares of the checkerboard:
$$res(3:n-2:2,2:n-1:2)=h2i*(u(4:n-1:2,2:n-1:2)+u(2:n-3:2,2:n-1:2)+u(3:n-2:2,1:n-2:2)+u(3:n-2:2,2:n-2:2)-4.0_dp*u(3:n-2:2,2:n-1:2))$$
$$u(3:n-2:2,2:n-1:2)=u(3:n-2:2,2:n-1:2)-res(3:n-2:2,2:n-1:2)/(foh2+2.0_dp*u(3:n-2:2,2:n-1:2))$$
$$res(2:n-1:2,3:n-2:2)=h2i*(u(3:n-2:2,3:n-2:2)+u(1:n-2:2,3:n-2:2)+u(2:n-1:2,4:n-1:2)+u(2:n-1:2,3:n-2:2)-4.0_dp*u(2:n-1:2,3:n-2:2))$$
$$u(2:n-1:2,3:n-2:2)=u(2:n-1:2,3:n-2:2)-res(2:n-1:2,3:n-2:2)/(foh2+2.0_dp*u(2:n-1:2,3:n-2:2))$$
END SUBROUTINE relax2

See the discussion of red-black relaxation after sor on p. 1333.
SUBROUTINE slvsm2(u,rhs)

USE nrtype

IMPLICIT NONE

REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs

Solution of equation (19.6.44) on the coarsest grid, where \( h = \frac{1}{2} \). The right-hand side is input in \( \text{rhs}(1:3,1:3) \) and the solution is returned in \( u(1:3,1:3) \).

REAL(DP) :: disc, fact, h

\( u = 0.0 \)
\( h = 0.5_{\text{dp}} \)
\( \text{fact} = 2.0_{\text{dp}}/h^2 \)

\( \text{disc} = \sqrt{\text{fact}^2 + \text{rhs}(2,2)} \)
\( u(2,2) = -\text{rhs}(2,2)/(\text{fact} + \text{disc}) \)

END SUBROUTINE slvsm2

FUNCTION lop(u)

USE nrtype, USE nrutil, ONLY : assert_eq

IMPLICIT NONE

REAL(DP), DIMENSION(:,:), INTENT(IN) :: u
REAL(DP), DIMENSION(size(u,1),size(u,1)) :: lop

Given \( u \), returns \( L_h(\tilde{u}_h) \) for equation (19.6.44). \( u \) and \( lop \) are square arrays of the same odd dimension.

INTEGER(I4B) :: n
REAL(DP) :: h, h2i

\( n = \text{assert_eq(size}(u,1), \text{size}(u,2), 'lop') \)
\( h = 1.0_{\text{dp}}/(n-1) \)
\( h2i = 1.0_{\text{dp}}/(h*h) \)

\( \text{lop}(2:n-1,2:n-1) = h2i*(u(3:n,2:n-1)+u(1:n-2,2:n-1)+u(2:n-1,3:n)+ &
    u(2:n-1,1:n-2)-4.0_{\text{dp}}*u(2:n-1,2:n-1)) + u(2:n-1,2:n-1)**2 \) \quad \text{Interior points.} \\
\( \text{lop}(1:n,1) = 0.0 \) \quad \text{Boundary points.} \\
\( \text{lop}(1:n,n) = 0.0 \) \\
\( \text{lop}(1,1:n) = 0.0 \) \\
\( \text{lop}(n,1:n) = 0.0 \)

END FUNCTION lop
Chapter B20. Less-Numerical Algorithms

Volume 1’s Fortran 77 routine machar performed various clever contortions (due to Cody, Malcolm, and others) to discover the underlying properties of a machine’s floating-point representation. Fortran 90, by contrast, provides a built-in set of “numeric inquiry functions” that accomplish the same goal. The routine machar included here makes use of these and is included largely for compatibility with the previous version.

SUBROUTINE machar(ibeta,it,irnd,ngrd,machep,negep,iexp,minexp,&
maxexp,eps,epsneg,xmin,xmax)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: ibeta,iexp,irnd,it,machep,maxexp,minexp,negep,ngrd
REAL(SP), INTENT(OUT) :: eps,epsneg,xmax,xmin
REAL(SP), PARAMETER :: RX=1.0
Determines and returns machine-specific parameters affecting floating-point arithmetic. Returned values include ibeta, the number of base-ibeta digits in the floating-point mantissa; it, the number of base-ibeta digits in the floating-point mantissa; eps, the smallest positive number that, added to 1.0, is not equal to 1.0; epsneg, the smallest positive number that, subtracted from 1.0, is not equal to 1.0; xmin, the smallest representable positive number; and xmax, the largest representable positive number. See text for description of other returned parameters. Change all REAL(SP) declarations to REAL(DP) to find double-precision parameters.
REAL(SP) :: a,beta,betah,one,temp,tempa,two,zero
ibeta=radix(RX)
Most of the parameters are easily determined from intrinsic functions.
it=digits(RX)
machep=exponent(nearest(RX,RX)-RX)-1
negep=exponent(nearest(RX,-RX)-RX)-1
minexp=minexponent(RX)-1
maxexp=maxexponent(RX)
iexp=nint(log(real(maxexp-minexp+2,sp))/log(2.0_sp))
eps=real(ibeta,sp)**machep
epsneg=real(ibeta,sp)**negep
xmax=huge(RX)
xmin=tiny(RX)
one=RX
Determine irnd.
two=one+one
zero=one-one
beta=real(ibeta,sp)
a=beta**(-negep)
irnd=0
betah=beta/two
temp=a+betah
if (temp-a /= zero) irnd=1
temp=a+beta
temp=temp+betah
if ((irnd == 0) .and. (temp-tempa /= zero)) irnd=2
ngrd=0
Determine ngrd.
temp = one + eps
if ((irnd == 0) .and. (temp*one-one /= zero)) ngrd=1
temp=xmin/two
if (temp /= zero) irnd=irnd+3

Adjust irnd to reflect partial underflow.

END SUBROUTINE machar

⋆⋆⋆

FUNCTION igray(n,is)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n,is
INTEGER(I4B) :: igray

For zero or positive values of is, return the Gray code of n; if is is negative, return the inverse Gray code of n.

INTEGER(I4B) :: idiv,ish
if (is >= 0) then

This is the easy direction!
igray=ieor(n,n/2)
else

This is the more complicated direction: In hierarchical stages, starting with a one-bit right shift, cause each bit to be XORed with all more significant bits.
ish=1
igray=n
do
idiv=ishft(igray,ish)
igray=ieor(igray,idiv)
if (idiv <= 1 .or. ish == -16) RETURN
ish=ish+ish

Double the amount of shift on the next cycle.
end do
end if

END FUNCTION igray

⋆⋆⋆

FUNCTION icrc(crc,buf,jinit,jrev)
USE nrtype
IMPLICIT NONE
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: buf
INTEGER(I2B), INTENT(IN) :: crc,jinit
INTEGER(I4B), INTENT(IN) :: jrev

Computes a 16-bit Cyclic Redundancy Check for an array buf of bytes, using any of several conventions as determined by the settings of jinit and jrev (see accompanying table). The result is returned both as an integer icrc and as a 2-byte array crc. If jinit is negative, then crc is used on input to initialize the remainder register, in effect concatenating buf to the previous call.

INTEGER(I4B), SAVE :: init=0
INTEGER(I4B) :: j,cword,ich
INTEGER(I2B), DIMENSION(0:255), SAVE :: icrctb,rchr
INTEGER(I2B), DIMENSION(0:15) :: it = (/ 0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15 /)

if (init == 0) then

Do we need to initialize tables?
init=1
do j=0,255

The two tables are: CRCS of all characters, icrctb(j)=icrc1(ishft(j,8),char(0)) and bit-reverses of all characters, rchr(j)=ishft(it(iand(j,15_I2B)),4)+it(ishft(j,-4))
end do
end if

cword=crc
if (jinit >= 0) then

Initialize the remainder register.
cword=ior(jinit,ishft(jinit,8))
else if (jrev < 0) then  
   If not initializing, do we reverse the register?
   cword = ior(rchr(hibyte()), ishft(rchr(lobyte()), 8))
end if

do j=1, size(buf)  
   Main loop over the characters in the array.
   ich = ichar(buf(j))
   if (jrev < 0) ich = rchr(ich)
   cword = ieor(icrcb(ieor(ich, hibyte())), ishft(lobyte(), 8))
end do

icrc = merge(cword, &  
   Do we need to reverse the output?
   ior(rchr(hibyte()), ishft(rchr(lobyte()), 8)), jrev >= 0)
CONTAINS

FUNCTION hibyte()  
INTEGER(I2B) :: hibyte
   Extracts the high byte of the 2-byte integer cword.
   hibyte = ishft(cword, -8)
END FUNCTION hibyte

FUNCTION lobyte()  
INTEGER(I2B) :: lobyte
   Extracts the low byte of the 2-byte integer cword.
   lobyte = iand(cword, 255_I2B)
END FUNCTION lobyte

FUNCTION icrc1(crc, onech)  
INTEGER(I2B), INTENT(IN) :: crc
CHARACTER(1), INTENT(IN) :: onech
INTEGER(I2B) :: icrc1
   Given a remainder up to now, return the new CRC after one character is added. This routine is
   functionally equivalent to icrc(crc, -1, 1), but slower. It is used by icrc to initialize its table.
   icrc1 = ieor(crc, ishft(ich, 8))  
   Here is where the character is folded into the
   icrc1 = ieor(ccitt, ishft(icrc1, 1)), &  
   with the generator polynomial,
   ishft(icrc1, 1), iand(icrc1, bit16) /= 0)  
   are done.
end do
END FUNCTION icrc1
END FUNCTION icrc

The embedded functions hibyte and lobyte always act on the same
variable, cword. Thus they don’t need any explicit argument.

FUNCTION decchk(string, ch)
USE nrtype; USE nrutil, ONLY : ifirstloc
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(IN) :: string
CHARACTER(1), INTENT(OUT) :: ch
LOGICAL(LGT) :: decchk
   Decimal check digit computation or verification. Returns as ch a check digit for appending
to string. In this mode, ignore the returned logical value. If string already ends with
a check digit, returns the function value .true. If the check digit is valid, otherwise
.false. In this mode, ignore the returned value of ch. Note that string and ch contain
ASCII characters corresponding to the digits 0-9, not byte values in that range. Other ASCII
characters are allowed in string, and are ignored in calculating the check digit.
   INTEGER(I4B) :: i, j, k, m
   INTEGER(I4B) :: ip(0:9,0:7) = reshape((/  
   Group multiplication and permuta-
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 5, 7, 6, 2, 8, 3, 0, 9, 4,  
   5, 8, 0, 3, 7, 9, 6, 1, 4, 2, 8, 6, 5, 7, 3, 9, 0, 1, 2, 7, 9, 3, 8, 0, 6, 4, 1, 5, 7, 0, 4, 6, 9, 1, 3, 2, 5, 8 /), &  
end FUNCTION decchk
INTEGER(I4B) :: ij(0:9,0:9) = reshape((/ &
0,1,2,3,4,5,6,7,8,9,1,2,3,4,0,9,5,6,7,8,2,3,4,0,1,8,9,5,6,&
7,3,4,0,1,2,7,8,9,5,6,4,0,1,2,3,6,7,8,9,5,6,7,8,9,0,1,2,3,&
4,6,7,8,9,6,4,0,1,2,3,7,8,9,5,6,3,4,0,1,2,8,9,5,6,7,2,3,4,0,&
1,9,5,6,7,8,1,2,3,4,0 /),(/ 10,10 /))

k=0
m=0
do j=1,size(string)
Look at successive characters.
i=ichar(string(j))
if (i >= 48 .and. i <= 57) then
Ignore everything except digits.
k=ij(k,ip(mod(i+2,10),mod(m,8)))
m=m+1
end if
end do

decchk=logical(k == 0,kind=lgt)
i=mod(m,8)
i=ifirstloc(ij(k,ip(0:9,i)) == 0)-1
Find which appended digit will check properly.
ch=char(i+48)
Convert to ASCII.
END FUNCTION decchk

Note the use of the utility function ifirstloc to find the first (in this case, the only) correct check digit.

* * *

The Huffman and arithmetic coding routines exemplify the use of modules to encapsulate user-defined data types. In these algorithms, “the code” is a fairly complicated construct containing scalar and array data. We define types huffcode and arithcode, then can pass “the code” from the routine that constructs it to the routine that uses it as a single variable.

MODULE huf_info
USE nrtype
IMPLICIT NONE
TYPE huffcode
INTEGER(I4B) :: nch,nodemax
INTEGER(I4B), DIMENSION(:), POINTER :: icode,left,iright,ncode
END TYPE huffcode
CONTAINS
SUBROUTINE huff_allocate(hcode,mc)
USE nrtype
IMPLICIT NONE
TYPE(huffcode) :: hcode
INTEGER(I4B) :: mc
mq=2*mc-1
allocate(hcode%icode(mq),hcode%ncode(mq),hcode%left(mq),hcode%iright(mq))
hcode%ncode(:)=0
mc=mc+1
END SUBROUTINE huff_allocate
SUBROUTINE huff_deallocate(hcode)
USE nrtype
IMPLICIT NONE
TYPE(huffcode) :: hcode
deallocate(hcode%iright,hcode%left,hcode%ncode,hcode%icode)
nullify(hcode%icode)
nulify(hcode%ncode)
nulify(hcode%left)
nulify(hcode%iright)
SUBROUTINE hufmak(nfreq,ilong,nlong,hcode)
USE nrtype; USE nrutil, ONLY : array_copy, arth, imaxloc, nrerror
USE huf_info
IMPLICIT NONE

INTEGER(I4B), INTENT(OUT) :: ilong,nlong
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nfreq

TYPE(huffcode) :: hcode
Given the frequency of occurrence table nfreq of size(nfreq) characters, return the
Huffman code hcode. Returned values ilong and nlong are the character number that
produced the longest code symbol, and the length of that symbol.

INTEGER(I4B) :: ibit,j,k,n,node,nused,nerr
INTEGER(I4B), DIMENSION(2*size(nfreq)-1) :: indx,iup,nprob

hcode%nch=size(nfreq)
Initialization.
call huff_allocate(hcode,size(nfreq))
nused=0
nprob(1:hcode%nch)=nfreq(1:hcode%nch)
call array_copy(pack(arth(1,1,hcode%nch), nfreq(1:hcode%nch) /= 0 ),&
indx,nused,nerr)
do j=nused,1,-1
Sort nprob into a heap structure in indx.
call hufapp(j)
end do
k=hcode%nch
Combine heap nodes, remaking the heap at each stage.
do if (nused <= 1) exit
node=indx(1)
indx(1)=indx(nused)
nused=nused-1
call hufapp(1)
k=k+1
nprob(k)=nprob(indx(1))+nprob(node)
hcode%left(k)=node
Store left and right children of a node.
hcode%right(k)=indx(1)
iup(indx(1))=-k
iup(node)=k
indx(1)=k
call hufapp(1)
end do
hcode%nodemax=k
iup(hcode%nodemax)=0
Make the Huffman code from the tree.
do j=1,hcode%nch
if (nprob(j) /= 0) then
n=0
ibit=0
node=iup(j)
do if (node == 0) exit
if (node < 0) then
n=ibset(n,ibit)
node=-node
end if
node=iup(node)
ibit=ibit+1
end do
hcode%icode(j)=n
hcode%ncode(j)=ibit
end if
end do
ilong=imaxloc(hcode%ncode(1:hcode%nch))
nlong=hcode%ncode(ilong)
if (nlong > bit_size(1_i4b)) call & Check along not larger than word length.
rerror('hufmak: Number of possible bits for code exceeded')

CONTAINS

SUBROUTINE hufapp(l)
IMPLICIT NONE
INTENSE(I4B), INTENT(IN) :: l

Used by hufmak to maintain a heap structure in the array indx(1:l).

n=nused
i=1
k=indx(1)
do
   if (i > n/2) exit
   j=i+i
   if (j < n .and. nprob(indx(j)) > nprob(indx(j+1))) &
      j=j+1
   if (nprob(k) <= nprob(indx(j))) exit
   indx(i)=indx(j)
i=j
end do
indx(i)=k
END SUBROUTINE hufapp

SUBROUTINE hufmak
IMPLICIT NONE

INTEGER(I4B), INTENT(IN) :: l

Used by hufmak to maintain a heap structure in the array indx(1:l).

n=nused
i=1
k=indx(1)
do
   if (i > n/2) exit
   j=i+i
   if (j < n .and. nprob(indx(j)) > nprob(indx(j+1))) &
      j=j+1
   if (nprob(k) <= nprob(indx(j))) exit
   indx(i)=indx(j)
i=j
end do
indx(i)=k
END SUBROUTINE hufmak

SUBROUTINE hufapp(l)
IMPLICIT NONE
INTENSE(I4B), INTENT(IN) :: l

Used by hufmak to maintain a heap structure in the array indx(1:l).

n=nused
i=1
k=indx(1)
do
   if (i > n/2) exit
   j=i+i
   if (j < n .and. nprob(indx(j)) > nprob(indx(j+1))) &
      j=j+1
   if (nprob(k) <= nprob(indx(j))) exit
   indx(i)=indx(j)
i=j
end do
indx(i)=k
END SUBROUTINE hufapp

SUBROUTINE hufenc(ich,codep,nb,hcode)
IMPLICIT NONE

INTEGER(I4B), INTENT(IN) :: ich
INTEGER(I4B), INTENT(INOUT) :: nb
CHARACTER(1), DIMENSION(:), POINTER :: codep
TYPE(huffcode) :: hcode

Huffman encode the single character ich (in the range 0..nch-1) using the code in hcode,
write the result to the character array pointed to by codep starting at bit nb (whose smallest
valid value is zero), and increment nb appropriately. This routine is called repeatedly to
encode consecutive characters in a message, but must be preceded by a single initializing
call to hufmak.

INTEGER(I4B) :: k,l,n,nc,ntmp
k=ich+1
if (k > hcode%nch .or. k < 1) call & Convert character range 0..nch-1 to array index range 1..nch.
rerror('hufenc: ich out of range')
do n=hcode%code(k),1,-1 Loop over the bits in the stored Huffman
   nc=nb/8+1 code for ich.
   if (nc > size(codep)) codep=>reallocate(codep,2*size(codep))
   l=mod(nb,8)
   if (l == 0) codep(nc)=char(0)
   if (btest(hcode%code(k),n-1)) then Set appropriate bits in codep.
      ntmp=ibset(ichar(codep(nc)),l)
      codep(nc)=char(ntmp)
   end if
   nb=nb+1
end do
END SUBROUTINE hufenc
SUBROUTINE hufdec(ich, code, nb, hcode)
USE nrtype
USE huf_info
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: ich
INTEGER(I4B), INTENT(INOUT) :: nb
CHARACTER(1), DIMENSION(:), INTENT(IN) :: code
TYPE(huffcode) :: hcode

Starting at bit number nb in the character array code, use the Huffman code in hcode to decode a single character (returned as ich in the range 0..nch-1) and increment nb appropriately. Repeated calls, starting with nb = 0, will return successive characters in a compressed message. The returned value ich=nch indicates end-of-message. This routine must be preceded by a single initializing call to hufmak.

INTEGER(I4B) :: l,nc,node
node=hcode%nodemax

nc=nb/8+1
if (nc > size(code)) then
    nb=nb+1
    if (btest(ichar(code(nc)),l)) then
        node=hcode%right(node)
    else
        node=hcode%left(node)
    end if
    if (node <= hcode%nch) then
        ich=node-1
        RETURN
    end if
end if

do
    if (nb > size(code)) return with ich=nch indicating end of message.
    l=mod(nb,8)
    nc=nb/l
    if (btest(ichar(code(nc)),l)) then
        node=hcode%right(node)
    else
        node=hcode%left(node)
    end if
    if (node <= hcode%nch) then
        ich=node-1
        RETURN
    end if
end do

END SUBROUTINE hufdec

* * *

MODULE arcode_info
USE nrtype
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: NWK=20

NWK is the number of working digits (see text).

TYPE arithcode
    INTEGER(I4B), DIMENSION(:), POINTER :: ilob,iupb,ncumfq!
    INTEGER(I4B) :: jdif,nc,minint,nch,ncum,nrad
END TYPE arithcode

CONTAINS

SUBROUTINE arcode_allocate(acode,mc)
USE nrtype
IMPLICIT NONE
TYPE(arithcode) :: acode
INTEGER(I4B) :: mc
allocate(acode%ilob(NWK),acode%iupb(NWK),acode%ncumfq(mc+2))
END SUBROUTINE arcode_allocate

SUBROUTINE arcode_deallocate(acode)
USE nrtype
IMPLICIT NONE
TYPE(arithcode) :: acode
nullify(acode%ilob)
nullify(acode%iupb)
nullify(acode%ncumfq)
nullify(acode%ncumfq)
END SUBROUTINE arcode_deallocate

END MODULE arcode_info

SUBROUTINE arcmak(nfreq,nradd,acode)
USE nrtype; USE nrutil, ONLY : cumsum,nrerror
USE arcode_info
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: nradd
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nfreq
TYPE(arithcode) :: acode
INTEGER(I4B), PARAMETER :: MAXINT=huge(nradd)

Given a table nfreq of the frequency of occurrence of size(nfreq) symbols, and given a desired output radix nradd, initialize the cumulative frequency table and other variables for arithmetic compression. Store the code in acode.

MAXINT is a large positive integer that does not overflow.

if (nradd > 256) call nrerror('output radix may not exceed 256 in arcmak')
acode%minint=MAXINT/nradd
acode%nch=size(nfreq)
acode%nrad=nradd
call arcode_allocate(acode,acode%nch)
acode%ncumfq(1)=0
acode%ncumfq(2:acode%nch+1)=cumsum(max(nfreq(1:acode%nch),1))
acode%ncumfq(acode%nch+2)=acode%ncumfq(acode%nch+1)+1
acode%ncum=acode%ncumfq(acode%nch+2)
END SUBROUTINE arcmak

SUBROUTINE arcode(ich,codep,lcd,isign,acode)
USE nrtype; USE nrutil, ONLY : nrerror,reallocate
USE arcode_info
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: ich,lcd
INTEGER(I4B), INTENT(IN) :: isign
CHARACTER(1), DIMENSION(:), POINTER :: codep
TYPE(arithcode) :: acode

Compress (isign = 1) or decompress (isign = -1) the single character ich into or out of the character array pointed to by codep, starting with byte codep(lcd) and (if necessary) incrementing lcd so that, on return, lcd points to the first unused byte in codep. Note that this routine saves the result of previous calls until a new byte of code is produced, and only then increments lcd. An initializing call with isign=0 is required for each different array codep. The routine arcmak must have previously been called to initialize the code acode. A call with ich=arcode%nch (as set in arcmak) has the reserved meaning “end of message.”

INTEGER(I4B) :: ihi,j,ja,jh,jl,m

if (isign == 0) then
Initialize enough digits of the upper and lower bounds.
acode%jdif=acode%nrad-1
acode%ilob(:)=0
acode%iupb(:)=acode%nrad-1
do j=NWK,1,-1
acode%nc=j
if (acode%jdif > acode%minint) RETURN Initialization complete.
acode%jdif=(acode%jdif+1)*acode%nrad-1
end do
call nrerror('NWK too small in arcode')
else
if (isign > 0) then
If encoding, check for valid input character.
if (ich > acode%nch .or. ich < 0) call nrerror('bad ich in arcode')
else
If decoding, locate the character ich by bi-
ja=ichar(codep(lcd))-acode%ilob(acode%nc)
do j=acode%nc+1,NWK

ja=ja*acode%nrad+(ichar(codep(j+lcd-acode%nc))-acode%ilob(j))
end do
ich=0
ihi=acode%nch+1
do
    if (ihi-ich <= 1) exit
    m=(ich+ihi)/2
    if (ja >= jtry(acode%jdif,acode%ncumfq(m+1),acode%ncum)) then
        ich=m
    else
        ihi=m
    end if
end do
if (ich == acode%nch) RETURN  
end if

Following code is common for encoding and decoding. Convert character ich to a new subrange [ilob,iupb).
jh=jtry(acode%jdif,acode%ncumfq(ich+2),acode%ncum)
jl=jtry(acode%jdif,acode%ncumfq(ich+1),acode%ncum)
acode%jdif=jh-jl
call arcsum(acode%ilob,acode%iupb,jh,NWK,acode%nrad,acode%nc)
How many leading digits to output (if encoding) or skip over?
call arcsum(acode%ilob,acode%ilob,jl,NWK,acode%nrad,acode%nc)
do j=acode%nc,NWK
    if (ich /= acode%nch .and. acode%iupb(j) /= acode%ilob(j)) exit
    if (acode%nc > size(codep)) codep=>reallocate(codep,2*size(codep))
    if (isign > 0) codep(lcd)=char(acode%ilob(j))
    lcd=lcd+1
end do
if (j > NWK) RETURN  
Ran out of message. Did someone forget to
acode%nc=j
j=0
How many digits to shift?
do
    if (acode%jdif >= acode%minint) exit
    j=j+1
    acode%jdif=acode%jdif*acode%nrad
end do
if (acode%nc-j < 1) call nrerror('NWK too small in arcde')
if (j /= 0) then
    acode%iupb(acode%nc-j:acode%nc)=0
    acode%ilob(acode%nc-j:acode%nc)=0
end if
acode%nc=acode%nc-j
acode%iupb(NWK-j+1:NWK)=0
acode%ilob(NWK-j+1:NWK)=0
end if
Normal return.
CONTAINS
FUNCTION jtry(m,n,k)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: m,n,k
INTEGER(I4B) :: jtry
Calculate (m*n)/k without overflow. Program efficiency can be improved by substituting an
assembly language routine that does integer multiply to a double register.
jtry=int((real(m,dp)*real(n,dp))/real(k,dp))
END FUNCTION jtry
SUBROUTINE arcsum(iin,iout,ja,nwk,nrad,nc)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: iin
INTEGER(I4B), DIMENSION(:,), INTENT(INOUT) :: iout
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: nwk,nrad,nc
INTEGER(I4B), INTENT(INOUT) :: ja
Add the integer ja to the radix nrad multiple-precision integer iin(nc..nwk). Return the result in iout(nc..nwk).

```fortran
INTEGER(I4B) :: j, jtmp, karry
karry=0
DO j=nwk,nc+1,-1
  jtmp=ja
  ja=ja/nrad
  iout(j)=iin(j)+(jtmp-ja*nrad)+karry
  IF (iout(j) >= nrad) THEN
    iout(j)=iout(j)-nrad
    karry=1
  ELSE
    karry=0
  END IF
END DO
iout(nc)=iin(nc)+ja+karry
END SUBROUTINE arcsum
END SUBROUTINE arcode
```

---

### MODULE mpops

USE nrtype
INTEGER(I4B), PARAMETER :: NPAR_ICARRY=64
CONTAINS

**SUBROUTINE icarry(karry,isum,nbits)**

**IMPLICIT NONE**

INTEGER(I4B), INTENT(OUT) :: karry

**Perform deferred carry operation on an array isum of multiple-precision digits. Nonzero bits of higher order than nbits (typically 8) are carried to the next-lower (leftward) component of isum. The final (most leftward) carry value is returned as karry.**

INTEGER(I2B), DIMENSION(:), INTENT(INOUT) :: isum
INTEGER(I4B), INTENT(IN) :: nbits
INTEGER(I4B) :: n,j
INTEGER(I2B), DIMENSION(size(isum)) :: ihi
INTEGER(I2B) :: mb,ihh
n=size(isum)
mb=ishft(1,nbits)-1
karry=0
IF (n < NPAR_ICARRY ) THEN
  DO j=n,2,-1
    ihh=ishft(isum(j),-nbits)
    IF (ihh /= 0) THEN
      isum(j)=iand(isum(j),mb)
      isum(j-1)=isum(j-1)+ihh
    END IF
  END DO
  ihh=ishft(isum(1),-nbits)
  isum(1)=iand(isum(1),mb)
  karry=karry+ihh
ELSE
  DO
    ihi=ishft(isum,-nbits)
    IF (all(ihi == 0)) EXIT
    WHERE (ihi /= 0) isum=iand(isum,mb)
    WHERE (ihi(2:n) /= 0) isum(1:n-1)=isum(1:n-1)+ihi(2:n)
  karry=karry+ihi(1)
  END DO
END IF
END SUBROUTINE icarry
SUBROUTINE mpadd(w,u,v,n)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:,), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: u,v
INTEGER(I4B), INTENT(IN) :: n

Adds the unsigned radix 256 integers u(1:n) and v(1:n) yielding the unsigned integer w(1:n+1).

INTEGER(I2B), DIMENSION(n) :: isum
INTEGER(I4B) :: karry
isum=ichar(u(1:n))+ichar(v(1:n))
call icarry(karry,isum,8_I4B)
w(2:n+1)=char(isum)
w(1)=char(karry)
END SUBROUTINE mpadd

SUBROUTINE mpsub(is,w,u,v,n)
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: is
CHARACTER(1), DIMENSION(:,), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: u,v
INTEGER(I4B), INTENT(IN) :: n

Subtracts the unsigned radix 256 integer v(1:n) from u(1:n) yielding the unsigned integer w(1:n). If the result is negative (wraps around), is is returned as −1; otherwise it is returned as 0.

INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=255+ichar(u(1:n))-ichar(v(1:n))
isum(n)=isum(n)+1
call icarry(karry,isum,8_I4B)
w(1:n)=char(isum)
is=karry-1
END SUBROUTINE mpsub

SUBROUTINE mpsad(w,u,n,iv)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:,), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: u
INTEGER(I4B), INTENT(IN) :: n,iv

Short addition: The integer iv (in the range 0 ≤ iv ≤ 255) is added to the unsigned radix 256 integer u(1:n), yielding w(1:n+1).

INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=ichar(u(1:n))
isum(n)=isum(n)+iv
call icarry(karry,isum,8_I4B)
w(2:n+1)=char(isum)
w(1)=char(karry)
END SUBROUTINE mpsad

SUBROUTINE mpsmu(w,u,n,iv)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:,), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: u
INTEGER(I4B), INTENT(IN) :: n,iv

Short multiplication: The unsigned radix 256 integer u(1:n) is multiplied by the integer iv (in the range 0 ≤ iv ≤ 255), yielding w(1:n+1).

INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=ichar(u(1:n))*iv
call icarry(karry,isum,8_I4B)
w(2:n+1)=char(isum)
w(1)=char(karry)
END SUBROUTINE mpsmu

SUBROUTINE mpneg(u,n)
IMPLICIT NONE

SUBROUTINE mpneg(u,n)
CHARACTER(1), DIMENSION(:,), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: n
   Ones-complement negate the unsigned radix 256 integer u(1:n).
INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=255-ichar(u(1:n))
isum(n)=isum(n)+1
call icarry(karry,isum,8_I4B)
u(1:n)=char(isum)
END SUBROUTINE mpneg

SUBROUTINE mplsh(u,n)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:,), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: n
   Left shift u(2..n+1) onto u(1:n).
u(1:n)=u(2:n+1)
END SUBROUTINE mplsh

SUBROUTINE mpmov(u,v,n)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: v
CHARACTER(1), DIMENSION(:,), INTENT(OUT) :: u
INTEGER(I4B), INTENT(IN) :: n
   Move v(1:n) onto u(1:n).
u(1:n)=v(1:n)
END SUBROUTINE mpmov

SUBROUTINE mpsdv(w,u,n,iv,ir)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: u
CHARACTER(1), DIMENSION(:,), INTENT(OUT) :: w
INTEGER(I4B), INTENT(IN) :: n,iv
INTEGER(I4B), INTENT(OUT) :: ir
   Short division: The unsigned radix 256 integer u(1:n) is divided by
   the integer iv (in the range 0 ≤ iv ≤ 255), yielding a quotient
   w(1:n) and a remainder ir (with 0 ≤ ir ≤ 255).
Note: Your Numerical Recipes authors don't know how to parallelize this routine
   in Fortran 90!
INTEGER(I4B) :: i,j
ir=0
do j=1,n
   i=256*ir+ichar(u(j))
w(j)=char(i/iv)
   ir=mod(i,iv)
end do
END SUBROUTINE mpsdv

SUBROUTINE mpsmul(w,u,v,n,m)
USE nrtype; USE nuutil, ONLY : nrerror
USE nr, ONLY : realft
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n,m
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: u,v
! The logical dimensions are: CHARACTER(1) :: w(n+m),u(n),v(m)
REAL(DP), PARAMETER :: RX=256.0
   Uses fast Fourier transform to multiply the unsigned radix 256 integers u(1:n) and v(1:m),
   yielding a product w(1:n+m).
INTEGER(I4B) :: j,nn
REAL(DP) :: cy,t
REAL(DP), DIMENSION(:,), ALLOCATABLE :: a,b, tb
nn=max(m,n)

   Find the smallest usable power of two for the transform.
do
  if (nn >= mn) exit
  nn=nn+nn
end do
nn=nn+nn
allocate(a(nn),b(nn),tb((nn-1)/2))
a(1:n)=ichar(u(1:n))  Move U to a double-precision floating array.
a(n+1:nn)=0.0
b(1:m)=ichar(v(1:m))  Move V to a double-precision floating array.
b(m+1:nn)=0.0
call realft(a(1:nn),1)  Perform the convolution: First, the two Fourier transforms.
call realft(b(1:nn),1)  Then multiply the complex results (real and imaginary parts).
b(1)=b(1)*a(1)
b(2)=b(2)*a(2)
tb=b(3:nn:2)
b(3:nn:2)=tb*a(3:nn:2)-b(4:nn:2)*a(4:nn:2)
b(4:nn:2)=tb*a(4:nn:2)+b(4:nn:2)*a(3:nn:2)
b(:)=b(:)/(nn/2)  Then do the inverse Fourier transform.
cy=0.0  Make a final pass to do all the carries.
do  The 0.5 allows for roundoff error.
j=nn,1,-1
  t=b(j)+cy+0.5_dp
  b(j)=mod(t,RX)
  cy=int(t/RX)
end do
if (cy >= RX) call nrerror('mpmul: sanity check failed in fftmul')
  w(1)=char(int(cy))  Copy answer to output.
w(2:(n+m))=char(int(b(1:(n+m-1))))
deallocate(a,b,tb)
END SUBROUTINE mpmul

SUBROUTINE mpinv(u,v,n,m)
  USE nrtype; USE nrutil, ONLY : poly
  USE nr, ONLY : mpmul
  USE mpops, ONLY : mpmov,mpneg
  IMPLICIT NONE
  CHARACTER(1), DIMENSION(:), INTENT(OUT) :: u
  CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
  INTEGER(I4B), INTENT(IN) :: n,m
  INTEGER(I4B), PARAMETER :: MF=4
  REAL(SP), PARAMETER :: BI=1.0_sp/256.0_sp

  Character string v(1:m) is interpreted as a radix 256 number with the radix point after (nonzero) v(1); u(1:n) is set to the most significant digits of its reciprocal, with the radix point after u(1).

  INTEGER(I4B) :: i,j,mm
  REAL(SP) :: fu
  CHARACTER(1), DIMENSION(:), ALLOCATABLE :: rr,s
  allocate(rr(max(n,m)+n+1),s(n))
  mm=min(MF,m)
  fu=1.0_sp/poly(BI,real(ichar(v(:)),sp))  Use ordinary floating arithmetic to get an initial approximation.
do  Iterate Newton's rule to convergence.
j=1,n
  i=int(fu)
  u(i)=char(i)
  fu=256.0_sp*(fu-i)
end do
  Construct 2 − UV in S.
call mpmul(rr,u,v,n,m)
call mpmov(s,rr(2:),n)
call mpmneg(s,n)
s(1)=char(ichar(s(1))-254)
call mpmul(rr,s,u,v,n)
call mpmul(u,rr(2:),n)
if (all(ichar(s(2:n-1)) == 0)) exit
end do
deallocate(rr,s)
END SUBROUTINE mpinv

SUBROUTINE mpdiv(q,r,u,v,n,m)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : mpinv,mpmul
USE mpops, ONLY : mpsad,mpmov,mpsub
IMPLICIT NONE
CHARACTER(1), DIMENSION(:,), INTENT(OUT) :: q,r
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: u,v
! The logical dimensions are: CHARACTER(1) :: q(n-m+1),r(m),u(n),v(m)
INTEGER(1B), INTENT(IN) :: n,m

Divides unsigned radix 256 integers u(1:n) by v(1:m) (with m ≤ n required), yielding a
quotient q(1:n-m+1) and a remainder r(1:m).

INTEGER(1B), PARAMETER :: MACC=6
INTEGER(1B) :: is
CHARACTER(1), DIMENSION(:), ALLOCATABLE, TARGET :: rr,s
CHARACTER(1), DIMENSION(:), POINTER :: rr2,s3
allocate(rr(2*(n+MACC)),s(2*(n+MACC)))
rr2=>rr(2:)
s3=>s(3:)
call mpinv(s,v,n+MACC,m)
Set S = 1/V.
call mpmul(rr,s,u,n+MACC,n)
Set Q = SU.
call mpsad(s,rr,n+MACC-1,1)
call mpmov(q,s3,n-m+1)
call mpmul(rr,q,v,n-m+1,m)
Multiply and subtract to get the remainder.
call mpsub(is,rr2,u,rr2,n)
if (is /= 0) call nrerror('MACC too small in mpdiv')
call mpmov(r,rr(n-m+2:),m)
deallocate(rr,s)
END SUBROUTINE mpdiv

SUBROUTINE mpsqrt(w,u,v,n,m)
USE nrtype; USE nrutil, ONLY : poly
USE nr, ONLY : mpmul
USE mpops, ONLY : mplsh,mpmov,mpneg,mpsdv
IMPLICIT NONE
CHARACTER(1), DIMENSION(:,), INTENT(OUT) :: w,u
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: v
INTEGER(1B), INTENT(IN) :: n,m
INTEGER(1B), PARAMETER :: MF=3
REAL(SP), PARAMETER :: BI=1.0_sp/256.0_sp
Character string v(1:m) is interpreted as a radix 256 number with the radix point after
v(1); w(1:n) is set to its square root (radix point before w(1)), and u(1:n) is set to the
reciprocal thereof (radix point before u(1)). w and u need not be distinct, in which case
they are set to the square root.

INTEGER(1B), :: i,ir,j,mm
REAL(SP) :: fu
CHARACTER(1), DIMENSION(:,), ALLOCATABLE :: r,s
allocate(r(2*n),s(2*n))
mm=min(m,MF)
fu=1.0_sp/sqrt(poly(BI,real(ichar(v(:,)),sp)))
Use ordinary floating arithmetic
to get an initial approximation.
do j=1,n
i=int(fu)
u(j)=char(i)
fu=256.0_sp*(fu-i)
end do
do
call mpmul(r,u,u,n,n)
Iterate Newton’s rule to convergence.
Construct S = (3 - VU^2)/2.
call mplsh(r,n)
call mpmul(s,r,v,n,min(m,n))
call mplsh(s,n)
call mpneg(s,n)
s(1)=char(ichar(s(1))-253)
call mpsdv(s,s,n,2,ir)
if (any(ichar(s(2:n-1)) /= 0)) then
  If fractional part of $S$ is not zero, it has not converged to 1.
call mpmul(r,s,u,n,n) Replace $U$ by $SU$.
call mpmov(u,r(2:),n)
cycle
end if

call mpmul(r,u,v,n,min(m,n)) Get square root from reciprocal and return.
call mpmov(w,r(2:),n)
deallocate(r,s)
RETURN
end do
END SUBROUTINE mpsqrt

SUBROUTINE mp2dfr(a,s,n,m)
USE nrtype
USE mpops, ONLY : mplsh,mpsmu
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), INTENT(OUT) :: m
CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: a
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: s
INTEGER(I4B), PARAMETER :: IAZ=48
Converts a radix 256 fraction $a(1:n)$ (radix point before $a(1)$) to a decimal fraction represented as an ascii string $s(1:m)$, where $m$ is a returned value. The input array $a(1:n)$ is destroyed. NOTE: For simplicity, this routine implements a slow ($\propto N^2$) algorithm. Fast ($\propto N \ln N$), more complicated, radix conversion algorithms do exist.

INTEGER(I4B) :: j
m=int(2.408_sp*n)
do j=1,m
  call mpsmu(a,a,n,10)
s(j)=char(ichar(a(1))+IAZ)
call mplsh(a,n)
end do
END SUBROUTINE mp2dfr

SUBROUTINE mppi(n)
USE nrtype
USE nr, ONLY : mp2dfr,mpinv,mpmul,mpsqrt
USE mpops, ONLY : mpadd,mplsh,mpmov,mpsdv
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), PARAMETER :: IAOFF=48
Demonstrate multiple precision routines by calculating and printing the first $n$ bytes of $\pi$.

INTEGER(I4B) :: ir,j,m
CHARACTER(1), DIMENSION(n) :: sx,sxi
CHARACTER(1), DIMENSION(2*n) :: t,y
CHARACTER(1), DIMENSION(3*n) :: s
CHARACTER(1), DIMENSION(n+1) :: x,bigpi

$t(1)$=char(2) Set $T = 2$.
t(2:n)=char(0)
call mpsqrt(x,t,n,n) Set $X_0 = \sqrt{2}$.
call mpadd(bigpi,t,x,n) Set $\pi_0 = 2 + \sqrt{2}$.
call mplsh(bigpi,n)
call mpsqrt(sx,sxi,n,n) Set $Y_0 = 2^{1/4}$.
call mpmov(y,sx,n)
do
  call mpadd(x,sx,sxi,n)  
  call mpsqrt(x,y,sx,n)  
  call mpadd(t,y,sx,sxi,n)  
  x(1)=char(ichar(x(1))+1)  
  y(1)=char(ichar(y(1))+1)  
  call mpinv(s,y,n,n)  
  call mpmul(y,t(3:),s,n,n)  
  call mplsh(y,n)
  x(1)=char(ichar(x(1))+1)  
  y(1)=char(ichar(y(1))+1)  
  call mpmov(bigpi,t(2:),n,n)
  s(1)=char(ichar(bigpi(1))+IAOFF)  
  s(2)='.  
  s(3:)=m2dfr(bigpi(2:),s(3:),n-1,m)
cycle  
if (abs(ichar(t(n+1))-m) > 1 .or. any(ichar(t(3:n)) /= m)) then  
  call mpmul(s,bigpi,t(2:),n,n)  
  call mpmov(bigpi,s(2:),n)
end if
write (*,*) 'pi='
write (*,'(1x,64a1)') (s(j),j=1,m+1)
return  
end do  
END SUBROUTINE mppi
References

The references collected here are those of general usefulness, cited in this volume. For references to the material in Volume 1, see the References section of that volume.

A first group of references relates to the Fortran 90 language itself:


A second group of references relates to, or includes material on, parallel programming and algorithms:


C1. Listing of Utility Modules (nrtype and nrutil)

C1.1 Numerical Recipes Types (nrtype)

The file supplied as nrtype.f90 contains a single module named nrtype, which in turn contains definitions for a number of named constants (that is, PARAMETERS), and a couple of elementary derived data types used by the sparse matrix routines in this book. Of the named constants, by far the most important are those that define the KIND types of virtually all the variables used in this book: I4B, I2B, and I1B for integer variables, SP and DP for real variables (and SPC and DPC for the corresponding complex cases), and LGT for the default logical type.

MODULE nrtype
Symbolic names for kind types of 4-, 2-, and 1-byte integers:
INTEGER, PARAMETER :: I4B = SELECTED_INT_KIND(9)
INTEGER, PARAMETER :: I2B = SELECTED_INT_KIND(4)
INTEGER, PARAMETER :: I1B = SELECTED_INT_KIND(2)
Symbolic names for kind types of single- and double-precision reals:
INTEGER, PARAMETER :: SP = KIND(1.0)
INTEGER, PARAMETER :: DP = KIND(1.000)
Symbolic names for kind types of single- and double-precision complex:
INTEGER, PARAMETER :: SPC = KIND((1.0, 1.0))
INTEGER, PARAMETER :: DPC = KIND((1.000, 1.000))
Symbolic name for kind type of default logical:
INTEGER, PARAMETER :: LGT = KIND(.true.)

Frequently used mathematical constants (with precision to spare):
REAL(SP), PARAMETER :: PI=3.141592653589793238462643383279502884197_sp
REAL(SP), PARAMETER :: PIO2=1.57079632679489661923132169163975144209858_sp
REAL(SP), PARAMETER :: TWOPI=6.2831853071795864766565905768394_sp
REAL(SP), PARAMETER :: SQRT2=1.41421356237309504880168872420969807856397_sp
REAL(SP), PARAMETER :: EULER=0.57721566490153286065120900824024310422_sp
REAL(DP), PARAMETER :: PI_D=3.141592653589793238462643383279502884197_dp
REAL(DP), PARAMETER :: PIO2_D=1.57079632679489661923132169163975144209858_dp
REAL(DP), PARAMETER :: TWOPI_D=6.2831853071795864766565905768394_dp
REAL(DP), PARAMETER :: SQRT2_D=1.41421356237309504880168872420969807856397_dp
REAL(DP), PARAMETER :: EULER_D=0.57721566490153286065120900824024310422_dp

Derived data types for sparse matrices, single and double precision (see use in Chapter B2):

TYPE sprs2_sp
  INTEGER(I4B) :: n, len
  REAL(SP), DIMENSION(:,), POINTER :: val
  INTEGER(I4B), DIMENSION(:,), POINTER :: irow
  INTEGER(I4B), DIMENSION(:,), POINTER :: jcol
END TYPE sprs2_sp

TYPE sprs2_dp
  INTEGER(I4B) :: n, len
  REAL(DP), DIMENSION(:,), POINTER :: val
  INTEGER(I4B), DIMENSION(:,), POINTER :: irow
  INTEGER(I4B), DIMENSION(:,), POINTER :: jcol
END TYPE sprs2_dp
About Converting to Higher Precision

You might hope that changing all the Numerical Recipes routines from single precision to double precision would be as simple as redefining the values of SP and DP in nrtype. Well ... not quite.

Converting algorithms to a higher precision is not a purely mechanical task because of the distinction between “roundoff error” and “truncation error.” (Please see Volume 1, §1.2, if you are not familiar with these concepts.) While increasing the precision implied by the kind values SP and DP will indeed reduce a routine’s roundoff error, it will not reduce any truncation error that may be intrinsic to the algorithm. Sometimes, a routine contains “accuracy parameters” that can be adjusted to reduce the truncation error to the new, desired level. In other cases, however, the truncation error cannot be so easily reduced; then, a whole new algorithm is needed. Clearly such new algorithms are beyond the scope of a simple mechanical “conversion.”

If, despite these cautionary words, you want to proceed with converting some routines to a higher precision, here are some hints:

If your machine has a kind type that is distinct from, and has equal or greater precision than, the kind type that we use for DP, then, in nrtype, you can simply redefine DP to this new highest precision and redefine SP to what was previously DP. For example, DEC machines usually have a “quadruple precision” real type available, which can be used in this way. You should not need to make any further edits of nrtype or nrutil.

If, on the other hand, the kind type that we already use for DP is the highest precision available, then you must leave DP defined as it is, and redefine SP in nrtype to be this same kind type. Now, however, you will also have to edit nrutil, because some overloaded routines that were previously distinguishable (by the different kind types) will now be seen by the compiler as indistinguishable — and it will object strenuously. Simply delete all the “_dp” function names from the list of overloaded procedures (i.e., from the MODULE PROCEDURE statements). Note that it is not necessary to delete the routines from the MODULE itself. Similarly, in the interface file nr.f90 you must delete the “_dp” interfaces, except for the sprs... routines. (Since they have TYPE(sprs2_dp) or TYPE(sprs2_sp), they are treated as distinct even though they have functionally equivalent kind types.)

Finally, the following table gives some suggestions for changing the accuracy parameters, or constants, in some of the routines. Please note that this table is not necessarily complete, and that higher-precision performance is not guaranteed for all the routines, even if you make all the changes indicated. The above edits, and these suggestions, do, however, work in the majority of cases.
<table>
<thead>
<tr>
<th>In routine...</th>
<th>change...</th>
<th>to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>beschb</td>
<td>NUSE1=5, NUSE2=5</td>
<td>NUSE1=7, NUSE2=8</td>
</tr>
<tr>
<td>bessi</td>
<td>IACC=40</td>
<td>IACC=200</td>
</tr>
<tr>
<td>bessik</td>
<td>EPS=1.0e-10_dp</td>
<td>EPS=epsilon(x)</td>
</tr>
<tr>
<td>bessj</td>
<td>IACC=40</td>
<td>IACC=160</td>
</tr>
<tr>
<td>bessjy</td>
<td>EPS=1.0e-10_dp</td>
<td>EPS=epsilon(x)</td>
</tr>
<tr>
<td>broydn</td>
<td>TOLF=1.0e-4_sp</td>
<td>TOLF=1.0e-8_sp</td>
</tr>
<tr>
<td></td>
<td>TOLMIN=1.0e-6_sp</td>
<td>TOLMIN=1.0e-12_sp</td>
</tr>
<tr>
<td>fdjac</td>
<td>EPS=1.0e-4_sp</td>
<td>EPS=1.0e-8_sp</td>
</tr>
<tr>
<td>frpran</td>
<td>EPS=1.0e-10_sp</td>
<td>EPS=1.0e-18_sp</td>
</tr>
<tr>
<td>gauher</td>
<td>EPS=3.0e-13_dp</td>
<td>EPS=1.0e-14_dp</td>
</tr>
<tr>
<td>gaujac</td>
<td>EPS=3.0e-14_dp</td>
<td>EPS=1.0e-14_dp</td>
</tr>
<tr>
<td>gaulag</td>
<td>EPS=3.0e-13_dp</td>
<td>EPS=1.0e-14_dp</td>
</tr>
<tr>
<td>guleg</td>
<td>EPS=3.0e-14_dp</td>
<td>EPS=1.0e-14_dp</td>
</tr>
<tr>
<td>hypgeo</td>
<td>EPS=1.0e-16_sp</td>
<td>EPS=1.0e-14_sp</td>
</tr>
<tr>
<td>linmin</td>
<td>TOL=1.0e-4_sp</td>
<td>TOL=1.0e-8_sp</td>
</tr>
<tr>
<td>newt</td>
<td>TOLF=1.0e-4_sp</td>
<td>TOLF=1.0e-8_sp</td>
</tr>
<tr>
<td></td>
<td>TOLMIN=1.0e-6_sp</td>
<td>TOLMIN=1.0e-12_sp</td>
</tr>
<tr>
<td>probks</td>
<td>EPS1=0.001_sp</td>
<td>EPS1=1.0e-6_sp</td>
</tr>
<tr>
<td></td>
<td>EPS2=1.0e-8_sp</td>
<td>EPS2=1.0e-16_sp</td>
</tr>
<tr>
<td>qromb</td>
<td>EPS=1.0e-6_sp</td>
<td>EPS=1.0e-10_sp</td>
</tr>
<tr>
<td>qromo</td>
<td>EPS=1.0e-6_sp</td>
<td>EPS=1.0e-10_sp</td>
</tr>
<tr>
<td>qroot</td>
<td>TINY=1.0e-6_sp</td>
<td>TINY=1.0e-14_sp</td>
</tr>
<tr>
<td>qsimp</td>
<td>EPS=1.0e-6_sp</td>
<td>EPS=1.0e-10_sp</td>
</tr>
<tr>
<td>qtrap</td>
<td>EPS=1.0e-6_sp</td>
<td>EPS=1.0e-10_sp</td>
</tr>
<tr>
<td>rc</td>
<td>ERRTOL=0.012_sp</td>
<td>ERRTOL=0.0012_sp</td>
</tr>
<tr>
<td>rd</td>
<td>ERRTOL=0.005_sp</td>
<td>ERRTOL=0.0015_sp</td>
</tr>
<tr>
<td>rf</td>
<td>ERRTOL=0.008_sp</td>
<td>ERRTOL=0.0025_sp</td>
</tr>
<tr>
<td>rj</td>
<td>ERRTOL=0.005_sp</td>
<td>ERRTOL=0.0015_sp</td>
</tr>
<tr>
<td>sfroid</td>
<td>conv=5.0e-6_sp</td>
<td>conv=1.0e-14_sp</td>
</tr>
<tr>
<td>shoot</td>
<td>EPS=1.0e-6_sp</td>
<td>EPS=1.0e-14_sp</td>
</tr>
<tr>
<td>shootf</td>
<td>EPS=1.0e-6_sp</td>
<td>EPS=1.0e-14_sp</td>
</tr>
<tr>
<td>simplex</td>
<td>EPS=1.0e-6_sp</td>
<td>EPS=1.0e-14_sp</td>
</tr>
<tr>
<td>sncndn</td>
<td>CA=0.0003_sp</td>
<td>CA=1.0e-8_sp</td>
</tr>
<tr>
<td>sor</td>
<td>EPS=1.0e-5_dp</td>
<td>EPS=1.0e-13_dp</td>
</tr>
<tr>
<td>sphfpt</td>
<td>dxx=1.0e-4_sp</td>
<td>dxx=1.0e-8_sp</td>
</tr>
<tr>
<td>spshoot</td>
<td>dx=1.0e-4_sp</td>
<td>dx=1.0e-8_sp</td>
</tr>
<tr>
<td>svdfit</td>
<td>TOL=1.0e-5_sp</td>
<td>TOL=1.0e-13_sp</td>
</tr>
<tr>
<td>zroots</td>
<td>EPS=1.0e-6_sp</td>
<td>EPS=1.0e-14_sp</td>
</tr>
</tbody>
</table>
C1.2 Numerical Recipes Utilities (nrutil)

The file supplied as nrutil.f90 contains a single module named nrutil, which contains specific implementations for all the Numerical Recipes utility functions described in detail in Chapter 23.

The specific implementations given are something of a compromise between demonstrating parallel techniques (when they can be achieved in Fortran 90) and running efficiently on conventional, serial machines. The parameters at the beginning of the module (names beginning with NPAR...) are typically related to array lengths below which the implementations revert to serial operations. On a purely serial machine, these can be set to large values to suppress many parallel constructions.

The length and repetitiveness of the nrutil.f90 file stems in large part from its extensive use of overloading. Indeed, the file would be even longer if we overloaded versions for all the applicable data types that each utility could, in principle, instantiate. The descriptions in Chapter 23 detail both the full set of intended data types and shapes for each routine, and also the types and shapes actually here implemented (which can also be gleaned by examining the file). The intended result of all this overloading is, in essence, to give the utility routines the desirable properties of many of the Fortran 90 intrinsic functions, namely, to be both generic (apply to many data types) and elemental (apply element-by-element to arbitrary shapes). Fortran 95’s provision of user-defined elemental functions will reduce the multiplicity of overloading in some of our routines; unfortunately the necessity to overload for multiple data types will still be present.

Finally, it is worth reemphasizing the following point, already made in Chapter 23: The purpose of the nrutil utilities is to remove from the Numerical Recipes programs just those programming tasks and “idioms” whose efficient implementation is most hardware and compiler dependent, so as to allow for specific, efficient implementations on different machines. One should therefore not expect the utmost in efficiency from the general purpose, one-size-fits-all, implementation listed here.

Correspondingly, we would encourage the incorporation of efficient nrutil implementations, and/or comparable capabilities under different names, with as broad as possible a set of overloaded data types, in libraries associated with specific compilers or machines. In support of this goal, we have specifically put this Appendix C1, and the files nrtype.f90 and nrutil.f90, into the public domain.

**MODULE nrutil**

TABLE OF CONTENTS OF THE NRUTIL MODULE:
- routines that move data:
  - array_copy, swap, reallocate
- routines returning a location as an integer value:
  - ifirstloc, imaxloc, iminloc
- routines for argument checking and error handling:
  - assert, assert_eq, nrerror
- routines relating to polynomials and recurrences:
  - arth, geop, cumsum, cumprod, poly, polyterm, zroots, unity
- routines for “outer” operations on vectors:
  - outerand, outersum, outerdiff, outerprod, outerdiv
- routines for scatter-with-combine:
  - scatter_add, scatter_max
- routines for skew operations on matrices:
  - diagadd, diammult, get_diag, put_diag,
Parameters for crossover from serial to parallel algorithms (these are used only within this nrutil module):

**IMPLICIT NONE**

```
INTEGER(I4B), PARAMETER :: NPAR_ARTH=16, NPAR2_ARTH=8         ! Each NPAR2 must be ≤ the corresponding NPAR.
INTEGER(I4B), PARAMETER :: NPAR_GEOP=4, NPAR2_GEOP=2
INTEGER(I4B), PARAMETER :: NPAR_CUMSUM=16
INTEGER(I4B), PARAMETER :: NPAR_CUMPROD=8
INTEGER(I4B), PARAMETER :: NPAR_POLY=8
INTEGER(I4B), PARAMETER :: NPAR_POLYTERM=8
```

Next, generic interfaces for routines with overloaded versions. Naming conventions for appended codes in the names of overloaded routines are as follows: \( r = \text{real, } d = \text{double precision, } \ldots \) Any of \( r,d,i,c,z,h,l \) may be followed by \( v = \text{vector or } m = \text{matrix} \) (\( v,m \) suffixes are used only when needed to resolve ambiguities).

Routines that move data:

```
INTERFACE array_copy
  MODULE PROCEDURE array_copy_r, array_copy_d, array_copy_i
END INTERFACE
```

```
INTERFACE swap
  MODULE PROCEDURE swap_i, swap_r, swap_rv, swap_c, &
               swap_cv, swap_cm, swap_z, swap_zv, swap_zm, &
               masked_swap_r, masked_swap_rv, masked_swap_cm, &
               masked_swap_z, masked_swap_zv, masked_swap_zm
END INTERFACE
```

```
INTERFACE reallocate
  MODULE PROCEDURE reallocate_r, reallocate_d, reallocate_c, &
               reallocate_cv, reallocate_cm, reallocate_z, &
               reallocate_zv, reallocate_zm
END INTERFACE
```

Routines returning a location as an integer value (ifirstloc, iminloc are not currently overloaded) and so do not have a generic interface here:

```
INTERFACE imaxloc
  MODULE PROCEDURE imaxloc_r, imaxloc_i
END INTERFACE
```

Routines for argument checking and error handling (nrerror is not currently overloaded):

```
INTERFACE assert
  MODULE PROCEDURE assert1, assert2, assert3, assert4, assert_v
END INTERFACE
```

```
INTERFACE assert_eq
  MODULE PROCEDURE assert_eq2, assert_eq3, assert_eq4, assert_eqn
END INTERFACE
```

Routines relating to polynomials and recurrences (cumprod, zroots_unity are not currently overloaded):

```
INTERFACE arth
  MODULE PROCEDURE arth_r, arth_d, arth_i
END INTERFACE
```

```
INTERFACE geop
  MODULE PROCEDURE geop_r, geop_d, geop_i, geop_c, geop_dv
END INTERFACE
```

```
INTERFACE cumsum
  MODULE PROCEDURE cumsum_r, cumsum_i
END INTERFACE
```

```
INTERFACE poly
  MODULE PROCEDURE poly_rr, poly_rv, poly_dd, poly_ddv, &
               poly_cv, poly_cm, poly_z, poly_zv, poly_zm
END INTERFACE
```

```
INTERFACE poly_term
  MODULE PROCEDURE poly_term_r, poly_term_cm
END INTERFACE
```

Routines for “outer” operations on vectors (outrand, outersum, outerdiv are not currently overloaded):

```
INTERFACE outerprod
```
MODULE PROCEDURE outerprod_r, outerprod_d
END INTERFACE

INTERFACE outerdiff
  MODULE PROCEDURE outerdiff_r, outerdiff_d, outerdiff_i
END INTERFACE

Routines for scatter-with-combine, scatter_add, scatter_max:

INTERFACE scatter_add
  MODULE PROCEDURE scatter_add_r, scatter_add_d
END INTERFACE

INTERFACE scatter_max
  MODULE PROCEDURE scatter_max_r, scatter_max_d
END INTERFACE

Routines for skew operations on matrices (unit_matrix, lower_triangle, upper_triangle not currently overloaded):

INTERFACE diagadd
  MODULE PROCEDURE diagadd_rv, diagadd_r
END INTERFACE

INTERFACE diagmult
  MODULE PROCEDURE diagmult_rv, diagmult_r
END INTERFACE

INTERFACE get_diag
  MODULE PROCEDURE get_diag_rv, get_diag_dv
END INTERFACE

INTERFACE put_diag
  MODULE PROCEDURE put_diag_rv, put_diag_r
END INTERFACE

Other routines (vabs is not currently overloaded):

CONTAINS

Routines that move data:

SUBROUTINE array_copy_r(src, dest, n_copied, n_not_copied)
  Copy array where size of source not known in advance.
  REAL(SP), DIMENSION(:,), INTENT(IN) :: src
  REAL(SP), DIMENSION(:,), INTENT(OUT) :: dest
  INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
  n_copied = min(size(src), size(dest))
  n_not_copied = size(src) - n_copied
  dest(1:n_copied) = src(1:n_copied)
END SUBROUTINE array_copy_r

SUBROUTINE array_copy_d(src, dest, n_copied, n_not_copied)
  REAL(DP), DIMENSION(:,), INTENT(IN) :: src
  REAL(DP), DIMENSION(:,), INTENT(OUT) :: dest
  INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
  n_copied = min(size(src), size(dest))
  n_not_copied = size(src) - n_copied
  dest(1:n_copied) = src(1:n_copied)
END SUBROUTINE array_copy_d

SUBROUTINE array_copy_i(src, dest, n_copied, n_not_copied)
  INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: src
  INTEGER(I4B), DIMENSION(:,), INTENT(OUT) :: dest
  INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
  n_copied = min(size(src), size(dest))
  n_not_copied = size(src) - n_copied
  dest(1:n_copied) = src(1:n_copied)
END SUBROUTINE array_copy_i

SUBROUTINE swap_i(a, b)
  Swap the contents of a and b.
  INTEGER(I4B), INTENT(INOUT) :: a, b
  INTEGER(I4B) :: dum
  dum = a
  a = b
  b = dum
END SUBROUTINE swap_i
SUBROUTINE swap_r(a,b)
REAL(SP), INTENT(INOUT) :: a,b
REAL(SP) :: dum
a=dum
b=a
END SUBROUTINE swap_r

SUBROUTINE swap_rv(a,b)
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
REAL(SP), DIMENSION(SIZE(a)) :: dum
a=dum
b=a
END SUBROUTINE swap_rv

SUBROUTINE swap_c(a,b)
COMPLEX(SPC), INTENT(INOUT) :: a,b
COMPLEX(SPC) :: dum
a=dum
b=a
END SUBROUTINE swap_c

SUBROUTINE swap_cv(a,b)
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: a,b
COMPLEX(SPC), DIMENSION(SIZE(a)) :: dum
a=dum
b=a
END SUBROUTINE swap_cv

SUBROUTINE swap_cm(a,b)
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: a,b
COMPLEX(SPC), DIMENSION(size(a,1),size(a,2)) :: dum
a=dum
b=a
END SUBROUTINE swap_cm

SUBROUTINE swap_z(a,b)
COMPLEX(DPC), INTENT(INOUT) :: a,b
COMPLEX(DPC) :: dum
a=dum
b=a
END SUBROUTINE swap_z

SUBROUTINE swap_zv(a,b)
COMPLEX(DPC), DIMENSION(:), INTENT(INOUT) :: a,b
COMPLEX(DPC), DIMENSION(SIZE(a)) :: dum
a=dum
b=a
END SUBROUTINE swap_zv

SUBROUTINE masked_swap_rng(a,b,mask)
REAL(SP), INTENT(INOUT) :: a,b
LOGICAL(LGT), INTENT(IN) :: mask
REAL(SP) :: swp

if (mask) then
    swp = a
    a = b
    b = swp
end if
END SUBROUTINE masked_swap_rs

SUBROUTINE masked_swap_rv(a,b,mask)
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a, b
    LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
    REAL(SP), DIMENSION(size(a)) :: swp
    where (mask)
        swp = a
        a = b
        b = swp
    end where
END SUBROUTINE masked_swap_rv

SUBROUTINE masked_swap_rm(a,b,mask)
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a, b
    LOGICAL(LGT), DIMENSION(:,:), INTENT(IN) :: mask
    REAL(SP), DIMENSION(size(a,1),size(a,2)) :: swp
    where (mask)
        swp = a
        a = b
        b = swp
    end where
END SUBROUTINE masked_swap_rm

FUNCTION reallocate_rv(p,n)
    Reallocate a pointer to a new size, preserving its previous contents.
    REAL(SP), DIMENSION(:), POINTER :: p, reallocate_rv
    INTEGER(I4B), INTENT(IN) :: n
    INTEGER(I4B) :: nold, ierr
    allocate(reallocate_rv(n),stat=ierr)
    if (ierr /= 0) call &
       nrerror('reallocate_rv: problem in attempt to allocate memory')
    if (.not. associated(p)) RETURN
    nold = size(p)
    reallocate_rv(1:min(nold,n))=p(1:min(nold,n))
    deallocate(p)
END FUNCTION reallocate_rv

FUNCTION reallocate_iv(p,n)
    INTEGER(I4B), DIMENSION(:), POINTER :: p, reallocate_iv
    INTEGER(I4B), INTENT(IN) :: n
    INTEGER(I4B) :: nold, ierr
    allocate(reallocate_iv(n),stat=ierr)
    if (ierr /= 0) call &
       nrerror('reallocate_iv: problem in attempt to allocate memory')
    if (.not. associated(p)) RETURN
    nold = size(p)
    reallocate_iv(1:min(nold,n))=p(1:min(nold,n))
    deallocate(p)
END FUNCTION reallocate_iv

FUNCTION reallocate_hv(p,n)
    CHARACTER(1), DIMENSION(:), POINTER :: p, reallocate_hv
    INTEGER(I4B), INTENT(IN) :: n
    INTEGER(I4B) :: nold, ierr
    allocate(reallocate_hv(n),stat=ierr)
    if (ierr /= 0) call &
       nrerror('reallocate_hv: problem in attempt to allocate memory')
    if (.not. associated(p)) RETURN
    nold = size(p)
    reallocate_hv(1:min(nold,n))=p(1:min(nold,n))
FUNCTION deallocate_hv(p)
END FUNCTION

FUNCTION reallocate_rm(p,n,m)
REAL(SP), DIMENSION(:,::), POINTER :: p, reallocate_rm
INTEGER(I4B), INTENT(IN) :: n,m
INTEGER(I4B) :: nold,mold,ierr
allocate(reallocate_rm(n,m),stat=ierr)
if (ierr /= 0) call &
rerror(‘reallocate_rm: problem in attempt to allocate memory’) if (.not. associated(p)) RETURN
nold=size(p,1)
mold=size(p,2)
reallocate_rm(1:min(nold,n),1:min(mold,m))=&
p(1:min(nold,n),1:min(mold,m))
deallocate(p)
END FUNCTION

FUNCTION reallocate_im(p,n,m)
INTEGER(I4B), DIMENSION(:,::), POINTER :: p, reallocate_im
INTEGER(I4B), INTENT(IN) :: n,m
INTEGER(I4B) :: nold,mold,ierr
allocate(reallocate_im(n,m),stat=ierr)
if (ierr /= 0) call &
rerror(‘reallocate_im: problem in attempt to allocate memory’) if (.not. associated(p)) RETURN
nold=size(p,1)
mold=size(p,2)
reallocate_im(1:min(nold,n),1:min(mold,m))=&
p(1:min(nold,n),1:min(mold,m))
deallocate(p)
END FUNCTION

FUNCTION ifirstloc(mask)
INDEX of first occurrence of .true. in a logical vector.
LOGICAL(LGT), DIMENSION(:,), INTENT(IN) :: mask
INTEGER(I4B) :: ifirstloc
INTEGER(I4B), DIMENSION(1) :: loc
loc=maxloc(merge(1,0,mask))
ifirstloc=loc(1)
if (.not. mask(ifirstloc)) ifirstloc=size(mask)+1
END FUNCTION

FUNCTION imaxloc_r(arr)
INDEX of maxloc on an array.
REAL(SP), DIMENSION(:,::), INTENT(IN) :: arr
INTEGER(I4B) :: imaxloc_r
INTEGER(I4B), DIMENSION(1) :: imax
imax=maxloc(arr(:))
imaxloc_r=imax(1)
END FUNCTION

FUNCTION imaxloc_i(iarr)
INTEGER(I4B), DIMENSION(:,::), INTENT(IN) :: iarr
INTEGER(I4B) :: imaxloc_i
INTEGER(I4B), DIMENSION(1) :: imax
imax=maxloc(iarr(:))
imaxloc_i=imax(1)
END FUNCTION

FUNCTION iminloc(arr)
INDEX of minloc on an array.
REAL(SP), DIMENSION(:,::), INTENT(IN) :: arr
INTEGER(I4B) :: iminloc
INTEGER(I4B), DIMENSION(1) :: imax
imax=minloc(arr(:))
iminloc=imax(1)
END FUNCTION
iminloc=imin(1)
END FUNCTION iminloc

Routines for argument checking and error handling:

SUBROUTINE assert1(n1,string)
    Report and die if any logical is false (used for arg range checking).
    CHARACTER(LEN=*/, INTENT(IN)) :: string
    LOGICAL, INTENT(IN) :: n1
    if (.not. n1) then
        write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
        STOP 'program terminated by assert1'
    end if
END SUBROUTINE assert1

SUBROUTINE assert2(n1,n2,string)
    CHARACTER(LEN=*/, INTENT(IN)) :: string
    LOGICAL, INTENT(IN) :: n1,n2
    if (.not. (n1 .and. n2)) then
        write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
        STOP 'program terminated by assert2'
    end if
END SUBROUTINE assert2

SUBROUTINE assert3(n1,n2,n3,string)
    CHARACTER(LEN=*/, INTENT(IN)) :: string
    LOGICAL, INTENT(IN) :: n1,n2,n3
    if (.not. (n1 .and. n2 .and. n3)) then
        write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
        STOP 'program terminated by assert3'
    end if
END SUBROUTINE assert3

SUBROUTINE assert4(n1,n2,n3,n4,string)
    CHARACTER(LEN=*/, INTENT(IN)) :: string
    LOGICAL, INTENT(IN) :: n1,n2,n3,n4
    if (.not. (n1 .and. n2 .and. n3 .and. n4)) then
        write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
        STOP 'program terminated by assert4'
    end if
END SUBROUTINE assert4

SUBROUTINE assert_v(n,string)
    CHARACTER(LEN=*/, INTENT(IN)) :: string
    LOGICAL, DIMENSION(:), INTENT(IN) :: n
    if (.not. all(n)) then
        write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
        STOP 'program terminated by assert_v'
    end if
END SUBROUTINE assert_v

FUNCTION assert_eq2(n1,n2,string)
    Report and die if integers not all equal (used for size checking).
    CHARACTER(LEN=*/, INTENT(IN)) :: string
    INTEGER, INTENT(IN) :: n1,n2
    INTEGER :: assert_eq2
    if (n1 == n2) then
        assert_eq2=n1
    else
        write (*,*) 'nrerror: an assert_eq failed with this tag:', &
        string
        STOP 'program terminated by assert_eq2'
    end if
FUNCTION assert_eq2

FUNCTION assert_eq3(n1,n2,n3,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, INTENT(IN) :: n1,n2,n3
INTEGER :: assert_eq3
if (n1 == n2 .and. n2 == n3) then
  assert_eq3=n1
else
  write ('*',*) 'nrerror: an assert_eq failed with this tag:', &
  string
  STOP 'program terminated by assert_eq3'
end if

FUNCTION assert_eq4(n1,n2,n3,n4,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, INTENT(IN) :: n1,n2,n3,n4
INTEGER :: assert_eq4
if (n1 == n2 .and. n2 == n3 .and. n3 == n4) then
  assert_eq4=n1
else
  write ('*',*) 'nrerror: an assert_eq failed with this tag:', &
  string
  STOP 'program terminated by assert_eq4'
end if

FUNCTION assert_eqn(nn,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, DIMENSION(:), INTENT(IN) :: nn
INTEGER :: assert_eqn
if (all(nn(2:) == nn(1))) then
  assert_eqn=nn(1)
else
  write ('*',*) 'nrerror: an assert_eq failed with this tag:', &
  string
  STOP 'program terminated by assert_eqn'
end if

SUBROUTINE nrerror(string)
CHARACTER(LEN=*), INTENT(IN) :: string
write ('*',*) 'nrerror: ',string
STOP 'program terminated by nrerror'

FUNCTION arth_r(first,increment,n)
REAL(SP), INTENT(IN) :: first,increment
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: arth_r
INTEGER(I4B) :: k,k2
REAL(SP) :: temp
if (n > 0) arth_r(1)=first
if (n <= NPAR_ARTH) then
  do k=2,n
    arth_r(k)=arth_r(k-1)+increment
  end do
else
  do k=2,NPAR2_ARTH
    arth_r(k)=arth_r(k-1)+increment
  end do
  temp=increment*NPAR2_ARTH
  k=NPAR2_AR T H
endif
do
  if (k >= n) exit
  k2=k+k
  arth_r(k+1:min(k2,n))=temp+arth_r(1:min(k,n-k))
  temp=temp+temp
  k=k2
end do
end if
END FUNCTION arth_r

FUNCTION arth_d(first, increment, n)
REAL(DP), INTENT(IN) :: first, increment
INTEGER(I4B), INTENT(IN) :: n
REAL(DP), DIMENSION(n) :: arth_d
INTEGER(I4B) :: k, k2
REAL(DP) :: temp
if (n > 0) arth_d(1)=first
if (n <= NPAR_ARTH) then
  do k=2,n
    arth_d(k)=arth_d(k-1)+increment
  end do
else
  do k=2,NPAR2_ARTH
    arth_d(k)=arth_d(k-1)+increment
  end do
  temp=increment*NPAR2_ARTH
  k=NPAR2_ARTH
  do
    if (k >= n) exit
    k2=k+k
    arth_d(k+1:min(k2,n))=temp+arth_d(1:min(k,n-k))
    temp=temp+temp
    k=k2
  end do
end if
END FUNCTION arth_d

FUNCTION arth_i(first, increment, n)
INTEGER(I4B), INTENT(IN) :: first, increment, n
INTEGER(I4B), DIMENSION(n) :: arth_i
INTEGER(I4B) :: k, k2, temp
if (n > 0) arth_i(1)=first
if (n <= NPAR_ARTH) then
  do k=2,n
    arth_i(k)=arth_i(k-1)+increment
  end do
else
  do k=2,NPAR2_ARTH
    arth_i(k)=arth_i(k-1)+increment
  end do
  temp=increment*NPAR2_ARTH
  k=NPAR2_ARTH
  do
    if (k >= n) exit
    k2=k+k
    arth_i(k+1:min(k2,n))=temp+arth_i(1:min(k,n-k))
    temp=temp+temp
    k=k2
  end do
end if
END FUNCTION arth_i

FUNCTION geop_r(first, factor, n)
  Array function returning a geometric progression.
REAL(SP), INTENT(IN) :: first, factor
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: geop_r
INTEGER(I4B) :: k, k2
REAL(SP) :: temp
if (n > 0) geop_r(1)=first
if (n <= NPAR_GEOP) then
  do k=2,n
    geop_r(k)=geop_r(k-1)*factor
  end do
else
  do k=2,NPAR2_GEOP
    geop_r(k)=geop_r(k-1)*factor
  end do
temp=factor**NPAR2_GEOP
  k=NPAR2_GEOP
do
    if (k >= n) exit
    k2=k+k
    geop_r(k+1:min(k2,n))=temp*geop_r(1:min(k,n-k))
    temp=temp*temp
    k=k2
  end do
end if
END FUNCTION geop_r

FUNCTION geop_d(first,factor,n)
REAL(DP), INTENT(IN) :: first,factor
INTEGER(I4B), INTENT(IN) :: n
REAL(DP), DIMENSION(n) :: geop_d
INTEGER(I4B) :: k, k2
REAL(DP) :: temp
if (n > 0) geop_d(1)=first
if (n <= NPAR_GEOP) then
  do k=2,n
    geop_d(k)=geop_d(k-1)*factor
  end do
else
  do k=2,NPAR2_GEOP
    geop_d(k)=geop_d(k-1)*factor
  end do
temp=factor**NPAR2_GEOP
  k=NPAR2_GEOP
do
    if (k >= n) exit
    k2=k+k
    geop_d(k+1:min(k2,n))=temp*geop_d(1:min(k,n-k))
    temp=temp*temp
    k=k2
  end do
end if
END FUNCTION geop_d

FUNCTION geop_i(first,factor,n)
INTEGER(I4B), INTENT(IN) :: first,factor,n
INTEGER(I4B), DIMENSION(n) :: geop_i
INTEGER(I4B) :: k, k2, temp
if (n > 0) geop_i(1)=first
if (n <= NPAR_GEOP) then
  do k=2,n
    geop_i(k)=geop_i(k-1)*factor
  end do
else
  do k=2,NPAR2_GEOP
    geop_i(k)=geop_i(k-1)*factor
  end do
temp=factor**NPAR2_GEOP
k=NPAR2_GEOP
do
  if (k >= n) exit
  k2=k+k
  geop_i(k+1:min(k2,n))=temp*geop_i(1:min(k,n-k))
  temp=temp*temp
  k=k2
end do
end if
END FUNCTION geop_i

FUNCTION geop_c(first,factor,n)
  COMPLEX(SP), INTENT(IN) :: first,factor
  INTEGER(I4B), INTENT(IN) :: n
  COMPLEX(SP), DIMENSION(n) :: geop_c
  INTEGER(I4B) :: k,k2
  COMPLEX(SP) :: temp
  if (n > 0) geop_c(1)=first
  if (n <= NPAR_GEOP) then
    do k=2,n
      geop_c(k)=geop_c(k-1)*factor
    end do
  else
    do k=2,NPAR2_GEOP
      geop_c(k)=geop_c(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
      if (k >= n) exit
      k2=k+k
      geop_c(k+1:min(k2,n))=temp*geop_c(1:min(k,n-k))
      temp=temp*temp
      k=k2
    end do
  end if
END FUNCTION geop_c

FUNCTION geop_dv(first,factor,n)
  REAL(DP), DIMENSION(:), INTENT(IN) :: first,factor
  INTEGER(I4B), INTENT(IN) :: n
  REAL(DP), DIMENSION(size(first),n) :: geop_dv
  INTEGER(I4B) :: k,k2
  REAL(DP), DIMENSION(size(first)) :: temp
  if (n > 0) geop_dv(:,1)=first(:)
  if (n <= NPAR_GEOP) then
    do k=2,n
      geop_dv(:,k)=geop_dv(:,k-1)*factor(:)
    end do
  else
    do k=2,NPAR2_GEOP
      geop_dv(:,k)=geop_dv(:,k-1)*factor(:)
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
      if (k >= n) exit
      k2=k+k
      geop_dv(:,k+1:min(k2,n))=geop_dv(:,1:min(k,n-k))*spread(temp,2,size(geop_dv(:,1:min(k,n-k)),2))
      temp=temp*temp
      k=k2
    end do
  end if
END FUNCTION geop_dv

RECURSIVE FUNCTION cumsum_r(arr,seed) RESULT(ans)
    Cumulative sum on an array, with optional additive seed.
    REAL(SP), DIMENSION(:,), INTENT(IN) :: arr
    REAL(SP), OPTIONAL, INTENT(IN) :: seed
    INTEGER(I4B) :: n,j
    REAL(SP) :: sd
    n=size(arr)
    if (n == 0_i4b) RETURN
    sd=0.0_sp
    if (present(seed)) sd=seed
    ans(1)=arr(1)+sd
    if (n < NPAR_CUMSUM) then
        do j=2,n
            ans(j)=ans(j-1)+arr(j)
        end do
    else
        ans(2:n:2)=cumsum_r(arr(2:n:2)+arr(1:n-1:2),sd)
        ans(3:n:2)=ans(2:n-1:2)+arr(3:n:2)
    end if
END FUNCTION cumsum_r

RECURSIVE FUNCTION cumsum_i(arr,seed) RESULT(ans)
    INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: arr
    INTEGER(I4B), OPTIONAL, INTENT(IN) :: seed
    INTEGER(I4B), DIMENSION(size(arr)) :: ans
    INTEGER(I4B) :: n,j,sd
    n=size(arr)
    if (n == 0_i4b) RETURN
    sd=0_i4b
    if (present(seed)) sd=seed
    ans(1)=arr(1)+sd
    if (n < NPAR_CUMSUM) then
        do j=2,n
            ans(j)=ans(j-1)+arr(j)
        end do
    else
        ans(2:n:2)=cumsum_i(arr(2:n:2)+arr(1:n-1:2),sd)
        ans(3:n:2)=ans(2:n-1:2)+arr(3:n:2)
    end if
END FUNCTION cumsum_i

RECURSIVE FUNCTION cumprod(arr,seed) RESULT(ans)
    Cumulative product on an array, with optional multiplicative seed.
    REAL(SP), DIMENSION(:,), INTENT(IN) :: arr
    REAL(SP), OPTIONAL, INTENT(IN) :: seed
    REAL(SP), DIMENSION(size(arr)) :: ans
    INTEGER(I4B) :: n,j
    REAL(SP) :: sd
    n=size(arr)
    if (n == 0_i4b) RETURN
    sd=1.0_sp
    if (present(seed)) sd=seed
    ans(1)=arr(1)*sd
    if (n < NPAR_CUMPROD) then
        do j=2,n
            ans(j)=ans(j-1)*arr(j)
        end do
    else
        ans(2:n:2)=cumprod(arr(2:n:2)*arr(1:n-1:2),sd)
        ans(3:n:2)=ans(2:n-1:2)*arr(3:n:2)
    end if
END FUNCTION cumprod
FUNCTION poly_rr(x,coeffs)
    Polynomial evaluation.
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs
    REAL(SP) :: poly_rr
    REAL(SP) :: pow
    INTEGER(I4B) :: i,n,nn
    n=size(coeffs)
    if (n <= 0) then
        poly_rr=0.0_sp
    else if (n < NPAR_POLY) then
        poly_rr=coeffs(n)
        do i=n-1,1,-1
            poly_rr=x*poly_rr+coeffs(i)
        end do
    else
        allocate(vec(n+1))
        pow=x
        vec(1:n)=coeffs
        do
            vec(n+1)=0.0_sp
            nn=ishft(n+1,-1)
            vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
            if (nn == 1) exit
            pow=pow*pow
            n=nn
        end do
        poly_rr=vec(1)
        deallocate(vec)
    end if
END FUNCTION poly_rr

FUNCTION poly_dd(x,coeffs)
    REAL(DP), INTENT(IN) :: x
    REAL(DP), DIMENSION(:), INTENT(IN) :: coeffs
    REAL(DP) :: poly_dd
    REAL(DP) :: pow
    REAL(DP), DIMENSION(:), ALLOCATABLE :: vec
    INTEGER(I4B) :: i,n,nn
    n=size(coeffs)
    if (n <= 0) then
        poly_dd=0.0_dp
    else if (n < NPAR_POLY) then
        poly_dd=coeffs(n)
        do i=n-1,1,-1
            poly_dd=x*poly_dd+coeffs(i)
        end do
    else
        allocate(vec(n+1))
        pow=x
        vec(1:n)=coeffs
        do
            vec(n+1)=0.0_dp
            nn=ishft(n+1,-1)
            vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
            if (nn == 1) exit
            pow=pow*pow
            n=nn
        end do
        poly_dd=vec(1)
        deallocate(vec)
    end if
END FUNCTION poly_dd
FUNCTION poly_rc(x,coeffs)
COMPLEX(SPC), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs
COMPLEX(SPC) :: poly_rc
COMPLEX(SPC) :: pow
COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: vec
INTEGER(I4B) :: i,n,nn
n=size(coeffs)
if (n <= 0) then
  poly_rc=0.0_sp
else if (n < NPAR_POLY) then
  poly_rc=coeffs(n)
  do i=n-1,1,-1
     poly_rc=x*poly_rc+coeffs(i)
  end do
else
  allocate(vec(n+1))
  pow=x
  vec(1:n)=coeffs
  do
     vec(n+1)=0.0_sp
     nn=ishft(n+1,-1)
     vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
     if (nn == 1) exit
     pow=pow*pow
     n=nn
  end do
  poly_rc=vec(1)
  deallocate(vec)
end if
END FUNCTION poly_rc

FUNCTION poly_cc(x,coeffs)
COMPLEX(SPC), INTENT(IN) :: x
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: coeffs
COMPLEX(SPC) :: poly_cc
COMPLEX(SPC) :: pow
COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: vec
INTEGER(I4B) :: i,n,nn
n=size(coeffs)
if (n <= 0) then
  poly_cc=0.0_sp
else if (n < NPAR_POLY) then
  poly_cc=coeffs(n)
  do i=n-1,1,-1
     poly_cc=x*poly_cc+coeffs(i)
  end do
else
  allocate(vec(n+1))
  pow=x
  vec(1:n)=coeffs
  do
     vec(n+1)=0.0_sp
     nn=ishft(n+1,-1)
     vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
     if (nn == 1) exit
     pow=pow*pow
     n=nn
  end do
  poly_cc=vec(1)
  deallocate(vec)
end if
END FUNCTION poly_cc

FUNCTION poly_rrv(x,coeffs)
REAL(SP), DIMENSION(:,), INTENT(IN) :: coeffs, x
REAL(SP), DIMENSION(size(x)) :: poly_rv
INTEGER(I4B) :: i, n, m
m=size(coeffs)
n=size(x)
if (m <= 0) then
poly_rv=0.0_sp
else if (m < n .or. m < NPAR_POLY) then
poly_rv=coeffs(m)
do i=m-1,1,-1
   poly_rv=x*poly_rv+coeffs(i)
end do
else
   do i=1,n
      poly_rv(i)=poly_rv(x(i),coeffs)
   end do
end if
END FUNCTION poly_rv

FUNCTION poly_ddv(x, coeffs)
REAL(DP), DIMENSION(:,), INTENT(IN) :: coeffs, x
REAL(DP), DIMENSION(size(x)) :: poly_ddv
INTEGER(I4B) :: i, n, m
m=size(coeffs)
n=size(x)
if (m <= 0) then
poly_ddv=0.0_dp
else if (m < n .or. m < NPAR_POLY) then
poly_ddv=coeffs(m)
do i=m-1,1,-1
   poly_ddv=x*poly_ddv+coeffs(i)
end do
else
   do i=1,n
      poly_ddv(i)=poly_dd(x(i),coeffs)
   end do
end if
END FUNCTION poly_ddv

FUNCTION poly_msk_rrv(x, coeffs, mask)
REAL(SP), DIMENSION(:,), INTENT(IN) :: coeffs, x
LOGICAL(LGT), DIMENSION(:,), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(x)) :: poly_msk_rrv
poly_msk_rrv=unpack(poly_rv(pack(x,mask),coeffs),mask,0.0_sp)
END FUNCTION poly_msk_rrv

FUNCTION poly_msk_ddv(x, coeffs, mask)
REAL(DP), DIMENSION(:,), INTENT(IN) :: coeffs, x
LOGICAL(LGT), DIMENSION(:,), INTENT(IN) :: mask
REAL(DP), DIMENSION(size(x)) :: poly_msk_ddv
poly_msk_ddv=unpack(poly_ddv(pack(x,mask),coeffs),mask,0.0_dp)
END FUNCTION poly_msk_ddv

RECURSIVE FUNCTION poly_term_rr(a, b) RESULT(u)
! Tabulate cumulants of a polynomial.
REAL(SP), DIMENSION(:,), INTENT(IN) :: a
REAL(SP), INTENT(IN) :: b
REAL(SP), DIMENSION(size(a)) :: u
INTEGER(I4B) :: n, j
n=size(a)
if (n <= 0) RETURN
u(1)=a(1)
if (n < NPAR_POLYTERM) then
   do j=2,n
      u(j)=a(j)+b*u(j-1)
   end do
else
   do i=1,n
      poly_rv(i)=poly_rv(x(i),coeffs)
   end do
end if
END FUNCTION poly_term_rr
else
  u(2:n:2)=poly_term_rr(a(2:n:2)+a(1:n-1:2)*b,b*b)
u(3:n:2)=a(3:n:2)+b*u(2:n-1:2)
end if
END FUNCTION poly_term_rr

RECURSIVE FUNCTION poly_term_cc(a,b) RESULT(u)
  COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
  COMPLEX(SPC), INTENT(IN) :: b
  COMPLEX(SPC), DIMENSION(size(a)) :: u
  INTEGER(I4B) :: n,j
  n=size(a)
  if (n <= 0) RETURN
  u(1)=a(1)
  if (n < NPAR_POLYTERM) then
    do j=2,n
      u(j)=a(j)+b*u(j-1)
    end do
  else
    u(2:n:2)=poly_term_cc(a(2:n:2)+a(1:n-1:2)*b,b*b)
    u(3:n:2)=a(3:n:2)+b*u(2:n-1:2)
  end if
END FUNCTION poly_term_cc

FUNCTION zroots_unity(n,nn)
  Complex function returning nn powers of the n-th root of unity.
  INTEGER(I4B), INTENT(IN) :: n,nn
  COMPLEX(SPC), DIMENSION(nn) :: zroots_unity
  INTEGER(I4B) :: k
  REAL(SP) :: theta
  zroots_unity(1)=1.0
  theta=TWOPI/n
  k=1
  do
    if (k >= nn) exit
    zroots_unity(k+1)=cmplx(cos(k*theta),sin(k*theta),SPC)
    zroots_unity(k+2:min(2*k,nn))=zroots_unity(k+1)*&
      zroots_unity(2:min(k,nn-k))
    k=2*k
  end do
END FUNCTION zroots_unity

Routines for "outer" operations on vectors. The order convention is: result(i,j) = first_operand(i) (op) second_operand(j).

FUNCTION outerprod_r(a,b)
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
  REAL(SP), DIMENSION(size(a),size(b)) :: outerprod_r
  outerprod_r = spread(a,dim=2,ncopies=size(b)) * &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerprod_r

FUNCTION outerprod_d(a,b)
  REAL(DP), DIMENSION(:), INTENT(IN) :: a,b
  REAL(DP), DIMENSION(size(a),size(b)) :: outerprod_d
  outerprod_d = spread(a,dim=2,ncopies=size(b)) * &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerprod_d

FUNCTION outerdiv(a,b)
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
  REAL(SP), DIMENSION(size(a),size(b)) :: outerdiv
  outerdiv = spread(a,dim=2,ncopies=size(b)) / &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiv

FUNCTION outersum(a,b)
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a),size(b)) :: outersum
outersum = spread(a,dim=2,ncopies=size(b)) + &
           spread(b,dim=1,ncopies=size(a))
END FUNCTION outersum

FUNCTION outerdiff_r(a,b)
REAL(SP), DIMENSION(:,), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a),size(b)) :: outerdiff_r
outerdiff_r = spread(a,dim=2,ncopies=size(b)) - &
                spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiff_r

FUNCTION outerdiff_d(a,b)
REAL(DP), DIMENSION(:,), INTENT(IN) :: a,b
REAL(DP), DIMENSION(size(a),size(b)) :: outerdiff_d
outerdiff_d = spread(a,dim=2,ncopies=size(b)) - &
                spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiff_d

FUNCTION outerdiff_i(a,b)
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: a,b
INTEGER(I4B), DIMENSION(size(a),size(b)) :: outerdiff_i
outerdiff_i = spread(a,dim=2,ncopies=size(b)) - &
               spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiff_i

FUNCTION outerand(a,b)
LOGICAL(LGT), DIMENSION(:,), INTENT(IN) :: a,b
LOGICAL(LGT), DIMENSION(size(a),size(b)) :: outerand
outerand = spread(a,dim=2,ncopies=size(b)) .and. &
           spread(b,dim=1,ncopies=size(a))
END FUNCTION outerand

Routines for scatter-with-combine.
SUBROUTINE scatter_add_r(dest,source,dest_index)
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dest
REAL(SP), DIMENSION(:,), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_add_r')
m=size(dest)
do j=1,n
   i=dest_index(j)
   if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
END SUBROUTINE scatter_add_r

SUBROUTINE scatter_add_d(dest,source,dest_index)
REAL(DP), DIMENSION(:,), INTENT(OUT) :: dest
REAL(DP), DIMENSION(:,), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_add_d')
m=size(dest)
do j=1,n
   i=dest_index(j)
   if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
END SUBROUTINE scatter_add_d

SUBROUTINE scatter_max_r(dest,source,dest_index)
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dest
REAL(SP), DIMENSION(:,), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_max_r')
m=size(dest)
do j=1,n
   i=dest_index(j)
   if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
END SUBROUTINE scatter_max_r

SUBROUTINE scatter_max_d(dest,source,dest_index)
REAL(DP), DIMENSION(:,), INTENT(OUT) :: dest
REAL(DP), DIMENSION(:,), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_max_d')
m=size(dest)
do j=1,n
   i=dest_index(j)
   if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
END SUBROUTINE scatter_max_d
if (i > 0 .and. i <= m) dest(i)=max(dest(i),source(j))
end do
END SUBROUTINE scatter_max_r
SUBROUTINE scatter_max_d(dest,source,dest_index)
REAL(DP), DIMENSION(:), INTENT(OUT) :: dest
REAL(DP), DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
m=size(dest)
do j=1,n
   i=dest_index(j)
   if (i > 0 .and. i <= m) dest(i)=max(dest(i),source(j))
end do
END SUBROUTINE scatter_max_d

Routines for skew operations on matrices:
SUBROUTINE diagadd_rv(mat,diag)
    Adds vector or scalar diag to the diagonal of matrix mat.
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: mat
REAL(SP), DIMENSION(:), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = assert_eq2(size(mat,1),size(mat,2),'diagadd_rv')
do j=1,n
   mat(j,j)=mat(j,j)+diag(j)
end do
END SUBROUTINE diagadd_rv
SUBROUTINE diagadd_r(mat,diag)
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: mat
REAL(SP), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = min(size(mat,1),size(mat,2))
do j=1,n
   mat(j,j)=mat(j,j)+diag
end do
END SUBROUTINE diagadd_r
SUBROUTINE diagmult_rv(mat,diag)
    Multiplies vector or scalar diag into the diagonal of matrix mat.
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: mat
REAL(SP), DIMENSION(:), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = assert_eq2(size(mat,1),size(mat,2),'diagmult_rv')
do j=1,n
   mat(j,j)=mat(j,j)*diag(j)
end do
END SUBROUTINE diagmult_rv
SUBROUTINE diagmult_r(mat,diag)
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: mat
REAL(SP), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = min(size(mat,1),size(mat,2))
do j=1,n
   mat(j,j)=mat(j,j)*diag
end do
END SUBROUTINE diagmult_r

FUNCTION get_diag_rv(mat)
    Return as a vector the diagonal of matrix mat.
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: mat
REAL(SP), INTENT(IN) :: get_diag_rv
INTEGER(I4B) :: j
j = assert_eq2(size(mat,1),size(mat,2),'get_diag_rv')
do j=1,size(mat,1)
   get_diag_rv(j)=mat(j,j)
end do
END FUNCTION get_diag_rv
FUNCTION get_diag_rv(mat)
  REAL(DP), DIMENSION(:,,:), INTENT(IN) :: mat
END FUNCTION get_diag_rv

FUNCTION get_diag_dv(mat)
  REAL(DP), DIMENSION(size(mat,1),size(mat,2)) :: get_diag_dv
  INTEGER(I4B) :: j
  j=assert_eq2(size(mat,1),size(mat,2),'get_diag_dv')
  do j=1,size(mat,1)
    get_diag_dv(j)=mat(j,j)
  end do
END FUNCTION get_diag_dv

SUBROUTINE put_diag_rv(diagv,mat)
  Set the diagonal of matrix mat to the values of a vector or scalar.
  REAL(SP), DIMENSION(:), INTENT(IN) :: diagv
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
  INTEGER(I4B) :: j,n
  n=assert_eq2(size(diagv),min(size(mat,1),size(mat,2)),'put_diag_rv')
  do j=1,n
    mat(j,j)=diagv(j)
  end do
END SUBROUTINE put_diag_rv

SUBROUTINE put_diag_r(scal,mat)
  REAL(SP), INTENT(IN) :: scal
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
  INTEGER(I4B) :: j,n
  n = min(size(mat,1),size(mat,2))
  do j=1,n
    mat(j,j)=scal
  end do
END SUBROUTINE put_diag_r

SUBROUTINE unit_matrix(mat)
  Set the matrix mat to be a unit matrix (if it is square).
  REAL(SP), DIMENSION(:,:), INTENT(OUT) :: mat
  INTEGER(I4B) :: i,n
  n=min(size(mat,1),size(mat,2))
  mat(:,:)=0.0_sp
  do i=1,n
    mat(i,i)=1.0_sp
  end do
END SUBROUTINE unit_matrix

FUNCTION upper_triangle(j,k,extra)
  Return an upper triangular logical mask.
  INTEGER(I4B), INTENT(IN) :: j,k
  INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
  LOGICAL(LGT), DIMENSION(j,k) :: upper_triangle
  INTEGER(I4B) :: n
  n=0
  if (present(extra)) n=extra
  upper_triangle=(outerdiff(arth_i(1,1,j),arth_i(1,1,k)) < n)
END FUNCTION upper_triangle

FUNCTION lower_triangle(j,k,extra)
  Return a lower triangular logical mask.
  INTEGER(I4B), INTENT(IN) :: j,k
  INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
  LOGICAL(LGT), DIMENSION(j,k) :: lower_triangle
  INTEGER(I4B) :: n
  n=0
  if (present(extra)) n=extra
  lower_triangle=(outerdiff(arth_i(1,1,j),arth_i(1,1,k)) > -n)
END FUNCTION lower_triangle

Other routines:
FUNCTION vabs(v)
    RETURN the length (ordinary $L_2$ norm) of a vector.
REAL(SP), DIMENSION(:,), INTENT(IN) :: v
REAL(SP) :: vabs
vabs=sqrt(dot_product(v,v))
END FUNCTION vabs

END MODULE nrutil
C2. Alphabetical Listing of Explicit Interfaces

The file supplied as nr.f90 contains explicit interfaces for all the Numerical Recipes routines (except those already in the module nrutil). The interfaces are in alphabetical order, by the generic interface name, if one exists, or by the specific routine name if there is no generic name.

The file nr.f90 is normally invoked via a USE statement within a main program or subroutine that references a Numerical Recipes routine. See §21.1 for an example.

```
MODULE nr
INTERFACE
  SUBROUTINE airy(x,ai,bi,aip,bip)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: ai,bi,aip,bip
  END SUBROUTINE airy
END INTERFACE

INTERFACE
  SUBROUTINE amebsa(p,y,pb,yb,ftol,func,iter,temptr)
    USE nrtype
    INTEGER(I4B), INTENT(INOUT) :: iter
    REAL(SP), INTENT(INOUT) :: yb
    REAL(SP), INTENT(IN) :: ftol,temptr
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y,pb
    REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: p
    INTERFACE
      FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP) :: func
      END FUNCTION func
    END INTERFACE
  END SUBROUTINE amebsa
END INTERFACE

INTERFACE
  SUBROUTINE amoeba(p,y,ftol,func,iter)
    USE nrtype
    INTEGER(I4B), INTENT(OUT) :: iter
    REAL(SP), INTENT(IN) :: ftol
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: p
    INTERFACE
      FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP) :: func
      END FUNCTION func
    END INTERFACE
  END SUBROUTINE amoeba
END INTERFACE
```

1384
Appendix C2. Alphabetical Listing of Explicit Interfaces

END SUBROUTINE amoeba
END INTERFACE

INTERFACE
SUBROUTINE anneal(x,y,iorder)
USE nrtype
INTEGER(I4B), DIMENSION(:,), INTENT(INOUT) :: iorder
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y
END SUBROUTINE anneal
END INTERFACE

INTERFACE
SUBROUTINE asolve(b,x,itrnsp)
USE nrtype
REAL(DP), DIMENSION(:,), INTENT(IN) :: b
REAL(DP), DIMENSION(:,), INTENT(OUT) :: x
INTEGER(I4B), INTENT(IN) :: itrnsp
END SUBROUTINE asolve
END INTERFACE

INTERFACE
SUBROUTINE atimes(x,r,itrnsp)
USE nrtype
REAL(DP), DIMENSION(:,), INTENT(IN) :: x
REAL(DP), DIMENSION(:,), INTENT(OUT) :: r
INTEGER(I4B), INTENT(IN) :: itrnsp
END SUBROUTINE atimes
END INTERFACE

INTERFACE
SUBROUTINE avevar(data,ave,var)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: data
REAL(SP), INTENT(OUT) :: ave,var
END SUBROUTINE avevar
END INTERFACE

INTERFACE
SUBROUTINE balanc(a)
USE nrtype
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: a
END SUBROUTINE balanc
END INTERFACE

INTERFACE
SUBROUTINE banbks(a,m1,m2,al,indx,b)
USE nrtype
INTEGER(I4B), INTENT(IN) :: m1,m2
INTEGER(I4B), DIMENSION(:,), INTENT(INOUT) :: indx
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: a,al
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
END SUBROUTINE banbks
END INTERFACE

INTERFACE
SUBROUTINE bandec(a,m1,m2,al,indx,d)
USE nrtype
INTEGER(I4B), INTENT(IN) :: m1,m2
INTEGER(I4B), DIMENSION(:,), INTENT(INOUT) :: indx
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: d
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: al
END SUBROUTINE bandec
END INTERFACE

INTERFACE
SUBROUTINE banmul(a,m1,m2,x,b)
USE nrtype
INTEGER(I4B), INTENT(IN) :: m1,m2
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(OUT) :: b
REAL(SP), DIMENSION(:,), INTENT(IN) :: a
END SUBROUTINE banmul
END SUBROUTINE banmul
END INTERFACE

INTERFACE
SUBROUTINE bcucof(y,y1,y2,y12,d1,d2,c)
USE nrtype
REAL(SP), INTENT(IN) :: d1,d2
REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
REAL(SP), DIMENSION(4,4), INTENT(OUT) :: c
END SUBROUTINE bcucof
END INTERFACE

INTERFACE
SUBROUTINE bcuint(y,y1,y2,y12,x1l,x1u,x2l,x2u,x1,x2,ansy,ansy1,ansy2)
USE nrtype
REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
REAL(SP), INTENT(IN) :: x1l,x1u,x2l,x2u,x1,x2
REAL(SP), INTENT(OUT) :: ansy,ansy1,ansy2
END SUBROUTINE bcuint
END INTERFACE

INTERFACE beschb
SUBROUTINE beschb_s(x,gam1,gam2,gampl,gammi)
USE nrtype
REAL(DP), INTENT(IN) :: x
REAL(DP), INTENT(OUT) :: gam1,gam2,gampl,gammi
END SUBROUTINE beschb_s
SUBROUTINE beschb_v(x,gam1,gam2,gampl,gammi)
USE nrtype
REAL(DP), DIMENSION(:), INTENT(IN) :: x
REAL(DP), DIMENSION(:), INTENT(OUT) :: gam1,gam2,gampl,gammi
END SUBROUTINE beschb_v
END INTERFACE

INTERFACE bessi
FUNCTION bessi_s(n,x)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessi_s
END FUNCTION bessi_s
FUNCTION bessi_v(n,x)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessi_v
END FUNCTION bessi_v
END INTERFACE

INTERFACE bessi0
FUNCTION bessi0_s(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessi0_s
END FUNCTION bessi0_s
FUNCTION bessi0_v(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessi0_v
END FUNCTION bessi0_v
END INTERFACE

INTERFACE bessi1
FUNCTION bessi1_s(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessi1_s
END FUNCTION bessi1_s
END INTERFACE
FUNCTION bessi1_v(x)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessi1_v
END FUNCTION bessi1_v
END INTERFACE

INTERFACE
SUBROUTINE bessik(x,xnu,ri,rk,rip,rkp)
USE nrtype
REAL(SP), INTENT(IN) :: x,xnu
REAL(SP), INTENT(OUT) :: ri,rk,rip,rkp
END SUBROUTINE bessik
END INTERFACE
END INTERFACE bessj
FUNCTION bessj_s(n,x)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessj_s
END FUNCTION bessj_s
FUNCTION bessj_v(n,x)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessj_v
END FUNCTION bessj_v
END INTERFACE bessj0
FUNCTION bessj0_s(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessj0_s
END FUNCTION bessj0_s
FUNCTION bessj0_v(x)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessj0_v
END FUNCTION bessj0_v
END INTERFACE bessj1
FUNCTION bessj1_s(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessj1_s
END FUNCTION bessj1_s
FUNCTION bessj1_v(x)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessj1_v
END FUNCTION bessj1_v
END INTERFACE

INTERFACE bessjy
SUBROUTINE bessjy_s(x,xnu,rj,ry,rjp,ryp)
USE nrtype
REAL(SP), INTENT(IN) :: x,xnu
REAL(SP), INTENT(OUT) :: rj,ry,rjp,ryp
END SUBROUTINE bessjy_s
SUBROUTINE bessjy_v(x,xnu,rj,ry,rjp,ryp)
USE nrtype
REAL(SP), INTENT(IN) :: xnu
REAL(SP), DIMENSION(:,), INTENT(IN) :: rj,ry,rjp,ryp
REAL(SP), DIMENSION(:,), INTENT(OUT) :: rj,ry,rjp,ryp
END SUBROUTINE bessjy_v
END INTERFACE

END SUBROUTINE bessjy_v
END INTERFACE

INTERFACE bessk
  FUNCTION bessk_s(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessk_s
  END FUNCTION bessk_s
  FUNCTION bessk_v(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessk_v
  END FUNCTION bessk_v
END INTERFACE

INTERFACE bessk0
  FUNCTION bessk0_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessk0_s
  END FUNCTION bessk0_s
  FUNCTION bessk0_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessk0_v
  END FUNCTION bessk0_v
END INTERFACE

INTERFACE bessk1
  FUNCTION bessk1_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessk1_s
  END FUNCTION bessk1_s
  FUNCTION bessk1_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessk1_v
  END FUNCTION bessk1_v
END INTERFACE

INTERFACE bessy
  FUNCTION bessy_s(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessy_s
  END FUNCTION bessy_s
  FUNCTION bessy_v(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessy_v
  END FUNCTION bessy_v
END INTERFACE

INTERFACE bessy0
  FUNCTION bessy0_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessy0_s
  END FUNCTION bessy0_s
  FUNCTION bessy0_v(x)
    USE nrtype

Appendix C2. Alphabetical Listing of Explicit Interfaces

REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessy0_v
END FUNCTION bessy0_v
END INTERFACE

INTERFACE bessy1
    FUNCTION bessy1_s(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: bessy1_s
        END FUNCTION bessy1_s
    
    FUNCTION bessy1_v(x)
        USE nrtype
        REAL(SP), DIMENSION(:,), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: bessy1_v
        END FUNCTION bessy1_v
END INTERFACE

INTERFACE beta
    FUNCTION beta_s(z,w)
        USE nrtype
        REAL(SP), INTENT(IN) :: z,w
        REAL(SP) :: beta_s
        END FUNCTION beta_s
    
    FUNCTION beta_v(z,w)
        USE nrtype
        REAL(SP), DIMENSION(:,), INTENT(IN) :: z,w
        REAL(SP), DIMENSION(size(z)) :: beta_v
        END FUNCTION beta_v
END INTERFACE

INTERFACE betacf
    FUNCTION betacf_s(a,b,x)
        USE nrtype
        REAL(SP), INTENT(IN) :: a,b,x
        REAL(SP) :: betacf_s
        END FUNCTION betacf_s
    
    FUNCTION betacf_v(a,b,x)
        USE nrtype
        REAL(SP), DIMENSION(:,), INTENT(IN) :: a,b,x
        REAL(SP), DIMENSION(size(x)) :: betacf_v
        END FUNCTION betacf_v
END INTERFACE

INTERFACE betai
    FUNCTION betai_s(a,b,x)
        USE nrtype
        REAL(SP), INTENT(IN) :: a,b,x
        REAL(SP) :: betai_s
        END FUNCTION betai_s
    
    FUNCTION betai_v(a,b,x)
        USE nrtype
        REAL(SP), DIMENSION(:,), INTENT(IN) :: a,b,x
        REAL(SP), DIMENSION(size(a)) :: betai_v
        END FUNCTION betai_v
END INTERFACE

INTERFACE bico
    FUNCTION bico_s(n,k)
        USE nrtype
        INTEGER(I4B), INTENT(IN) :: n,k
        REAL(SP) :: bico_s
        END FUNCTION bico_s
    
    FUNCTION bico_v(n,k)
        USE nrtype
        INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: n,k
        REAL(SP), DIMENSION(size(n)) :: bico_v
        END FUNCTION bico_v
    
    END INTERFACE
END FUNCTION bico_v
END INTERFACE

END INTERFACE
FUNCTION bnldev(pp,n)
USE nrtype
REAL(SP), INTENT(IN) :: pp
INTEGER(I4B), INTENT(IN) :: n
REAL(SP) :: bnldev
END FUNCTION bnldev
END INTERFACE

FUNCTION brent(ax,bx,cx,func,tol,xmin)
USE nrtype
REAL(SP), INTENT(IN) :: ax,bx,cx,tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: brent
END INTERFACE

FUNCTION func(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
END FUNCTION brent
END INTERFACE

SUBROUTINE broydn(x,check)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
LOGICAL(LGT), INTENT(OUT) :: check
END SUBROUTINE broydn
END INTERFACE

SUBROUTINE bsstep(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE bsstep
END INTERFACE

SUBROUTINE caldat(julian,mm,id,iyyy)
USE nrtype
INTEGER(I4B), INTENT(IN) :: julian
INTEGER(I4B), INTENT(OUT) :: mm,id,iyyy
END SUBROUTINE caldat
END INTERFACE

FUNCTION chder(a,b,c)
USE nrtype
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chder
END FUNCTION chder
Appendix C2. Alphabetical Listing of Explicit Interfaces

END INTERFACE

INTERFACE chebev
    FUNCTION chebev_s(a,b,c,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b,x
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP) :: chebev_s
    END FUNCTION chebev_s
END INTERFACE

INTERFACE chebev_v(a,b,c,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(IN) :: c,x
    REAL(SP), DIMENSION(size(x)) :: chebev_v
    END FUNCTION chebev_v
END INTERFACE

INTERFACE chebft(a,b,n,func)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(n) :: chebft
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION chebft
END INTERFACE

INTERFACE chebpc(c)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP), DIMENSION(size(c)) :: chebpc
    END FUNCTION chebpc
END INTERFACE

INTERFACE chint(a,b,c)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP), DIMENSION(size(c)) :: chint
    END FUNCTION chint
END INTERFACE

INTERFACE choldc(a,p)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: p
    END SUBROUTINE choldc
END INTERFACE

INTERFACE cholsl(a,p,b,x)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
    REAL(SP), DIMENSION(:), INTENT(IN) :: p,b
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
    END SUBROUTINE cholsl
END INTERFACE

INTERFACE chsone(bins,ebins,knstrn,df,chsq,prob)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: bins,ebins,knstrn,df,chsq
    REAL(SP), DIMENSION(:), INTENT(OUT) :: prob
    END SUBROUTINE chsone
END INTERFACE

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. ... or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).
INTEGER(I4B), INTENT(IN) :: knstrn
REAL(SP), INTENT(OUT) :: df, chsq, prob
REAL(SP), DIMENSION(:,), INTENT(IN) :: bins, ebins
END SUBROUTINE chsone

END INTERFACE

INTERFACE
SUBROUTINE chstwo(bins1, bins2, knstrn, df, chsq, prob)
USE nrtype
INTEGER(I4B), INTENT(IN) :: knstrn
REAL(SP), INTENT(OUT) :: df, chsq, prob
REAL(SP), DIMENSION(:), INTENT(IN) :: bins1, bins2
END SUBROUTINE chstwo

END INTERFACE

INTERFACE
SUBROUTINE cisi(x, ci, si)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: ci, si
END SUBROUTINE cisi

END INTERFACE

INTERFACE
SUBROUTINE cntab1(nn, chisq, df, prob, cramrv, ccc)
USE nrtype
INTEGER(I4B), DIMENSION(:,,:), INTENT(IN) :: nn
REAL(SP), INTENT(OUT) :: chisq, df, prob, cramrv, ccc
END SUBROUTINE cntab1

END INTERFACE

INTERFACE
SUBROUTINE cntab2(nn, h, hx, hy, hygx, hxgy, uygx, uxgy, uxy)
USE nrtype
INTEGER(I4B), DIMENSION(:,,:), INTENT(IN) :: nn
REAL(SP), INTENT(OUT) :: h, hx, hy, hygx, hxgy, uygx, uxgy, uxy
END SUBROUTINE cntab2

END INTERFACE

INTERFACE
FUNCTION convlv(data, respsns, isign)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: data
REAL(SP), DIMENSION(:), INTENT(IN) :: respsns
INTEGER(I4B), INTENT(IN) :: isign
REAL(SP), DIMENSION(size(data)) :: convlv
END FUNCTION convlv

END INTERFACE

INTERFACE
FUNCTION correl(data1, data2)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: data1, data2
REAL(SP), DIMENSION(size(data1)) :: correl
END FUNCTION correl

END INTERFACE

INTERFACE
SUBROUTINE cosft1(y)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: y
END SUBROUTINE cosft1

END INTERFACE

INTERFACE
SUBROUTINE cosft2(y, isign)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: y
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE cosft2

END INTERFACE

INTERFACE
SUBROUTINE covsrt(covar,maska)
USE nrtype
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: covar
LOGICAL(LGT), DIMENSION(:,), INTENT(IN) :: maska
END SUBROUTINE covsrt
END INTERFACE

INTERFACE
SUBROUTINE cyclic(a,b,c,alpha,beta,r,x)
USE nrtype
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a,b,c,r
REAL(SP), INTENT(IN) :: alpha,beta
REAL(SP), DIMENSION(:,), INTENT(OUT) :: x
END SUBROUTINE cyclic
END INTERFACE

INTERFACE
SUBROUTINE daub4(a,isign)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE daub4
END INTERFACE

INTERFACE
dawson
  FUNCTION dawson_s(x)
  USE nrtype
  REAL(SP), INTENT(IN) :: x
  REAL(SP) :: dawson_s
  END FUNCTION dawson_s
  FUNCTION dawson_v(x)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: dawson_v
  END FUNCTION dawson_v
END INTERFACE

INTERFACE
FUNCTION dbrent(ax, bx, cx, func, dbrent_dfunc, tol, xmin)
USE nrtype
REAL(SP), INTENT(IN) :: ax, bx, cx, tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: dbrent
INTERFACE
  FUNCTION func(x)
  USE nrtype
  REAL(SP), INTENT(IN) :: x
  REAL(SP) :: func
  END FUNCTION func
  FUNCTION dbrent_dfunc(x)
  USE nrtype
  REAL(SP), INTENT(IN) :: x
  REAL(SP) :: dbrent_dfunc
  END FUNCTION dbrent_dfunc
END INTERFACE
END FUNCTION dbrent
END INTERFACE

INTERFACE
ddpoly(c, x, pd)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: c
REAL(SP), DIMENSION(:,), INTENT(OUT) :: pd
END SUBROUTINE ddpoly
END INTERFACE

INTERFACE
FUNCTION decchk(string, ch)
END INTERFACE
USE nrtype
CHARACTER(1), DIMENSION(:), INTENT(IN) :: string
CHARACTER(1), INTENT(INOUT) :: ch
LOGICAL(LGT) :: decchk
END FUNCTION decchk
END INTERFACE

INTERFACE
SUBROUTINE dfpmin(p,gtol,iter,fret,func,dfunc)
USE nrtype
INTEGER(I4B), INTENT(INOUT) :: iter
REAL(SP), INTENT(IN) :: gtol
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
INTERFACE
FUNCTION func(p)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: p
REAL(SP) :: func
END FUNCTION func
FUNCTION dfunc(p)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: p
REAL(SP), DIMENSION(size(p)) :: dfunc
END FUNCTION dfunc
END INTERFACE
END SUBROUTINE dfpmin
END INTERFACE

INTERFACE
SUBROUTINE dfridr(func,x,h,err)
USE nrtype
REAL(SP), INTENT(IN) :: x,h
REAL(SP), INTENT(OUT) :: err
REAL(SP) :: dfridr
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
END FUNCTION dfridr
END INTERFACE

INTERFACE
SUBROUTINE dftcor(w,delta,a,b,endpts,corre,corim,corfac)
USE nrtype
REAL(SP), INTENT(IN) :: w,delta,a,b
REAL(SP), INTENT(OUT) :: corre,corim,corfac
REAL(SP), DIMENSION(:), INTENT(INOUT) :: endpts
END SUBROUTINE dftcor
END INTERFACE

INTERFACE
SUBROUTINE dftint(func,a,b,w,cosint,sinint)
USE nrtype
REAL(SP), INTENT(IN) :: a,b,w
REAL(SP), INTENT(OUT) :: cosint,sinint
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
END SUBROUTINE dftint
END INTERFACE

INTERFACE
SUBROUTINE difeq(k,k1,k2,jsf,is1,isf,indexv,s,y)
USE nrtype
INTEGER(I4B), INTENT(IN) :: is1,isf,jsf,k,k1,k2
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: indexv
REAL(SP), DIMENSION(:,), INTENT(OUT) :: s
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
END SUBROUTINE difeq
END INTERFACE

INTERFACE
FUNCTION eclass(lista,listb,n)
USE nrtype
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: lista,listb
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), DIMENSION(n) :: eclass
END FUNCTION eclass
END INTERFACE

INTERFACE
FUNCTION eclazz(equiv,n)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), DIMENSION(n) :: eclazz
END FUNCTION eclazz
END INTERFACE

INTERFACE
FUNCTION ei(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: ei
END FUNCTION ei
END INTERFACE

INTERFACE
SUBROUTINE eigsrt(d,v)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: v
END SUBROUTINE eigsrt
END INTERFACE

INTERFACE elle
FUNCTION elle_s(phi,ak)
USE nrtype
REAL(SP), INTENT(IN) :: phi,ak
REAL(SP) :: elle_s
END FUNCTION elle_s

FUNCTION elle_v(phi,ak)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: phi,ak
REAL(SP), DIMENSION(size(phi)) :: elle_v
END FUNCTION elle_v
END INTERFACE

INTERFACE ellf
FUNCTION ellf_s(phi,ak)
USE nrtype
REAL(SP), INTENT(IN) :: phi,ak
REAL(SP) :: ellf_s
END FUNCTION ellf_s
END INTERFACE
FUNCTION ellf_s
END INTERFACE

INTERFACE ellf_v
FUNCTION ellf_v(phi,ak)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: phi,ak
REAL(SP), DIMENSION(size(phi)) :: ellf_v
END FUNCTION ellf_v
END INTERFACE

INTERFACE ellpi
FUNCTION ellpi_s(phi,en,ak)
USE nrtype
REAL(SP), INTENT(IN) :: phi,en,ak
REAL(SP) :: ellpi_s
END FUNCTION ellpi_s
FUNCTION ellpi_v(phi,en,ak)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: phi,en,ak
REAL(SP), DIMENSION(size(phi)) :: ellpi_v
END FUNCTION ellpi_v
END INTERFACE

INTERFACE elmhes
SUBROUTINE elmhes(a)
USE nrtype
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: a
END SUBROUTINE elmhes
END INTERFACE

INTERFACE erf
FUNCTION erf_s(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: erf_s
END FUNCTION erf_s
FUNCTION erf_v(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: erf_v
END FUNCTION erf_v
END INTERFACE

INTERFACE erfc
FUNCTION erfc_s(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: erfc_s
END FUNCTION erfc_s
FUNCTION erfc_v(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: erfc_v
END FUNCTION erfc_v
END INTERFACE

INTERFACE erfcc
FUNCTION erfcc_s(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: erfcc_s
END FUNCTION erfcc_s
FUNCTION erfcc_v(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: erfcc_v
END FUNCTION erfcc_v
END INTERFACE

INTERFACE
SUBROUTINE eulsum(sum, term, jterm)
USE nrtype
REAL(SP), INTENT(INOUT) :: sum
REAL(SP), INTENT(IN) :: term
INTEGER(I4B), INTENT(IN) :: jterm
END SUBROUTINE eulsum

END INTERFACE

INTERFACE
FUNCTION evlmem(fdt, d, xms)
USE nrtype
REAL(SP), INTENT(IN) :: fdt, xms
REAL(SP), DIMENSION(:), INTENT(IN) :: d
REAL(SP) :: evlmem
END FUNCTION evlmem
END INTERFACE

INTERFACE expdev
SUBROUTINE expdev_s(harvest)
USE nrtype
REAL(SP), INTENT(OUT) :: harvest
END SUBROUTINE expdev_s
SUBROUTINE expdev_v(harvest)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
END SUBROUTINE expdev_v
END INTERFACE

INTERFACE
FUNCTION expint(n, x)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: expint
END FUNCTION expint
END INTERFACE

INTERFACE factln
FUNCTION factln_s(n)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
REAL(SP) :: factln_s
END FUNCTION factln_s
FUNCTION factln_v(n)
USE nrtype
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
REAL(SP), DIMENSION(size(n)) :: factln_v
END FUNCTION factln_v
END INTERFACE

INTERFACE factrl
FUNCTION factrl_s(n)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
REAL(SP) :: factrl_s
END FUNCTION factrl_s
FUNCTION factrl_v(n)
USE nrtype
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
REAL(SP), DIMENSION(size(n)) :: factrl_v
END FUNCTION factrl_v
END INTERFACE

INTERFACE
SUBROUTINE fasper(x, y, ofac, hifac, px, py, jmax, prob)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x, y
REAL(SP), INTENT(IN) :: ofac, hifac
INTEGER(I4B), INTENT(OUT) :: jmax
END SUBROUTINE fasper

END INTERFACE
REAL(SP), INTENT(OUT) :: prob
REAL(SP), DIMENSION(:,), POINTER :: px,py
END SUBROUTINE fasper
END INTERFACE

INTERFACE
SUBROUTINE fdjac(x,fvec,df)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: fvec
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: x
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: df
END SUBROUTINE fdjac
END INTERFACE

INTERFACE
SUBROUTINE fgauss(x,a,y,dyda)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,a
REAL(SP), DIMENSION(:,), INTENT(OUT) :: y
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: dyda
END SUBROUTINE fgauss
END INTERFACE

INTERFACE
SUBROUTINE fit(x,y,a,b,siga,sigb,chi2,q,sig)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y
REAL(SP), INTENT(OUT) :: a,b,siga,sigb,chi2,q
REAL(SP), DIMENSION(:,), OPTIONAL, INTENT(IN) :: sig
END SUBROUTINE fit
END INTERFACE

INTERFACE
SUBROUTINE fitexy(x,y,sigx,sigy,a,b,siga,sigb,chi2,q)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y,sigx,sigy
REAL(SP), INTENT(OUT) :: a,b,siga,sigb,chi2,q
END SUBROUTINE fitexy
END INTERFACE

INTERFACE
SUBROUTINE fixrts(d)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: d
END SUBROUTINE fixrts
END INTERFACE

INTERFACE
FUNCTION fleg(x,n)
USE nrtype
REAL(SP), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: fleg
END FUNCTION fleg
END INTERFACE

INTERFACE
SUBROUTINE flmoon(n,nph,jd,frac)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n,nph
INTEGER(I4B), INTENT(OUT) :: jd
REAL(SP), INTENT(OUT) :: frac
END SUBROUTINE flmoon
END INTERFACE

INTERFACE four1
SUBROUTINE four1_dp(data,isign)
USE nrtype
COMPLEX(DPC), DIMENSION(:,), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE four1_dp
END INTERFACE
SUBROUTINE four1_sp(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE four1_sp

END INTERFACE

INTERFACE
SUBROUTINE four1_alt(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE four1_alt

END INTERFACE

INTERFACE
SUBROUTINE four1_gather(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE four1_gather

END INTERFACE

INTERFACE
SUBROUTINE four2(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE four2

END INTERFACE

INTERFACE
SUBROUTINE four2_alt(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE four2_alt

END INTERFACE

INTERFACE
SUBROUTINE four3(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE four3

END INTERFACE

INTERFACE
SUBROUTINE four3_alt(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE four3_alt

END INTERFACE

INTERFACE
SUBROUTINE fourcol(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE fourcol

END INTERFACE

INTERFACE
SUBROUTINE fourcol_3d(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:,:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE fourcol_3d

END INTERFACE

INTERFACE
SUBROUTINE fourn_gather(data,nn,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE fourn_gather

END INTERFACE
USE nrtype

COMPLEX(SPC), DIMENSION(:,), INTENT(INOUT) :: data
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: nn
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE fourn_gather

END INTERFACE

INTERFACE fournrow

SUBROUTINE fournrow_dp(data,isign)
USE nrtype
COMPLEX(DPC), DIMENSION(:,,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE fournrow_dp

SUBROUTINE fournrow_sp(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(:,,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE fournrow_sp

END INTERFACE

INTERFACE fournrow_3d

SUBROUTINE fournrow_3d(data,isign)
USE nrtype
COMPLEX(SPC), DIMENSION(::,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE fournrow_3d

END INTERFACE

INTERFACE

FUNCTION fpoly(x,n)
USE nrtype
REAL(SP), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: fpoly
END FUNCTION fpoly

END INTERFACE

INTERFACE

SUBROUTINE fred2(a,b,t,f,w,g,ak)
USE nrtype
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(OUT) :: t,f,w
FUNCTION g(t)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: t
REAL(SP), DIMENSION(size(t)) :: g
END FUNCTION g

FUNCTION ak(t,s)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: t,s
REAL(SP), DIMENSION(size(t),size(s)) :: ak
END FUNCTION ak

END SUBROUTINE fred2

END INTERFACE

INTERFACE

FUNCTION fredin(x,a,b,t,f,w,g,ak)
USE nrtype
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: x,t,f,w
REAL(SP), DIMENSION(size(x)) :: fredin
FUNCTION g(t)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: t
REAL(SP), DIMENSION(size(t)) :: g
END FUNCTION g

END INTERFACE

END SUBROUTINE fred2

END INTERFACE
END FUNCTION g
FUNCTION ak(t,s)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: t, s
REAL(SP), DIMENSION(size(t), size(s)) :: ak
END FUNCTION ak
END INTERFACE
END FUNCTION fredin
END INTERFACE
INTERFACE
SUBROUTINE frenel(x,s,c)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: s, c
END SUBROUTINE frenel
END INTERFACE
SUBROUTINE frprmn(p, ftol, iter, fret)
USE nrtype
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: p
END SUBROUTINE frprmn
END INTERFACE
SUBROUTINE ftest(data1, data2, f, prob)
USE nrtype
REAL(SP), INTENT(OUT) :: f, prob
REAL(SP), DIMENSION(:,), INTENT(IN) :: data1, data2
END SUBROUTINE ftest
END INTERFACE
INTERFACE
FUNCTION gamdev(ia)
USE nrtype
INTEGER(I4B), INTENT(IN) :: ia
REAL(SP) :: gamdev
END FUNCTION gamdev
END INTERFACE
FUNCTION gamln
FUNCTION gamln_s(xx)
USE nrtype
REAL(SP), INTENT(IN) :: xx
REAL(SP) :: gamln_s
END FUNCTION gamln_s
FUNCTION gamln_v(xx)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: xx
REAL(SP), DIMENSION(size(xx)) :: gamln_v
END FUNCTION gamln_v
END INTERFACE
FUNCTION gammp_s(a, x)
USE nrtype
REAL(SP), INTENT(IN) :: a, x
REAL(SP) :: gammp_s
END FUNCTION gammp_s
FUNCTION gammp_v(a, x)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: a, x
REAL(SP), DIMENSION(size(a)) :: gammp_v
END FUNCTION gammp_v
END INTERFACE
INTERFACE gammq
    FUNCTION gammq_s(a,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,x
    REAL(SP) :: gammq_s
    END FUNCTION gammq_s
    FUNCTION gammq_v(a,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
    REAL(SP), DIMENSION(size(a)) :: gammq_v
    END FUNCTION gammq_v
END INTERFACE

INTERFACE gasdev
    SUBROUTINE gasdev_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE gasdev_s
    SUBROUTINE gasdev_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE gasdev_v
END INTERFACE

INTERFACE
    SUBROUTINE gaucof(a,b,amu0,x,w)
    USE nrtype
    REAL(SP), INTENT(IN) :: amu0
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gaucof
END INTERFACE

INTERFACE
    SUBROUTINE gauher(x,w)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gauher
END INTERFACE

INTERFACE
    SUBROUTINE gaujac(x,w,alf,bet)
    USE nrtype
    REAL(SP), INTENT(IN) :: alf,bet
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gaujac
END INTERFACE

INTERFACE
    SUBROUTINE gaulag(x,w,alf)
    USE nrtype
    REAL(SP), INTENT(IN) :: alf
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gaulag
END INTERFACE

INTERFACE
    SUBROUTINE gauleg(x1,x2,x,w)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gauleg
END INTERFACE

INTERFACE
    SUBROUTINE gaussj(a,b)
    USE nrtype
    REAL(SP), DIMENSION(:,), INTENT(INOUT) :: a,b
    END SUBROUTINE gaussj
END INTERFACE
INTERFACE gcf
  FUNCTION gcf_s(a,x,gln)
  USE nrtype
  REAL(SP), INTENT(IN) :: a,x
  REAL(SP), OPTIONAL, INTENT(OUT) :: gln
  REAL(SP) :: gcf_s
  END FUNCTION gcf_s
  FUNCTION gcf_v(a,x,gln)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
  REAL(SP), DIMENSION(:), OPTIONAL, INTENT(OUT) :: gln
  REAL(SP), DIMENSION(size(a)) :: gcf_v
  END FUNCTION gcf_v
END INTERFACE

INTERFACE
  FUNCTION golden(ax,bx,cx,func,tol,xmin)
  USE nrtype
  REAL(SP), INTENT(IN) :: ax,bx,cx,tol
  REAL(SP), INTENT(OUT) :: xmin
  REAL(SP) :: golden
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
  END FUNCTION func
  END INTERFACE
  END FUNCTION golden
END INTERFACE

INTERFACE gser
  FUNCTION gser_s(a,x,gln)
  USE nrtype
  REAL(SP), INTENT(IN) :: a,x
  REAL(SP), OPTIONAL, INTENT(OUT) :: gln
  REAL(SP) :: gser_s
  END FUNCTION gser_s
  FUNCTION gser_v(a,x,gln)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
  REAL(SP), DIMENSION(:), OPTIONAL, INTENT(OUT) :: gln
  REAL(SP), DIMENSION(size(a)) :: gser_v
  END FUNCTION gser_v
END INTERFACE

INTERFACE
  SUBROUTINE hqr(a,wr,wi)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(OUT) :: wr,wi
  REAL(SP), DIMENSION(:,::) :: a
  END SUBROUTINE hqr
END INTERFACE

INTERFACE
  SUBROUTINE hunt(xx,x,jlo)
  USE nrtype
  INTEGER(I4B), INTENT(INOUT) :: jlo
  REAL(SP), INTENT(IN) :: x
  REAL(SP), DIMENSION(:,::) :: xx
  END SUBROUTINE hunt
END INTERFACE

INTERFACE
  SUBROUTINE hypdrv(s,ry,rdyds)
  USE nrtype
  REAL(SP), INTENT(IN) :: s
  REAL(SP), DIMENSION(:,::) :: ry
  END SUBROUTINE hypdrv
END INTERFACE
REAL(SP), DIMENSION(:,), INTENT(OUT) :: rdyds
END SUBROUTINE hypdrv

END INTERFACE

INTERFACE
FUNCTION hypgeo(a,b,c,z)
USE nrtype
COMPLEX(SPC), INTENT(IN) :: a,b,c,z
COMPLEX(SPC) :: hypgeo
END FUNCTION hypgeo

END INTERFACE

INTERFACE
SUBROUTINE hypser(a,b,c,z,series,deriv)
USE nrtype
COMPLEX(SPC), INTENT(IN) :: a,b,c,z
COMPLEX(SPC), INTENT(OUT) :: series,deriv
END SUBROUTINE hypser

END INTERFACE

INTERFACE
FUNCTION icrc(crc,buf,jinit,jrev)
USE nrtype
CHARACTER(1), DIMENSION(:,), INTENT(IN) :: buf
INTEGER(I2B), INTENT(IN) :: crc,jinit
INTEGER(I4B), INTENT(IN) :: jrev
INTEGER(I2B) :: icrc
END FUNCTION icrc

END INTERFACE

INTERFACE
FUNCTION igray(n,is)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n,is
INTEGER(I4B) :: igray
END FUNCTION igray

END INTERFACE

INTERFACE
RECURSIVE SUBROUTINE index_bypack(arr,index,partial)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: arr
INTEGER(I4B), DIMENSION(:,), INTENT(INOUT) :: index
INTEGER, OPTIONAL, INTENT(IN) :: partial
END SUBROUTINE index_bypack

END INTERFACE

INTERFACE indexx
SUBROUTINE indexx_sp(arr,index)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: arr
INTEGER(I4B), DIMENSION(:,), INTENT(OUT) :: index
END SUBROUTINE indexx_sp

SUBROUTINE indexx_i4b(iarr,index)
USE nrtype
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: iarr
INTEGER(I4B), DIMENSION(:,), INTENT(OUT) :: index
END SUBROUTINE indexx_i4b

END INTERFACE

INTERFACE
FUNCTION interp(uc)
USE nrtype
REAL(DP), DIMENSION(:,,:), INTENT(IN) :: uc
REAL(DP), DIMENSION(2*size(uc,1)-1,2*size(uc,1)-1) :: interp
END FUNCTION interp

END INTERFACE

INTERFACE
FUNCTION rank(indx)
USE nrtype
INTEGER(I4B), DIMENSION(:,), INTENT(IN) :: indx
END FUNCTION rank

END INTERFACE
Appendix C2. Alphabetical Listing of Explicit Interfaces

INTEGER(I4B), DIMENSION(size(indx)) :: rank
END FUNCTION rank

END INTERFACE

INTERFACE
FUNCTION irbit1(iseed)
USE nrtype
INTEGER(I4B), INTENT(INOUT) :: iseed
INTEGER(I4B) :: irbit1
END FUNCTION irbit1

END INTERFACE

INTERFACE
FUNCTION irbit2(iseed)
USE nrtype
INTEGER(I4B), INTENT(INOUT) :: iseed
INTEGER(I4B) :: irbit2
END FUNCTION irbit2

END INTERFACE

INTERFACE
SUBROUTINE jacobi(a,d,v,nrot)
USE nrtype
INTEGER(I4B), INTENT(OUT) :: nrot
REAL(SP), DIMENSION(:), INTENT(OUT) :: d
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
END SUBROUTINE jacobi

END INTERFACE

INTERFACE
SUBROUTINE jacobn(x,y,dfdx,dfdy)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
END SUBROUTINE jacobn

END INTERFACE

INTERFACE
FUNCTION julday(mm,id,iyyy)
USE nrtype
INTEGER(I4B), INTENT(IN) :: mm,id,iyyy
INTEGER(I4B) :: julday
END FUNCTION julday

END INTERFACE

INTERFACE
SUBROUTINE kendl1(data1,data2,tau,z,prob)
USE nrtype
REAL(SP), INTENT(OUT) :: tau,z,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
END SUBROUTINE kendl1

END INTERFACE

INTERFACE
SUBROUTINE kendl2(tab,tau,z,prob)
USE nrtype
REAL(SP), DIMENSION(:,:), INTENT(IN) :: tab
REAL(SP), INTENT(OUT) :: tau,z,prob
END SUBROUTINE kendl2

END INTERFACE

INTERFACE
FUNCTION kermom(y,m)
USE nrtype
REAL(DP), INTENT(IN) :: y
INTEGER(I4B), INTENT(IN) :: m
REAL(DP), DIMENSION(m) :: kermom
END FUNCTION kermom

END INTERFACE
INTERFACE
SUBROUTINE ks2d1s(x1,y1,quadvl,d1,prob)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x1,y1
REAL(SP), INTENT(OUT) :: d1,prob
INTERFACE
SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
USE nrtype
REAL(SP), INTENT(IN) :: x,y
REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
END SUBROUTINE quadvl
END INTERFACE
END SUBROUTINE ks2d1s
END INTERFACE
INTERFACE
SUBROUTINE ks2d2s(x1,y1,x2,y2,d,prob)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x1,y1,x2,y2
REAL(SP), INTENT(OUT) :: d,prob
END SUBROUTINE ks2d2s
END INTERFACE
INTERFACE
SUBROUTINE kstwo(data1,data2,d,prob)
USE nrtype
REAL(SP), INTENT(OUT) :: d,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
END SUBROUTINE kstwo
END INTERFACE
INTERFACE
SUBROUTINE laguer(a,x,its)
USE nrtype
INTEGER(I4B), INTENT(OUT) :: its
COMPLEX(SPC), INTENT(INOUT) :: x
COMPLEX(SPC), DIMENSION(:,), INTENT(IN) :: a
END SUBROUTINE laguer
END INTERFACE
INTERFACE
SUBROUTINE lfit(x,y,sig,a,maska,covar,chisq,funcs)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y,sig
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: a
LOGICAL(LGT), DIMENSION(:,), INTENT(IN) :: maska
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: covar
REAL(SP), INTENT(OUT) :: chisq
INTERFACE
SUBROUTINEfuncs(x,arr)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(OUT) :: arr
END SUBROUTINE funcs
END INTERFACE
END INTERFACE
END SUBROUTINE lfit
END INTERFACE

INTERFACE
SUBROUTINE linbcg(b,x,itol,tol,itmax,iter,err)
USE nrtype
REAL(DP), DIMENSION(:), INTENT(IN) :: b
REAL(DP), DIMENSION(:), INTENT(INOUT) :: x
INTEGER(I4B), INTENT(IN) :: itol,itmax
REAL(DP), INTENT(IN) :: tol
INTEGER(I4B), INTENT(INOUT) :: iter
REAL(DP), INTENT(INOUT) :: err
END SUBROUTINE linbcg
END INTERFACE

INTERFACE
SUBROUTINE linmin(p,xi,fret)
USE nrtype
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), TARGET, INTENT(INOUT) :: p,xi
END SUBROUTINE linmin
END INTERFACE

INTERFACE
SUBROUTINE lnsrch(xold,fold,g,p,x,f,stpmax,check,func)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: xold,g
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
REAL(SP), INTENT(IN) :: fold,stpmax
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
REAL(SP), INTENT(OUT) :: f
LOGICAL(LGT), INTENT(OUT) :: check
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP) :: func
REAL(SP), DIMENSION(:), TARGET :: x
END FUNCTION func
END INTERFACE
END SUBROUTINE lnsrch
END INTERFACE

INTERFACE
FUNCTION locate(xx,x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: xx
REAL(SP), INTENT(IN) :: x
INTEGER(I4B) :: locate
END FUNCTION locate
END INTERFACE

INTERFACE
FUNCTION lop(u)
USE nrtype
REAL(DP), DIMENSION(:,:), INTENT(IN) :: u
REAL(DP), DIMENSION(size(u,1),size(u,1)) :: lop
END FUNCTION lop
END INTERFACE

INTERFACE
SUBROUTINE lubksb(a,indx,b)
USE nrtype
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
END SUBROUTINE lubksb
END INTERFACE

INTERFACE
SUBROUTINE ludcmp(a,indx,d)
USE nrtype
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
END SUBROUTINE ludcmp
END INTERFACE
USE nrtype
REAL(SP), DIMENSION(:, :), INTENT(INOUT) :: a
INTEGER(14B), DIMENSION(:, :), INTENT(INOUT) :: index
REAL(SP), INTENT(OUT) :: d
END SUBROUTINE ludcmp
END INTERFACE

INTERFACE
SUBROUTINE machar(ibeta, irnd, ngrd, machep, negep, iexp, minexp, &
               maxexp, eps, epsneg, xmin, xmax)
USE nrtype
INTEGER(I4B), INTENT(OUT) :: ibeta, iexp, irnd, it, machep, maxexp, &
               minexp, negep, ngrd
REAL(SP), INTENT(OUT) :: eps, epsneg, xmax, xmin
END SUBROUTINE machar
END INTERFACE

INTERFACE
SUBROUTINE medfit(x, y, a, b, abdev)
USE nrtype
REAL(SP), DIMENSION(:, :), INTENT(IN) :: x, y
REAL(SP), INTENT(OUT) :: a, b, abdev
END SUBROUTINE medfit
END INTERFACE

INTERFACE
SUBROUTINE memcof(data, xms, d)
USE nrtype
REAL(SP), INTENT(OUT) :: xms
REAL(SP), DIMENSION(:, :), INTENT(IN) :: data
REAL(SP), DIMENSION(:, :), INTENT(OUT) :: d
END SUBROUTINE memcof
END INTERFACE

INTERFACE
SUBROUTINE mgfas(u, maxcyc)
USE nrtype
REAL(DP), DIMENSION(:, :, :), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: maxcyc
END SUBROUTINE mgfas
END INTERFACE

INTERFACE
SUBROUTINE mglin(u, ncycle)
USE nrtype
REAL(DP), DIMENSION(:, :, :), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: ncycle
END SUBROUTINE mglin
END INTERFACE

INTERFACE
SUBROUTINE midexp(funk, aa, bb, s, n)
USE nrtype
REAL(SP), INTENT(IN) :: aa, bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
FUNCTION funk(x)
USE nrtype
REAL(SP), DIMENSION(:, :), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: funk
END FUNCTION funk
END INTERFACE
END SUBROUTINE midexp
END INTERFACE

INTERFACE
SUBROUTINE midinf(funk, aa, bb, s, n)
USE nrtype
REAL(SP), INTENT(IN) :: aa, bb
REAL(SP), INTENT(INOUT) :: s
FUNCTION funk(x)
USE nrtype
REAL(SP), DIMENSION(:, :), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: funk
END FUNCTION funk
END INTERFACE
END SUBROUTINE midinf
END INTERFACE
INTEGER(4B), INTENT(IN) :: n
INTERFACE
  FUNCTION funk(x)
  USE nrtype
  REAL(SP), DIMENSION(:,), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: funk
END FUNCTION funk
END INTERFACE
END SUBROUTINE midinf
END INTERFACE
INTERFACE
SUBROUTINE midpnt(func,a,b,s,n)
USE nrtype
REAL(SP), INTENT(IN) :: a,b
REAL(SP), INTENT(INOUT) :: s
INTEGER(4B), INTENT(IN) :: n
INTERFACE
  FUNCTION func(x)
  USE nrtype
  REAL(SP), DIMENSION(:,), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
END SUBROUTINE midpnt
END INTERFACE
INTERFACE
SUBROUTINE midsql(funk,aa,bb,s,n)
USE nrtype
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(4B), INTENT(IN) :: n
INTERFACE
  FUNCTION funk(x)
  USE nrtype
  REAL(SP), DIMENSION(:,), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION funk
END INTERFACE
END SUBROUTINE midsql
END INTERFACE
INTERFACE
SUBROUTINE midsqu(funk,aa,bb,s,n)
USE nrtype
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(4B), INTENT(IN) :: n
INTERFACE
  FUNCTION funk(x)
  USE nrtype
  REAL(SP), DIMENSION(:,), INTENT(IN) :: x
  REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION funk
END INTERFACE
END SUBROUTINE midsqu
END INTERFACE
INTERFACE
RECURSIVE SUBROUTINE miser(func,regn,ndim,npts,dith,ave,var)
USE nrtype
INTERFACE
  FUNCTION func(x)
  USE nrtype
  REAL(SP) :: func
  REAL(SP), DIMENSION(:,), INTENT(IN) :: x
END FUNCTION func
END INTERFACE
END SUBROUTINE miser
END INTERFACE

END INTERFACE
REAL(SP), DIMENSION(:,), INTENT(IN) :: regn
INTEGER(14B), INTENT(IN) :: ndim, npts
REAL(SP), INTENT(IN) :: dith
REAL(SP), INTENT(OUT) :: ave, var
END SUBROUTINE miser
END INTERFACE
INTERFACE
SUBROUTINE mmid(y, dydx, xs, htot, nstep, yout, derivs)
USE nrtype
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), INTENT(IN) :: xs, htot
REAL(SP), DIMENSION(:,), INTENT(IN) :: y, dydx
REAL(SP), DIMENSION(:,), INTENT(OUT) :: yout
INTERFACE
SUBROUTINE derivs(x, y, dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE mmid
END INTERFACE
INTERFACE
SUBROUTINE mnbrak(ax, bx, cx, fa, fb, fc, func)
USE nrtype
REAL(SP), INTENT(INOUT) :: ax, bx
REAL(SP), INTENT(OUT) :: cx, fa, fb, fc
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
END SUBROUTINE mnbrak
END INTERFACE
INTERFACE
SUBROUTINE mnewt(ntrial, x, tolx, tolf, usrfun)
USE nrtype
INTEGER(I4B), INTENT(IN) :: ntrial
REAL(SP), INTENT(IN) :: tolx, tolf
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: x
INTERFACE
SUBROUTINE usrfun(x, fvec, fjac)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(OUT) :: fvec
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: fjac
END SUBROUTINE usrfun
END INTERFACE
END SUBROUTINE mnewt
END INTERFACE
INTERFACE
SUBROUTINE moment(data, ave, adev, sdev, var, skew, curt)
USE nrtype
REAL(SP), INTENT(OUT) :: ave, adev, sdev, var, skew, curt
REAL(SP), DIMENSION(:,), INTENT(IN) :: data
END SUBROUTINE moment
END INTERFACE
INTERFACE
SUBROUTINE mp2dfr(a, s, n, m)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), INTENT(OUT) :: m
CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: a
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: s
END SUBROUTINE mp2dfr
END INTERFACE

INTERFACE
SUBROUTINE mpdiv(q,r,u,v,n,m)
USE nrtype
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: q,r
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
INTEGER(I4B), INTENT(IN) :: n,m
END SUBROUTINE mpdiv
END INTERFACE

INTERFACE
SUBROUTINE mpdiv(u,v,n,m)
USE nrtype
CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: u,v
INTEGER(I4B), INTENT(IN) :: n,m
END SUBROUTINE mpdiv
END INTERFACE

INTERFACE
SUBROUTINE mpmul(w,u,v,n,m)
USE nrtype
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
INTEGER(I4B), INTENT(IN) :: n,m
END SUBROUTINE mpmul
END INTERFACE

INTERFACE
SUBROUTINE mppi(n)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
END SUBROUTINE mppi
END INTERFACE

INTERFACE
SUBROUTINE mprove(a,alud,indx,b,x)
USE nrtype
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a,alud
REAL(SP), DIMENSION(:,:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(IN) :: b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
END SUBROUTINE mprove
END INTERFACE

INTERFACE
SUBROUTINE mpsqrt(w,u,v,n,m)
USE nrtype
CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: w,u
CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
INTEGER(I4B), INTENT(IN) :: n,m
END SUBROUTINE mpsqrt
END INTERFACE

INTERFACE
SUBROUTINE mrqcof(x,y,sig,a,maska,alpha,beta,chisq,funcs)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,a,sig
REAL(SP), DIMENSION(:), INTENT(OUT) :: beta
REAL(SP), DIMENSION(:), INTENT(INOUT) :: alpha
REAL(SP), INTENT(OUT) :: chisq
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
END SUBROUTINE mrqcof
END INTERFACE

SUBROUTINE funcs(x,a,yfit,dyda)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x, a
REAL(SP), DIMENSION(:), INTENT(OUT) :: yfit
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: dyda
END SUBROUTINE funcs
END INTERFACE
END SUBROUTINE mrqcof
END INTERFACE

INTERFACE
SUBROUTINE mrqmin(x,y,sig,a,maska,covar,alpha,chisq,funcs,alamda)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x, y, sig
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: covar, alpha
REAL(SP), INTENT(OUT) :: chisq
REAL(SP), INTENT(INOUT) :: alamda
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
INTERFACE
SUBROUTINE funcs(x,a,yfit,dyda)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x, a
REAL(SP), DIMENSION(:), INTENT(OUT) :: yfit
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: dyda
END SUBROUTINE funcs
END INTERFACE
END SUBROUTINE mrqmin
END INTERFACE
INTERFACE
SUBROUTINE newt(x,check)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
LOGICAL(LGT), INTENT(OUT) :: check
END SUBROUTINE newt
END INTERFACE

INTERFACE
SUBROUTINE odeint(ystart,x1,x2,eps,h1,hmin,derivs,rkqs)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: ystart
REAL(SP), INTENT(IN) :: x1,x2,eps,h1,hmin
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:,), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x, htry, eps
REAL(SP), INTENT(IN) :: hdid, hnext
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:,), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x, htry, eps
REAL(SP), INTENT(IN) :: hdid, hnext
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
SUBROUTINE rkqs(y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE rkqs
END INTERFACE
END SUBROUTINE odeint
END INTERFACE
INTERFACE
SUBROUTINE orthog(anu,alpha,beta,a,b)
USE nrtype
REAL(SP), DIMENSION(,:), INTENT(IN) :: anu,alpha,beta
REAL(SP), DIMENSION(,:), INTENT(OUT) :: a,b
END SUBROUTINE orthog
END INTERFACE
INTERFACE
SUBROUTINE pade(cof,resid)
USE nrtype
REAL(DP), DIMENSION(:,), INTENT(INOUT) :: cof
REAL(SP), INTENT(OUT) :: resid
END SUBROUTINE pade
END INTERFACE
INTERFACE
SUBROUTINE pccheb(d)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: d
REAL(SP), DIMENSION(size(d)) :: pccheb
END FUNCTION pccheb
END INTERFACE
INTERFACE
SUBROUTINE pcshft(a,b,d)
USE nrtype
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
END SUBROUTINE pcshft
END INTERFACE
INTERFACE
SUBROUTINE pearsn(x,y,r,prob,z)
USE nrtype
REAL(SP), INTENT(OUT) :: r,prob,z
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
END SUBROUTINE pearsn
END INTERFACE
INTERFACE
SUBROUTINE period(x,y,ofac,hifac,px,py,jmax,prob)
USE nrtype
INTEGER(I4B), INTENT(OUT) :: jmax
REAL(SP), INTENT(IN) :: ofac,hifac
REAL(SP), INTENT(OUT) :: prob
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), DIMENSION(:,), INTENT(IN) :: px,py
END SUBROUTINE period
END INTERFACE
INTERFACE plgndr
FUNCTION plgndr_s(l,m,x)
USE nrtype
INTEGER(I4B), INTENT(IN) :: l,m
REAL(SP) :: plgndr_s
END FUNCTION plgndr_s
FUNCTION plgndr_v(l,m,x)
USE nrtype
INTEGER(I4B), INTENT(IN) :: l,m
REAL(SP), DIMENSION(size(x)) :: plgndr_v
END FUNCTION plgndr_v
END INTERFACE
INTERFACE
FUNCTION poidev(xm)
USE nrtype
END INTERFACE
REAL(SP), INTENT(IN) :: xm
REAL(SP) :: poidev
END FUNCTION poidev

END INTERFACE

INTERFACE
FUNCTION polcoe(x,y)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y
REAL(SP), DIMENSION(size(x)) :: polcoe
END FUNCTION polcoe

END INTERFACE

INTERFACE
FUNCTION polcof(xa,ya)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: xa,ya
REAL(SP), DIMENSION(size(xa)) :: polcof
END FUNCTION polcof

END INTERFACE

INTERFACE
SUBROUTINE poldiv(u,v,q,r)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: u,v
REAL(SP), DIMENSION(:,), INTENT(OUT) :: q,r
END SUBROUTINE poldiv

END INTERFACE

INTERFACE
SUBROUTINE polin2(xa,x2a,ya,x1,x2,y,dy)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: xa,x2a
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: ya
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), INTENT(OUT) :: y,dy
END SUBROUTINE polin2

END INTERFACE

INTERFACE
SUBROUTINE polint(xa,ya,x,y,dy)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: xa,ya
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: y,dy
END SUBROUTINE polint

END INTERFACE

INTERFACE
SUBROUTINE powell(p,xi,ftol,iter,fret)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: p
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: xi
INTEGER(I4B), INTENT(INOUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), INTENT(OUT) :: iter,fret
END SUBROUTINE powell

END INTERFACE

INTERFACE
FUNCTION predic(data,d,nfut)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: data,d
INTEGER(I4B), INTENT(IN) :: nfut
REAL(SP), DIMENSION(nfut) :: predic
END FUNCTION predic

END INTERFACE

INTERFACE
FUNCTION probks(alam)
USE nrtype
REAL(SP), INTENT(IN) :: alam
REAL(SP) :: probks
END FUNCTION probks
END INTERFACE
INTERFACE pdes
SUBROUTINE pdes_s(lword,rword)
USE nrtyp
INTEGER(I4B), INTENT(INOUT) :: lword,rword
END SUBROUTINE pdes_s
END INTERFACE
INTERFACE
SUBROUTINE pdes_v(lword,rword)
USE nrtyp
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: lword,rword
END SUBROUTINE pdes_v
END INTERFACE
INTERFACE
SUBROUTINE pwt(a,isign)
USE nrtyp
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE pwt
END INTERFACE
INTERFACE
SUBROUTINE pwtset(n)
USE nrtyp
INTEGER(I4B), INTENT(IN) :: n
END SUBROUTINE pwtset
END INTERFACE
INTERFACE
FUNCTION pythag_dp(a,b)
USE nrtyp
REAL(DP), INTENT(IN) :: a,b
REAL(DP) :: pythag_dp
END FUNCTION pythag_dp
FUNCTION pythag_sp(a,b)
USE nrtyp
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: pythag_sp
END FUNCTION pythag_sp
END INTERFACE
INTERFACE
SUBROUTINE pzeextr(iest,xest,yest,yz,dy)
USE nrtyp
INTEGER(I4B), INTENT(IN) :: iest
REAL(SP), INTENT(IN) :: xest
REAL(SP), DIMENSION(:), INTENT(IN) :: yest
REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
END SUBROUTINE pzeextr
END INTERFACE
INTERFACE
SUBROUTINE qrdcmp(a,c,d,sing)
USE nrtyp
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: c,d
LOGICAL(LGT), INTENT(OUT) :: sing
END SUBROUTINE qrdcmp
END INTERFACE
INTERFACE
FUNCTION qromb(func,a,b)
USE nrtyp
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qromb
END INTERFACE
FUNCTION func(x)
USE nrtyp
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
END FUNCTION qromb
END INTERFACE
FUNCTION qromo(func,a,b,choose)
USE nrtype
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qromo
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
INTERFACE
SUBROUTINE choose(funk,aa,bb,s,n)
USE nrtype
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
FUNCTION funk(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: funk
END FUNCTION funk
END INTERFACE
END SUBROUTINE choose
END INTERFACE
END FUNCTION qromo
END INTERFACE
INTERFACE
SUBROUTINE qroot(p,b,c,eps)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: p
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: eps
END SUBROUTINE qroot
END INTERFACE
INTERFACE
SUBROUTINE qrsolv(a,c,d,b)
USE nrtype
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: c,d
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
END SUBROUTINE qrsolv
END INTERFACE
INTERFACE
SUBROUTINE qrupdt(r,qt,u,v)
USE nrtype
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: r,qt
REAL(SP), DIMENSION(:), INTENT(INOUT) :: u
REAL(SP), DIMENSION(:), INTENT(IN) :: v
END SUBROUTINE qrupdt
END INTERFACE
INTERFACE
FUNCTION qsimp(func,a,b)
USE nrtype
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qsimp

INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
END FUNCTION qsimp
END INTERFACE
INTERFACE
FUNCTION trap(func,a,b)
USE nrtype
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: trap
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
END FUNCTION qtrap
END INTERFACE
INTERFACE
SUBROUTINE quadct(x,y,xx,yy,fa,fb,fc,fd)
USE nrtype
REAL(SP), INTENT(IN) :: x,y
REAL(SP), DIMENSION(:,), INTENT(IN) :: xx,yy
REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
END SUBROUTINE quadct
END INTERFACE
INTERFACE
SUBROUTINE quadmx(a)
USE nrtype
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: a
END SUBROUTINE quadmx
END INTERFACE
INTERFACE
SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
USE nrtype
REAL(SP), INTENT(IN) :: x,y
REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
END SUBROUTINE quadvl
END INTERFACE
INTERFACE
FUNCTION ran(idum)
INTEGER(selected_int_kind(9)), INTENT(INOUT) :: idum
REAL :: ran
END FUNCTION ran
END INTERFACE
INTERFACE ran0
SUBROUTINE ran0_s(harvest)
USE nrtype
REAL(SP), INTENT(OUT) :: harvest
END SUBROUTINE ran0_s
SUBROUTINE ran0_v(harvest)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
END SUBROUTINE ran0_v
END INTERFACE
INTERFACE ran1
SUBROUTINE ran1_s(harvest)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
END SUBROUTINE ran1_s
SUBROUTINE ran1_v(harvest)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
END SUBROUTINE ran1_v
END INTERFACE
INTERFACE
FUNCTION ran(idum)
INTEGER(selected_int_kind(9)), INTENT(INOUT) :: idum
REAL :: ran
END FUNCTION ran
END INTERFACE
REAL(SP), INTENT(OUT) :: harvest
END SUBROUTINE ran1_s
SUBROUTINE ran1_v(harvest)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(OUT) :: harvest
END SUBROUTINE ran1_v
END INTERFACE

INTERFACE ran2
SUBROUTINE ran2_s(harvest)
USE nrtype
REAL(SP), INTENT(OUT) :: harvest
END SUBROUTINE ran2_s
SUBROUTINE ran2_v(harvest)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(OUT) :: harvest
END SUBROUTINE ran2_v
END INTERFACE

INTERFACE ran3
SUBROUTINE ran3_s(harvest)
USE nrtype
REAL(SP), INTENT(OUT) :: harvest
END SUBROUTINE ran3_s
SUBROUTINE ran3_v(harvest)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(OUT) :: harvest
END SUBROUTINE ran3_v
END INTERFACE

INTERFACE
SUBROUTINE ratint(xa,ya,x,y,dy)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: xa,ya
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: y,dy
END SUBROUTINE ratint
END INTERFACE

INTERFACE
SUBROUTINE ratlsq(func,a,b,mm,kk,cof,dev)
USE nrtype
REAL(DP), INTENT(IN) :: a,b
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(:,), INTENT(OUT) :: cof
REAL(DP), INTENT(OUT) :: dev
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(DP), DIMENSION(:,), INTENT(IN) :: x
REAL(DP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
END SUBROUTINE ratlsq
END INTERFACE

INTERFACE ratval
FUNCTION ratval_s(x,cof,mm,kk)
USE nrtype
REAL(DP), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
REAL(DP) :: ratval_s
END FUNCTION ratval_s
END INTERFACE
FUNCTION ratval_v(x,cof,mm,kk)
USE nrtype
REAL(DP), DIMENSION(:,), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
REAL(DP), DIMENSION(size(x)) :: ratval_v
END FUNCTION ratval_v
END INTERFACE

INTERFACE rc
FUNCTION rc_s(x,y)
USE nrtype
REAL(SP), INTENT(IN) :: x,y
REAL(SP) :: rc_s
END FUNCTION rc_s
END INTERFACE

INTERFACE rc
FUNCTION rc_v(x,y)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), DIMENSION(size(x)) :: rc_v
END FUNCTION rc_v
END INTERFACE

INTERFACE rd
FUNCTION rd_s(x,y,z)
USE nrtype
REAL(SP), INTENT(IN) :: x,y,z
REAL(SP) :: rd_s
END FUNCTION rd_s
END INTERFACE

INTERFACE rd
FUNCTION rd_v(x,y,z)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z
REAL(SP), DIMENSION(size(x)) :: rd_v
END FUNCTION rd_v
END INTERFACE

INTERFACE realft
SUBROUTINE realft_dp(data,isign,zdata)
USE nrtype
REAL(DP), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(DPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
END SUBROUTINE realft_dp
SUBROUTINE realft_sp(data,isign,zdata)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(SPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
END SUBROUTINE realft_sp
END INTERFACE

INTERFACE recur1
RECURSIVE FUNCTION recur1(a,b) RESULT(u)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a)) :: u
END FUNCTION recur1
END INTERFACE

INTERFACE recur2
FUNCTION recur2(a,b,c)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c
REAL(SP), DIMENSION(size(a)) :: recur2
END FUNCTION recur2
END INTERFACE

INTERFACE relax
SUBROUTINE relax(u,rhs)
USE nrtype
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
END SUBROUTINE relax
Appendix C2. Alphabetical Listing of Explicit Interfaces

END INTERFACE
INTERFACE 
SUBROUTINE relax2(u,rhs)
USE nrtype
REAL(DP), DIMENSION(:,::), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,::), INTENT(IN) :: rhs
END SUBROUTINE relax2
END INTERFACE

INTERFACE 
FUNCTION resid(u,rhs)
USE nrtype
REAL(DP), DIMENSION(:,::), INTENT(IN) :: u,rhs
REAL(DP), DIMENSION(size(u,1),size(u,1)) :: resid
END FUNCTION resid
END INTERFACE

INTERFACE rf
FUNCTION rf_s(x,y,z)
USE nrtype
REAL(SP), INTENT(IN) :: x,y,z
REAL(SP) :: rf_s
END FUNCTION rf_s
FUNCTION rf_v(x,y,z)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y,z
REAL(SP), DIMENSION(size(x)) :: rf_v
END FUNCTION rf_v
END INTERFACE

INTERFACE rj
FUNCTION rj_s(x,y,z,p)
USE nrtype
REAL(SP), INTENT(IN) :: x,y,z,p
REAL(SP) :: rj_s
END FUNCTION rj_s
FUNCTION rj_v(x,y,z,p)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x,y,z,p
REAL(SP), DIMENSION(size(x)) :: rj_v
END FUNCTION rj_v
END INTERFACE

INTERFACE 
SUBROUTINE rk4(y,dydx,x,h,yout,derivs)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: y,dydx
REAL(SP), INTENT(IN) :: x,h
REAL(SP), DIMENSION(:,), INTENT(OUT) :: yout
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE rk4
END INTERFACE

INTERFACE 
SUBROUTINE rkck(y,dydx,x,h,yout,yerr,derivs)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: y,dydx
REAL(SP), INTENT(IN) :: x,h
REAL(SP), DIMENSION(:,), INTENT(OUT) :: yout, yerr
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE rkck
END INTERFACE

INTERFACE
SUBROUTINE rk4(y,dydx,x,h,yout,derivs)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: y,dydx
REAL(SP), INTENT(IN) :: x,h
REAL(SP), DIMENSION(:,), INTENT(OUT) :: yout, yerr
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE rkck
END INTERFACE
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE rkck
END INTERFACE

INTERFACE
SUBROUTINE rkdumb(vstart,x1,x2,nstep,derivs)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: vstart
REAL(SP), INTENT(IN) :: x1,x2
INTEGER(I4B), INTENT(IN) :: nstep
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE rkdumb
END INTERFACE

INTERFACE
SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:,), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE rkqs
END INTERFACE

INTERFACE
SUBROUTINE rlft2(data,spec,speq,isign)
USE nrtype
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: data
COMPLEX(SPC), DIMENSION(:,,:), INTENT(OUT) :: spec
COMPLEX(SPC), DIMENSION(:,), INTENT(INOUT) :: speq
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE rlft2
END INTERFACE

INTERFACE
SUBROUTINE rlft3(data,spec,speq,isign)
USE nrtype
REAL(SP), DIMENSION(:,,:,:), INTENT(INOUT) :: data
COMPLEX(SPC), DIMENSION(:,,:,:), INTENT(OUT) :: spec
COMPLEX(SPC), DIMENSION(:,,:), INTENT(INOUT) :: speq
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE rlft3
END INTERFACE

INTERFACE
SUBROUTINE rotate(r,qt,i,a,b)
USE nrtype
REAL(SP), DIMENSION(:,,:), TARGET, INTENT(INOUT) :: r,qt
INTEGER(I4B), INTENT(IN) :: i
REAL(SP), INTENT(IN) :: a,b
END SUBROUTINE rotate
END INTERFACE

INTERFACE
SUBROUTINE rresolv(a,d,b)
USE nrttype
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:,), INTENT(IN) :: d
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: b
END SUBROUTINE rresolv
END INTERFACE

INTERFACE
FUNCTION rrsrct(uf)
USE nrttype
REAL(DP), DIMENSION(:,,:), INTENT(IN) :: uf
REAL(DP), DIMENSION((size(uf,1)+1)/2,(size(uf,1)+1)/2) :: rrsrct
END FUNCTION rrsrct
END INTERFACE

INTERFACE
FUNCTION rtbism(func,x1,x2,xacc)
USE nrttype
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtbism
INTERFACE
FUNCTION func(x)
USE nrttype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
END FUNCTION rtbism
END INTERFACE

INTERFACE
FUNCTION rtflsps(func,x1,x2,xacc)
USE nrttype
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtflsps
INTERFACE
FUNCTION func(x)
USE nrttype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
END FUNCTION rtflsps
END INTERFACE

INTERFACE
FUNCTION rtnewt(funcd,x1,x2,xacc)
USE nrttype
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtnewt
INTERFACE
SUBROUTINE funcd(x,fval,fderiv)
USE nrttype
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: fval,fderiv
END SUBROUTINE funcd
END INTERFACE
END FUNCTION rtnewt
END INTERFACE

INTERFACE
FUNCTION rtsafe(funcd,x1,x2,xacc)
USE nrttype
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtsafe
INTERFACE
SUBROUTINE funcd(x,fval,fderiv)
USE nrttype
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: fval,fderiv
END SUBROUTINE funcd
END INTERFACE
END FUNCTION rtsafe
END INTERFACE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtsafe

INTERFACE
  SUBROUTINE funcd(x,fval,fderiv)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: fval,fderiv
  END SUBROUTINE funcd
END INTERFACE

END FUNCTION rtsafe

END INTERFACE

INTERFACE
  FUNCTION rtsec(func,x1,x2,xacc)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2,xacc
    REAL(SP) :: rtsec
    INTERFACE
      FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
      END FUNCTION func
    END INTERFACE
    END FUNCTION rtsec
  END FUNCTION rtsec
END INTERFACE

INTERFACE
  SUBROUTINE rzextr(iest,xest,yest,yz,dy)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: iest
    REAL(SP), INTENT(IN) :: xest
    REAL(SP), DIMENSION(:), INTENT(IN) :: yest
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
  END SUBROUTINE rzextr
END INTERFACE

INTERFACE
  FUNCTION savgol(nl,nrr,ld,m)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
    REAL(SP), DIMENSION(nl+nrr+1) :: savgol
  END FUNCTION savgol
END INTERFACE

INTERFACE
  SUBROUTINE scrsho(func)
    USE nrtype
    INTERFACE
      FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
      END FUNCTION func
    END INTERFACE
    END SUBROUTINE scrsho
END INTERFACE

INTERFACE
  FUNCTION select(k,arr)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: k
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    REAL(SP) :: select
  END FUNCTION select
END INTERFACE

INTERFACE
  FUNCTION select_bypack(k,arr)
    USE nrtype
    INTERFACE
      FUNCTION select(k,arr)
        USE nrtype
        INTEGER(I4B), INTENT(IN) :: k
        REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
        REAL(SP) :: select
      END FUNCTION select
    END INTERFACE
    END FUNCTION select_bypack
END INTERFACE
INTERFACE
  SUBROUTINE select_heap(arr, heap)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: arr
  REAL(SP), DIMENSION(:), INTENT(OUT) :: heap
END SUBROUTINE select_heap

END INTERFACE

INTERFACE
  SUBROUTINE select_inplace(k, arr)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: k
  REAL(SP), DIMENSION(:), INTENT(IN) :: arr
  REAL(SP) :: select_inplace
END FUNCTION select_inplace

END INTERFACE

INTERFACE
  SUBROUTINE simplx(a, m1, m2, m3, icase, izrov, iposv)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
  INTEGER(I4B), INTENT(IN) :: m1, m2, m3
  INTEGER(I4B), INTENT(OUT) :: icase
  INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: izrov, iposv
END SUBROUTINE simplx

END INTERFACE

INTERFACE
  SUBROUTINE simpr(y, dydx, dfdx, dfdy, xs, htot, nstep, yout, derivs)
  USE nrtype
  REAL(SP), INTENT(IN) :: xs, htot
  REAL(SP), DIMENSION(:), INTENT(IN) :: y, dydx, dfdx
  REAL(SP), DIMENSION(:,,:), INTENT(IN) :: dfdy
  INTEGER(I4B), INTENT(IN) :: nstep
  REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
END SUBROUTINE derivs

USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:,), INTENT(IN) :: y
REAL(SP), DIMENSION(:,), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE

END INTERFACE

INTERFACE
  SUBROUTINE sinft(y)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
END SUBROUTINE sinft

END INTERFACE

INTERFACE
  SUBROUTINE slvsm2(u, rhs)
  USE nrtype
  REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
  REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
END SUBROUTINE slvsm2

END INTERFACE

INTERFACE
  SUBROUTINE slvsm1(u, rhs)
  USE nrtype
  REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
END SUBROUTINE slvsm1
REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
END SUBROUTINE slvsml

END INTERFACE

INTERFACE
SUBROUTINE sncndn(uu,emmc,sn,cn,dn)
USE nrtype
REAL(SP), INTENT(IN) :: uu,emmc
REAL(SP), INTENT(OUT) :: sn,cn,dn
END SUBROUTINE sncndn
END INTERFACE

INTERFACE
FUNCTION snrm(sx,itol)
USE nrtype
REAL(DP), DIMENSION(:), INTENT(IN) :: sx
INTEGER(I4B), INTENT(IN) :: itol
REAL(DP) :: snrm
END FUNCTION snrm
END INTERFACE

INTERFACE
SUBROUTINE sobseq(x,init)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(OUT) :: x
INTEGER(I4B), OPTIONAL, INTENT(IN) :: init
END SUBROUTINE sobseq
END INTERFACE

INTERFACE
SUBROUTINE solvde(itmax,conv,slowc,scalv,indexv,nb,y)
USE nrtype
INTEGER(I4B), INTENT(IN) :: itmax,nb
REAL(SP), INTENT(IN) :: conv,slowc
REAL(SP), DIMENSION(:), INTENT(IN) :: scalv
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indexv
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: y
END SUBROUTINE solvde
END INTERFACE

INTERFACE
SUBROUTINE sor(a,b,c,d,e,f,u,rjac)
USE nrtype
REAL(DP), DIMENSION(:,:), INTENT(IN) :: a,b,c,d,e,f
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), INTENT(IN) :: rjac
END SUBROUTINE sor
END INTERFACE

INTERFACE
SUBROUTINE sort(arr)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
END SUBROUTINE sort
END INTERFACE

INTERFACE
SUBROUTINE sort2(arr,slave)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave
END SUBROUTINE sort2
END INTERFACE

INTERFACE
SUBROUTINE sort3(arr,slave1,slave2)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave1,slave2
END SUBROUTINE sort3
END INTERFACE

INTERFACE
SUBROUTINE sort_bypack(arr)
USE nrtype

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE Art of PARALLEL Scientific Computing (ISBN 0-521-57439-0) Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. Permission is granted for internal use bystitution users to make one paper copy for their personal use: Further reproduction, in any form, by any means, for any purpose, except as permitted under US copyright law, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: arr
END SUBROUTINE sort_bypack

END INTERFACE

INTERFACE
SUBROUTINE sort_byreshape(arr)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: arr
END SUBROUTINE sort_byreshape

END INTERFACE

INTERFACE
SUBROUTINE sort_heap(arr)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: arr
END SUBROUTINE sort_heap

END INTERFACE

INTERFACE
SUBROUTINE sort_pick(arr)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: arr
END SUBROUTINE sort_pick

END INTERFACE

INTERFACE
SUBROUTINE sort_radix(arr)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: arr
END SUBROUTINE sort_radix

END INTERFACE

INTERFACE
SUBROUTINE sort_shell(arr)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(INOUT) :: arr
END SUBROUTINE sort_shell

END INTERFACE

INTERFACE
SUBROUTINE spctrm(p,k,ovrlap,unit,n_window)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(OUT) :: p
INTEGER(I4B), INTENT(IN) :: k
LOGICAL(LGT), INTENT(IN) :: ovrlap
INTEGER(I4B), OPTIONAL, INTENT(IN) :: n_window,unit
END SUBROUTINE spctrm

END INTERFACE

INTERFACE
SUBROUTINE spear(data1,data2,d,zd,probd,rs,probrs)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: d,zd,probd,rs,probrs
END SUBROUTINE spear

END INTERFACE

INTERFACE
SUBROUTINE sphbes_s(n,x,sj,sy,sjp,syp)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: sj,sy,sjp,syp
END SUBROUTINE sphbes_s

SUBROUTINE sphbes_v(n,x,sj,sy,sjp,syp)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: sj,sy,sjp,syp
END SUBROUTINE sphbes_v

END INTERFACE
INTERFACE
SUBROUTINE splie2(x1a,x2a,ya,y2a)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x1a,x2a
REAL(SP), DIMENSION(:,), INTENT(IN) :: ya
REAL(SP), DIMENSION(:,), INTENT(OUT) :: y2a
END SUBROUTINE splie2
END INTERFACE

INTERFACE
FUNCTION splin2(x1a,x2a,ya,y2a,x1,x2)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: x1a,x2a
REAL(SP), DIMENSION(:,), INTENT(IN) :: ya,y2a
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP) :: splin2
END FUNCTION splin2
END INTERFACE

INTERFACE
SUBROUTINE spline(x,y,yp1,ypn,y2)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(IN) :: yp1,ypn
REAL(SP), DIMENSION(:), INTENT(OUT) :: y2
END SUBROUTINE spline
END INTERFACE

INTERFACE
FUNCTION splint(xa,ya,y2a,x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya,y2a
REAL(SP), INTENT(IN) :: x
REAL(SP) :: splint
END FUNCTION splint
END INTERFACE

INTERFACE sprsax
SUBROUTINE sprsax_dp(sa,x,b)
USE nrtype
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION (:), INTENT(IN) :: x
REAL(DP), DIMENSION (:), INTENT(OUT) :: b
END SUBROUTINE sprsax_dp
SUBROUTINE sprsax_sp(sa,x,b)
USE nrtype
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION (:), INTENT(IN) :: x
REAL(SP), DIMENSION (:), INTENT(OUT) :: b
END SUBROUTINE sprsax_sp
END INTERFACE

INTERFACE sprsdiag
SUBROUTINE sprsdiag_dp(sa,b)
USE nrtype
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION (:), INTENT(OUT) :: b
END SUBROUTINE sprsdiag_dp
SUBROUTINE sprsdiag_sp(sa,b)
USE nrtype
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION (:), INTENT(OUT) :: b
END SUBROUTINE sprsdiag_sp
END INTERFACE

INTERFACE sprsinv
SUBROUTINE sprsinv_sp(a,thresh,sa)
USE nrtype
REAL(SP), DIMENSION (:,:), INTENT(IN) :: a
REAL(SP), INTENT(IN) :: thresh
TYPE(sprs2_sp), INTENT(OUT) :: sa
END SUBROUTINE sprsin_sp

SUBROUTINE sprsin_dp(a,thresh,sa)
USE nrtype
REAL(DP), DIMENSION(:,), INTENT(IN) :: a
REAL(DP), INTENT(IN) :: thresh
TYPE(sprs2_dp), INTENT(OUT) :: sa
END SUBROUTINE sprsin_dp

END INTERFACE

INTERFACE
SUBROUTINE sprstp(sa)
USE nrtype
TYPE(sprs2_sp), INTENT(INOUT) :: sa
END SUBROUTINE sprstp

END INTERFACE

INTERFACE sprstx

SUBROUTINE sprstx_dp(sa,x,b)
USE nrtype
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION (:), INTENT(IN) :: x
REAL(DP), DIMENSION (:), INTENT(OUT) :: b
END SUBROUTINE sprstx_dp

SUBROUTINE sprstx_sp(sa,x,b)
USE nrtype
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION (:), INTENT(IN) :: x
REAL(SP), DIMENSION (:), INTENT(OUT) :: b
END SUBROUTINE sprstx_sp

END INTERFACE

INTERFACE
SUBROUTINE stifbs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype
REAL(SP), DIMENSION (:), INTENT(INOUT) :: y
REAL(SP), DIMENSION (:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(OUT) :: hdid,hnext
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION (:), INTENT(IN) :: y
REAL(SP), DIMENSION (:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE

END SUBROUTINE stifbs

END INTERFACE

INTERFACE
SUBROUTINE stiff(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype
REAL(SP), DIMENSION (:), INTENT(INOUT) :: y
REAL(SP), DIMENSION (:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION (:), INTENT(IN) :: y
REAL(SP), DIMENSION (:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE

END SUBROUTINE stiff
Appendix C2.  Alphabetical Listing of Explicit Interfaces

END INTERFACE
END SUBROUTINE stiff
END INTERFACE

INTERFACE
SUBROUTINE stoerm(y,d2y,xs,htot,nstep,yout,derivs)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: y,d2y
REAL(SP), INTENT(IN) :: xs,htot
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), DIMENSION(:,), INTENT(OUT) :: yout
END INTERFACE
SUBROUTINE derivs(x,y,dydx)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
END SUBROUTINE derivs
END INTERFACE
END SUBROUTINE stoerm
END INTERFACE

INTERFACE
SUBROUTINE svbkbsb_dp(u,w,v,b,x)
USE nrtype
REAL(DP), DIMENSION(:,,:), INTENT(IN) :: u,v
REAL(DP), DIMENSION(:), INTENT(IN) :: w,b
REAL(DP), DIMENSION(:), INTENT(OUT) :: x
END SUBROUTINE svbkbsb_dp
SUBROUTINE svbkbsb_sp(u,w,v,b,x)
USE nrtype
REAL(SP), DIMENSION(:,,:), INTENT(IN) :: u,v
REAL(SP), DIMENSION(:), INTENT(IN) :: w,b
REAL(SP), DIMENSION(:), INTENT(OUT) :: x
END SUBROUTINE svbkbsb_sp
END INTERFACE

INTERFACE
SUBROUTINE svdcmp_dp(a,w,v)
USE nrtype
REAL(DP), DIMENSION(:,,:), INTENT(INOUT) :: a
REAL(DP), DIMENSION(:), INTENT(OUT) :: w
REAL(DP), DIMENSION(:,,:), INTENT(OUT) :: v
END SUBROUTINE svdcmp_dp
SUBROUTINE svdcmp_sp(a,w,v)
USE nrtype
REAL(SP), DIMENSION(:,,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: w
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: v
END SUBROUTINE svdcmp_sp
END INTERFACE

INTERFACE
SUBROUTINE svdfit(x,y,sig,a,v,w,chisq,funcs)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
REAL(SP), DIMENSION(:), INTENT(OUT) :: a,v,w
REAL(SP), DIMENSION(:,), INTENT(OUT) :: chisq
INTERFACE
FUNCTION funcs(x,n)
USE nrtype
REAL(SP), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: funcs
END FUNCTION funcs
END INTERFACE
END SUBROUTINE svdfit
END INTERFACE
END SUBROUTINE svdfit
END INTERFACE

INTERFACE
SUBROUTINE svdvar(v,w,cvm)
USE nrtype
REAL(SP), DIMENSION(:,:), INTENT(IN) :: v
REAL(SP), DIMENSION(:), INTENT(IN) :: w
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: cvm
END SUBROUTINE svdvar
END INTERFACE

INTERFACE
FUNCTION toeplz(r,y)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: r,y
REAL(SP), DIMENSION(size(y)) :: toeplz
END FUNCTION toeplz
END INTERFACE

INTERFACE
SUBROUTINE tptest(data1,data2,t,prob)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: t,prob
END SUBROUTINE tptest
END INTERFACE

INTERFACE
SUBROUTINE tqli(d,e,z)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d,e
REAL(SP), DIMENSION(:,:), OPTIONAL, INTENT(INOUT) :: z
END SUBROUTINE tqli
END INTERFACE

INTERFACE
SUBROUTINE trapzd(func,a,b,s,n)
USE nrtype
REAL(SP), INTENT(IN) :: a,b
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: func
END FUNCTION func
END INTERFACE
END SUBROUTINE trapzd
END INTERFACE

INTERFACE
SUBROUTINE tred2(a,d,e,novectors)
USE nrtype
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d,e
LOGICAL(LGT), OPTIONAL, INTENT(IN) :: novectors
END SUBROUTINE tred2
END INTERFACE

INTERFACE tridag
RECURSIVE SUBROUTINE tridag_par(a,b,c,r,u)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
REAL(SP), DIMENSION(:), INTENT(OUT) :: u
END SUBROUTINE tridag_par
END INTERFACE
INTERFACE
SUBROUTINE tridag_ser(a,b,c,r,u)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: a,b,c,r
REAL(SP), DIMENSION(:,), INTENT(OUT) :: u
END SUBROUTINE tridag_ser
END INTERFACE

INTERFACE
SUBROUTINE ttest(data1,data2,t,prob)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: t,prob
END SUBROUTINE ttest
END INTERFACE

INTERFACE
SUBROUTINE tutest(data1,data2,t,prob)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: t,prob
END SUBROUTINE tutest
END INTERFACE

INTERFACE
SUBROUTINE twofft(data1,data2,fft1,fft2)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: data1,data2
COMPLEX(SPC), DIMENSION(:,), INTENT(OUT) :: fft1,fft2
END SUBROUTINE twofft
END INTERFACE

INTERFACE
FUNCTION vander(x,q)
USE nrtype
REAL(DP), DIMENSION(:,), INTENT(IN) :: x,q
REAL(DP), DIMENSION(size(x)) :: vander
END FUNCTION vander
END INTERFACE

INTERFACE
SUBROUTINE vegas(region,func,init,ncall,itmx,nprn,tgral,sd,chi2a)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: region
INTEGER(I4B), INTENT(IN) :: init,ncall,itmx,nprn
REAL(SP), INTENT(OUT) :: tgral,sd,chi2a
INTERFACE
FUNCTION func(pt,wgt)
USE nrtype
REAL(SP), DIMENSION(:,), INTENT(IN) :: pt
REAL(SP), INTENT(IN) :: wgt
REAL(SP) :: func
END FUNCTION func
END INTERFACE
END SUBROUTINE vegas
END INTERFACE

INTERFACE
SUBROUTINE voltra(t0,h,t,f,g,ak)
USE nrtype
REAL(SP), INTENT(IN) :: t0,h
REAL(SP), DIMENSION(:,), INTENT(OUT) :: t
REAL(SP), DIMENSION(:,,:), INTENT(OUT) :: f
INTERFACE
FUNCTION g(t)
USE nrtype
REAL(SP), INTENT(IN) :: t
REAL(SP), DIMENSION(:,), POINTER :: g
END FUNCTION g
END INTERFACE
FUNCTION ak(t,s)
USE nrtype
REAL(SP), INTENT(IN) :: t, s
REAL(SP), DIMENSION(:,:), POINTER :: ak
END FUNCTION ak
END INTERFACE
END SUBROUTINE voltra
END INTERFACE

INTERFACE
SUBROUTINE wt1(a,isign,wtstep)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
INTERFACE
SUBROUTINE wtstep(a,isign)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE wtstep
END INTERFACE
END SUBROUTINE wt1
END INTERFACE

INTERFACE
SUBROUTINE wtn(a,nn,isign,wtstep)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nn
INTEGER(I4B), INTENT(IN) :: isign
INTERFACE
SUBROUTINE wtstep(a,isign)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
END SUBROUTINE wtstep
END INTERFACE
END SUBROUTINE wtn
END INTERFACE

INTERFACE
FUNCTION wwghts(n,h,kermom)
USE nrtype
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: h
REAL(SP), DIMENSION(n) :: wwghts
INTERFACE
FUNCTION kermom(y,m)
USE nrtype
REAL(DP), INTENT(IN) :: y
INTEGER(I4B), INTENT(IN) :: m
REAL(DP), DIMENSION(m) :: kermom
END FUNCTION kermom
END INTERFACE
END FUNCTION wwghts
END INTERFACE

INTERFACE
SUBROUTINE zbrac(func,x1,x2,succes)
USE nrtype
REAL(SP), INTENT(INOUT) :: x1,x2
LOGICAL(LGT), INTENT(INOUT) :: succes
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
END INTERFACE
END SUBROUTINE zbrac
END INTERFACE

INTERFACE
SUBROUTINE zbrak(func,x1,x2,n,xb1,xb2,nb)
USE nrtype
INTEGER(14B), INTENT(IN) :: n
INTEGER(14B), INTENT(OUT) :: nb
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), DIMENSION(:), POINTER :: xb1,xb2
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
END SUBROUTINE zbrak
END INTERFACE

INTERFACE
FUNCTION zbrent(func,x1,x2,tol)
USE nrtype
REAL(SP), INTENT(IN) :: x1,x2,tol
REAL(SP) :: zbrent
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
END FUNCTION zbrent
END INTERFACE

INTERFACE
SUBROUTINE zrhqr(a,rtr,rti)
USE nrtype
REAL(SP), DIMENSION(:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: rtr,rti
END SUBROUTINE zrhqr
END INTERFACE

INTERFACE
FUNCTION zriddr(func,x1,x2,xacc)
USE nrtype
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: zriddr
INTERFACE
FUNCTION func(x)
USE nrtype
REAL(SP), INTENT(IN) :: x
REAL(SP) :: func
END FUNCTION func
END INTERFACE
END FUNCTION zriddr
END INTERFACE

INTERFACE
SUBROUTINE zroots(a,roots,polish)
USE nrtype
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: roots
LOGICAL(LGT), INTENT(IN) :: polish
END SUBROUTINE zroots
END INTERFACE

END INTERFACE
END MODULE nr
## C3. Index of Programs and Dependencies

The following table lists, in alphabetical order, all the routines in Volume 2 of *Numerical Recipes*. When a routine requires subsidiary routines, either from this book or else user-supplied, the full dependency tree is shown: A routine calls directly all routines to which it is connected by a solid line in the column immediately to its right; it calls indirectly the connected routines in all columns to its right. Typographical conventions: Routines from this book are in typewriter font (e.g., `eulsum`, `gammln`). The smaller, slanted font is used for the second and subsequent occurrences of a routine in a single dependency tree. (When you are getting routines from the *Numerical Recipes* machine-readable media or hypertext archives, you need specify names only in the larger, upright font.) User-supplied routines are indicated by the use of text font and square brackets, e.g., `[funcv]`. Consult the text for individual specifications of these routines. The right-hand side of the table lists chapter and page numbers for each program.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Dependencies</th>
<th>Chapter and Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>airy</td>
<td>bessik</td>
<td>B6 (p. 1121)</td>
</tr>
<tr>
<td></td>
<td>bessjy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>beschb</td>
<td></td>
</tr>
<tr>
<td></td>
<td>chebev</td>
<td></td>
</tr>
<tr>
<td>amebsa</td>
<td>ran</td>
<td>B10 (p. 1222)</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td></td>
</tr>
<tr>
<td>amoeba</td>
<td></td>
<td>B10 (p. 1208)</td>
</tr>
<tr>
<td>anneal</td>
<td>ran</td>
<td>B10 (p. 1219)</td>
</tr>
<tr>
<td></td>
<td>ran_state</td>
<td></td>
</tr>
<tr>
<td>arcmak</td>
<td></td>
<td>B20 (p. 1349)</td>
</tr>
<tr>
<td>arcde</td>
<td>arcmak</td>
<td>B20 (p. 1350)</td>
</tr>
<tr>
<td>avevar</td>
<td></td>
<td>B14 (p. 1270)</td>
</tr>
<tr>
<td>badluk</td>
<td>julday</td>
<td>B1 (p. 1011)</td>
</tr>
<tr>
<td></td>
<td>flmoon</td>
<td></td>
</tr>
<tr>
<td>balanc</td>
<td></td>
<td>B11 (p. 1230)</td>
</tr>
<tr>
<td>banbks</td>
<td></td>
<td>B2 (p. 1021)</td>
</tr>
<tr>
<td>bandec</td>
<td></td>
<td>B2 (p. 1020)</td>
</tr>
<tr>
<td>banmul</td>
<td></td>
<td>B2 (p. 1019)</td>
</tr>
<tr>
<td>bcucof</td>
<td></td>
<td>B3 (p. 1049)</td>
</tr>
<tr>
<td>bcuint</td>
<td>bcucof</td>
<td>B3 (p. 1050)</td>
</tr>
<tr>
<td>beschb</td>
<td>chebev</td>
<td>B6 (p. 1118)</td>
</tr>
<tr>
<td>Function</td>
<td>Notes</td>
<td>Page(s)</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>bessi</td>
<td></td>
<td>B6 (p. 1114)</td>
</tr>
<tr>
<td>bessi0</td>
<td></td>
<td>B6 (p. 1109)</td>
</tr>
<tr>
<td>bessi1</td>
<td></td>
<td>B6 (p. 1111)</td>
</tr>
<tr>
<td>bessik</td>
<td>beschb chebev</td>
<td>B6 (p. 1118)</td>
</tr>
<tr>
<td>bessj</td>
<td>bessj0 bessj1</td>
<td>B6 (p. 1106)</td>
</tr>
<tr>
<td>bessj0</td>
<td></td>
<td>B6 (p. 1101)</td>
</tr>
<tr>
<td>bessj1</td>
<td></td>
<td>B6 (p. 1103)</td>
</tr>
<tr>
<td>bessjy</td>
<td>beschb chebev</td>
<td>B6 (p. 1115)</td>
</tr>
<tr>
<td>bessk</td>
<td>bessk0 bessi0 bessi1</td>
<td>B6 (p. 1113)</td>
</tr>
<tr>
<td>bessk0</td>
<td>bessi0 bessi1</td>
<td>B6 (p. 1110)</td>
</tr>
<tr>
<td>bessk1</td>
<td>bessi1</td>
<td>B6 (p. 1112)</td>
</tr>
<tr>
<td>bessy</td>
<td>bessy1 bessj1</td>
<td>B6 (p. 1105)</td>
</tr>
<tr>
<td>bessy0</td>
<td>bessj0</td>
<td>B6 (p. 1102)</td>
</tr>
<tr>
<td>bessy1</td>
<td>bessj1</td>
<td>B6 (p. 1104)</td>
</tr>
<tr>
<td>beta</td>
<td>gammln</td>
<td>B6 (p. 1089)</td>
</tr>
<tr>
<td>betaacf</td>
<td></td>
<td>B6 (p. 1099)</td>
</tr>
<tr>
<td>betai</td>
<td>gammln betaacf</td>
<td>B6 (p. 1098)</td>
</tr>
<tr>
<td>bico</td>
<td>factln gammln</td>
<td>B6 (p. 1087)</td>
</tr>
<tr>
<td>bnldev</td>
<td>ran1 ran_state gammln</td>
<td>B7 (p. 1155)</td>
</tr>
<tr>
<td>brent</td>
<td>[func]</td>
<td>B10 (p. 1204)</td>
</tr>
<tr>
<td>broydn</td>
<td>fmin</td>
<td>B9 (p. 1199)</td>
</tr>
<tr>
<td></td>
<td>fdjac [funcv] pythag</td>
<td></td>
</tr>
<tr>
<td></td>
<td>qrupdt rotate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>qrdcmp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rsolv</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lnsrch [funcv] [derivs]</td>
<td></td>
</tr>
<tr>
<td>bsstep</td>
<td>mmid</td>
<td>B16 (p. 1303)</td>
</tr>
<tr>
<td></td>
<td>pzextr</td>
<td></td>
</tr>
<tr>
<td>caldat</td>
<td></td>
<td>B1 (p. 1013)</td>
</tr>
<tr>
<td>chder</td>
<td></td>
<td>B5 (p. 1077)</td>
</tr>
<tr>
<td>chebev</td>
<td></td>
<td>B5 (p. 1076)</td>
</tr>
<tr>
<td>chebfnt</td>
<td>[func]</td>
<td>B5 (p. 1076)</td>
</tr>
<tr>
<td>chebpc</td>
<td></td>
<td>B5 (p. 1078)</td>
</tr>
<tr>
<td>chint</td>
<td></td>
<td>B5 (p. 1078)</td>
</tr>
<tr>
<td>chixy</td>
<td></td>
<td>B15 (p. 1287)</td>
</tr>
</tbody>
</table>
## Appendix C3. Index of Programs and Dependencies

<table>
<thead>
<tr>
<th>Program</th>
<th>Dependencies</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>choldc</td>
<td></td>
<td>B2   (p. 1038)</td>
</tr>
<tr>
<td>chols1</td>
<td></td>
<td>B2   (p. 1039)</td>
</tr>
<tr>
<td>chsone</td>
<td>gammq, gser, gcf, gammln</td>
<td>B14 (p. 1272)</td>
</tr>
<tr>
<td>chstwo</td>
<td>gammq, gser, gcf, gammln</td>
<td>B14 (p. 1272)</td>
</tr>
<tr>
<td>cisi</td>
<td></td>
<td>B6   (p. 1125)</td>
</tr>
<tr>
<td>cntab1</td>
<td>gammq, gser, gcf, gammln</td>
<td>B14 (p. 1275)</td>
</tr>
<tr>
<td>cntab2</td>
<td></td>
<td>B14 (p. 1275)</td>
</tr>
<tr>
<td>convlv</td>
<td>realft, four1, fourrow</td>
<td>B13 (p. 1253)</td>
</tr>
<tr>
<td>correl</td>
<td>realft, four1, fourrow</td>
<td>B13 (p. 1254)</td>
</tr>
<tr>
<td>cosft1</td>
<td>realft, four1, fourrow</td>
<td>B12 (p. 1245)</td>
</tr>
<tr>
<td>cosft2</td>
<td>realft, four1, fourrow</td>
<td>B12 (p. 1246)</td>
</tr>
<tr>
<td>covsrt</td>
<td></td>
<td>B15 (p. 1289)</td>
</tr>
<tr>
<td>cyclic</td>
<td>tridag</td>
<td>B2   (p. 1030)</td>
</tr>
<tr>
<td>daub4</td>
<td></td>
<td>B13 (p. 1264)</td>
</tr>
<tr>
<td>dawson</td>
<td></td>
<td>B6   (p. 1127)</td>
</tr>
<tr>
<td>dbrent</td>
<td>[func], [dfunc]</td>
<td>B10 (p. 1205)</td>
</tr>
<tr>
<td>ddply</td>
<td></td>
<td>B5   (p. 1071)</td>
</tr>
<tr>
<td>decchk</td>
<td></td>
<td>B20 (p. 1345)</td>
</tr>
<tr>
<td>dfpmin</td>
<td>[func], [dfunc], lnsrch, [func]</td>
<td>B10 (p. 1215)</td>
</tr>
<tr>
<td>dfridr</td>
<td>[func]</td>
<td>B5   (p. 1075)</td>
</tr>
<tr>
<td>dftcor</td>
<td></td>
<td>B13 (p. 1261)</td>
</tr>
<tr>
<td>dftint</td>
<td>[func], realft, four1, fourrow, polint, dftcor</td>
<td>B13 (p. 1263)</td>
</tr>
<tr>
<td>difeq</td>
<td></td>
<td>B17 (p. 1320)</td>
</tr>
<tr>
<td>dlinmin</td>
<td>mnbrak, dbrent, [func], [dfunc]</td>
<td>B10 (p. 1212)</td>
</tr>
<tr>
<td>eclass</td>
<td></td>
<td>B8   (p. 1180)</td>
</tr>
<tr>
<td>eclazz</td>
<td>[equiv]</td>
<td>B8   (p. 1180)</td>
</tr>
<tr>
<td>ei</td>
<td></td>
<td>B6   (p. 1097)</td>
</tr>
<tr>
<td>eigsrt</td>
<td></td>
<td>B11 (p. 1227)</td>
</tr>
<tr>
<td>elle</td>
<td>rf, rd</td>
<td>B6   (p. 1136)</td>
</tr>
</tbody>
</table>
### Appendix C3. Index of Programs and Dependencies

<table>
<thead>
<tr>
<th>Program</th>
<th>Dependencies</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>elf</td>
<td>rf</td>
<td>B6 (p. 1135)</td>
</tr>
<tr>
<td>ellpi</td>
<td>rj, rc</td>
<td>B6 (p. 1136)</td>
</tr>
<tr>
<td>elmhes</td>
<td></td>
<td>B11 (p. 1231)</td>
</tr>
<tr>
<td>erf</td>
<td>gamm, gser, gcf, gammln</td>
<td>B6 (p. 1094)</td>
</tr>
<tr>
<td>erfc</td>
<td>gamm, gser, gcf, gammln</td>
<td>B6 (p. 1094)</td>
</tr>
<tr>
<td>erfcc</td>
<td></td>
<td>B6 (p. 1095)</td>
</tr>
<tr>
<td>eulsum</td>
<td></td>
<td>B5 (p. 1070)</td>
</tr>
<tr>
<td>evlmem</td>
<td></td>
<td>B13 (p. 1258)</td>
</tr>
<tr>
<td>expdev</td>
<td>ran, ran_state</td>
<td>B7 (p. 1151)</td>
</tr>
<tr>
<td>expint</td>
<td></td>
<td>B6 (p. 1096)</td>
</tr>
<tr>
<td>factln</td>
<td>gammln</td>
<td>B6 (p. 1088)</td>
</tr>
<tr>
<td>factrl</td>
<td>gammln</td>
<td>B6 (p. 1086)</td>
</tr>
<tr>
<td>fasper</td>
<td>avevar, four1, fourrow, realft</td>
<td>B13 (p. 1259)</td>
</tr>
<tr>
<td>fdjac</td>
<td>[funcv]</td>
<td>B9 (p. 1197)</td>
</tr>
<tr>
<td>fgauss</td>
<td></td>
<td>B15 (p. 1294)</td>
</tr>
<tr>
<td>fit</td>
<td>gamm, gser, gcf, gammln</td>
<td>B15 (p. 1285)</td>
</tr>
<tr>
<td>fitexy</td>
<td>avevar, four1, fourrow, realft</td>
<td>B15 (p. 1286)</td>
</tr>
<tr>
<td>fixrts</td>
<td>zroots, laguer</td>
<td>B13 (p. 1257)</td>
</tr>
<tr>
<td>fleg</td>
<td></td>
<td>B15 (p. 1291)</td>
</tr>
<tr>
<td>flmoon</td>
<td></td>
<td>B1 (p. 1010)</td>
</tr>
<tr>
<td>fmin</td>
<td>[funcv]</td>
<td>B9 (p. 1198)</td>
</tr>
<tr>
<td>four1</td>
<td>fourrow</td>
<td>B12 (p. 1239)</td>
</tr>
<tr>
<td>four1_alt</td>
<td>fourcol</td>
<td>B12 (p. 1240)</td>
</tr>
<tr>
<td>four1_gather</td>
<td></td>
<td>B12 (p. 1250)</td>
</tr>
<tr>
<td>four2</td>
<td>fourrow</td>
<td>B12 (p. 1241)</td>
</tr>
<tr>
<td>Function</td>
<td>Location</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>four2_alt</td>
<td>fourcol</td>
<td>B12 (p. 1242)</td>
</tr>
<tr>
<td>four3</td>
<td>fourrow_3d</td>
<td>B12 (p. 1246)</td>
</tr>
<tr>
<td>four3_alt</td>
<td>fourcol_3d</td>
<td>B12 (p. 1247)</td>
</tr>
<tr>
<td>fourcol</td>
<td>fourcol_3d</td>
<td>B12 (p. 1237)</td>
</tr>
<tr>
<td>fourcol_3d</td>
<td>fourn_gather</td>
<td>B12 (p. 1238)</td>
</tr>
<tr>
<td>fourrow</td>
<td>fourrow_3d</td>
<td>B12 (p. 1251)</td>
</tr>
<tr>
<td>fourrow_3d</td>
<td>fpoly</td>
<td>B12 (p. 1235)</td>
</tr>
<tr>
<td>fred2</td>
<td>gauleg</td>
<td>B12 (p. 1236)</td>
</tr>
<tr>
<td>fredex</td>
<td>quadmx</td>
<td>B15 (p. 1237)</td>
</tr>
<tr>
<td>fredin</td>
<td>gauleg</td>
<td>B15 (p. 1238)</td>
</tr>
<tr>
<td>ftest</td>
<td>avevar</td>
<td>B12 (p. 1239)</td>
</tr>
<tr>
<td>frprmn</td>
<td>ftest</td>
<td>B12 (p. 1240)</td>
</tr>
<tr>
<td>gamdev</td>
<td>ran1</td>
<td>B12 (p. 1241)</td>
</tr>
<tr>
<td>gammln</td>
<td>ran1</td>
<td>B12 (p. 1242)</td>
</tr>
<tr>
<td>gammq</td>
<td>tqli</td>
<td>B12 (p. 1243)</td>
</tr>
<tr>
<td>gasdev</td>
<td>ran1</td>
<td>B12 (p. 1244)</td>
</tr>
<tr>
<td>gaucof</td>
<td>tqli</td>
<td>B12 (p. 1245)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaucof</td>
<td>B12 (p. 1246)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1247)</td>
</tr>
<tr>
<td>gaulag</td>
<td>gaujac</td>
<td>B12 (p. 1248)</td>
</tr>
<tr>
<td>gaulag</td>
<td>gaujac</td>
<td>B12 (p. 1249)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1250)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1251)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1252)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1253)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1254)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1255)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1256)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1257)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1258)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1259)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1260)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1261)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1262)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1263)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1264)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1265)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1266)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1267)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1268)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1269)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1270)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1271)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1272)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1273)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1274)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1275)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1276)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1277)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1278)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1279)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1280)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1281)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1282)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1283)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1284)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1285)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1286)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1287)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1288)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1289)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1290)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1291)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1292)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1293)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1294)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1295)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1296)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1297)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1298)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1299)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1300)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1301)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1302)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1303)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1304)</td>
</tr>
<tr>
<td>gaujac</td>
<td>gaujac</td>
<td>B12 (p. 1305)</td>
</tr>
<tr>
<td>Program/Dependency</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>golden</td>
<td>B10</td>
<td></td>
</tr>
<tr>
<td>gser</td>
<td>B6</td>
<td></td>
</tr>
<tr>
<td>hqr</td>
<td>B11</td>
<td></td>
</tr>
<tr>
<td>hufdec</td>
<td>B20</td>
<td></td>
</tr>
<tr>
<td>hufenc</td>
<td>B20</td>
<td></td>
</tr>
<tr>
<td>hufmak</td>
<td>B20</td>
<td></td>
</tr>
<tr>
<td>hunt</td>
<td>B3</td>
<td></td>
</tr>
<tr>
<td>hypdrv</td>
<td>B6</td>
<td></td>
</tr>
<tr>
<td>hypgeo</td>
<td>B6</td>
<td></td>
</tr>
<tr>
<td>hypser</td>
<td>B6</td>
<td></td>
</tr>
<tr>
<td>icrc</td>
<td>B20</td>
<td></td>
</tr>
<tr>
<td>igray</td>
<td>B20</td>
<td></td>
</tr>
<tr>
<td>index_bpack</td>
<td>B8</td>
<td></td>
</tr>
<tr>
<td>indexx</td>
<td>B8</td>
<td></td>
</tr>
<tr>
<td>interp</td>
<td>B19</td>
<td></td>
</tr>
<tr>
<td>irbit1</td>
<td>B7</td>
<td></td>
</tr>
<tr>
<td>irbit2</td>
<td>B7</td>
<td></td>
</tr>
<tr>
<td>jacobi</td>
<td>B11</td>
<td></td>
</tr>
<tr>
<td>jacobian</td>
<td>B16</td>
<td></td>
</tr>
<tr>
<td>julday</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>kend11</td>
<td>B14</td>
<td></td>
</tr>
<tr>
<td>kend12</td>
<td>B14</td>
<td></td>
</tr>
<tr>
<td>kermom</td>
<td>B18</td>
<td></td>
</tr>
<tr>
<td>ks2d1s</td>
<td>B14</td>
<td></td>
</tr>
<tr>
<td>ks2d2s</td>
<td>B14</td>
<td></td>
</tr>
<tr>
<td>ksone</td>
<td>B14</td>
<td></td>
</tr>
<tr>
<td>kstwo</td>
<td>B14</td>
<td></td>
</tr>
<tr>
<td>laguer</td>
<td>B9</td>
<td></td>
</tr>
</tbody>
</table>
### Appendix C3. Index of Programs and Dependencies

<table>
<thead>
<tr>
<th>Program/Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfit</td>
<td>B15</td>
</tr>
<tr>
<td>gaussj</td>
<td></td>
</tr>
<tr>
<td>covrt</td>
<td></td>
</tr>
<tr>
<td>linbcg</td>
<td>B2</td>
</tr>
<tr>
<td>atimes</td>
<td></td>
</tr>
<tr>
<td>snrm</td>
<td></td>
</tr>
<tr>
<td>asolve</td>
<td></td>
</tr>
<tr>
<td>linmin</td>
<td>B10</td>
</tr>
<tr>
<td>mnbrak</td>
<td></td>
</tr>
<tr>
<td>brent</td>
<td></td>
</tr>
<tr>
<td>lnsrch</td>
<td>B9</td>
</tr>
<tr>
<td>locate</td>
<td></td>
</tr>
<tr>
<td>lop</td>
<td>B19</td>
</tr>
<tr>
<td>lubksb</td>
<td>B2</td>
</tr>
<tr>
<td>ludcmp</td>
<td>B2</td>
</tr>
<tr>
<td>machar</td>
<td>B20</td>
</tr>
<tr>
<td>medfit</td>
<td>B15</td>
</tr>
<tr>
<td>select</td>
<td></td>
</tr>
<tr>
<td>memcof</td>
<td>B13</td>
</tr>
<tr>
<td>mgfas</td>
<td>B19</td>
</tr>
<tr>
<td>rstrct</td>
<td></td>
</tr>
<tr>
<td>slvsm2</td>
<td></td>
</tr>
<tr>
<td>interp</td>
<td></td>
</tr>
<tr>
<td>relax2</td>
<td></td>
</tr>
<tr>
<td>lop</td>
<td></td>
</tr>
<tr>
<td>mglin</td>
<td>B19</td>
</tr>
<tr>
<td>rstrct</td>
<td></td>
</tr>
<tr>
<td>slvsm1</td>
<td></td>
</tr>
<tr>
<td>interp</td>
<td></td>
</tr>
<tr>
<td>relax</td>
<td></td>
</tr>
<tr>
<td>resid</td>
<td></td>
</tr>
<tr>
<td>midexp</td>
<td>B4</td>
</tr>
<tr>
<td>midsq</td>
<td></td>
</tr>
<tr>
<td>midsql</td>
<td></td>
</tr>
<tr>
<td>midsqu</td>
<td></td>
</tr>
<tr>
<td>miser</td>
<td>B7</td>
</tr>
<tr>
<td>ran1</td>
<td></td>
</tr>
<tr>
<td>ran_state</td>
<td></td>
</tr>
<tr>
<td>[func]</td>
<td></td>
</tr>
<tr>
<td>mmid</td>
<td>B16</td>
</tr>
<tr>
<td>mnbrak</td>
<td>B10</td>
</tr>
<tr>
<td>[func]</td>
<td></td>
</tr>
<tr>
<td>mnewt</td>
<td>B9</td>
</tr>
<tr>
<td>[usrfun]</td>
<td></td>
</tr>
<tr>
<td>ludcmp</td>
<td></td>
</tr>
<tr>
<td>lubksb</td>
<td></td>
</tr>
<tr>
<td>moment</td>
<td>B14</td>
</tr>
<tr>
<td>mp2df</td>
<td>B20</td>
</tr>
<tr>
<td>mpops</td>
<td></td>
</tr>
</tbody>
</table>

Sample page from NUMERICAL RECIPES IN FORTRAN 90: THE ART OF PARALLEL SCIENTIFIC COMPUTING (ISBN 0-521-57439-0)

Copyright (C) 1986-1996 by Cambridge University Press. Programs Copyright (C) 1986-1996 by Numerical Recipes Software. Permission is granted for the free use and distribution of this work for non-commercial, educational, or personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server or personal computer, is strictly prohibited. To order Numerical Recipes books or CD-ROMs, visit our web site at http://www.nr.com or call 1-800-872-7423 (North America only) or send email to directcustserv@cambridge.org (outside North America).
Appendix C3. Index of Programs and Dependencies

<table>
<thead>
<tr>
<th>Program</th>
<th>Dependencies</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpdiv</td>
<td>mpinv, mpmul, realft, four1, fourrow</td>
<td>B20</td>
</tr>
<tr>
<td>mpmul</td>
<td>realft, four1, fourrow</td>
<td>B20</td>
</tr>
<tr>
<td>mppi</td>
<td>mpsqrt, mpmul, realft, four1, fourrow</td>
<td>B20</td>
</tr>
<tr>
<td>mprove</td>
<td>lubksb</td>
<td>B2</td>
</tr>
<tr>
<td>mpsqrt</td>
<td>mpmul, realft, four1, fourrow</td>
<td>B20</td>
</tr>
<tr>
<td>mrqmin</td>
<td>gauss, covsrt, [funcs]</td>
<td>B15</td>
</tr>
<tr>
<td>newt</td>
<td>fmin, fdjac, [funcv]</td>
<td>B9</td>
</tr>
<tr>
<td>odeint</td>
<td>[derivs], rkqs, [derivs], rkck</td>
<td>B16</td>
</tr>
<tr>
<td>orthog</td>
<td></td>
<td>B4</td>
</tr>
<tr>
<td>pade</td>
<td>ludcmp, lubksb, mprove, lubksb</td>
<td>B5</td>
</tr>
<tr>
<td>pccheb</td>
<td></td>
<td>B5</td>
</tr>
<tr>
<td>pchift</td>
<td></td>
<td>B5</td>
</tr>
<tr>
<td>pearsn</td>
<td>beta, gamma, betacf</td>
<td>B14</td>
</tr>
<tr>
<td>period</td>
<td>avevar</td>
<td>B13</td>
</tr>
<tr>
<td>plgnrd</td>
<td></td>
<td>B6</td>
</tr>
<tr>
<td>poidev</td>
<td>ran, ranstate, gammeln</td>
<td>B7</td>
</tr>
<tr>
<td>polcoe</td>
<td></td>
<td>B3</td>
</tr>
<tr>
<td>polcof</td>
<td>polint</td>
<td>B3</td>
</tr>
<tr>
<td>poldiv</td>
<td></td>
<td>B5</td>
</tr>
</tbody>
</table>
### Appendix C3. Index of Programs and Dependencies

<table>
<thead>
<tr>
<th>Program</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>polin2</td>
<td>B3</td>
</tr>
<tr>
<td>polint</td>
<td>B3</td>
</tr>
<tr>
<td>powell</td>
<td>B10</td>
</tr>
<tr>
<td>linmin</td>
<td>B13</td>
</tr>
<tr>
<td>mnbrak</td>
<td>B13</td>
</tr>
<tr>
<td>brent</td>
<td>B13</td>
</tr>
<tr>
<td>predic</td>
<td>B13</td>
</tr>
<tr>
<td>probks</td>
<td>B13</td>
</tr>
<tr>
<td>psdes</td>
<td>B13</td>
</tr>
<tr>
<td>pwt</td>
<td>B13</td>
</tr>
<tr>
<td>pwtset</td>
<td>B13</td>
</tr>
<tr>
<td>pythag</td>
<td>B2</td>
</tr>
<tr>
<td>pzextr</td>
<td>B16</td>
</tr>
<tr>
<td>qrdcmp</td>
<td>B2</td>
</tr>
<tr>
<td>qromb</td>
<td>B4</td>
</tr>
<tr>
<td>trapzd</td>
<td>B4</td>
</tr>
<tr>
<td>polint</td>
<td>B4</td>
</tr>
<tr>
<td>qromo</td>
<td>B4</td>
</tr>
<tr>
<td>quadct</td>
<td>B4</td>
</tr>
<tr>
<td>quad3d</td>
<td>B4</td>
</tr>
<tr>
<td>quadmx</td>
<td>B4</td>
</tr>
<tr>
<td>quadvl</td>
<td>B4</td>
</tr>
<tr>
<td>ran</td>
<td>B7</td>
</tr>
<tr>
<td>ran0</td>
<td>B7</td>
</tr>
<tr>
<td>ran1</td>
<td>B7</td>
</tr>
<tr>
<td>ran2</td>
<td>B7</td>
</tr>
<tr>
<td>ran3</td>
<td>B7</td>
</tr>
<tr>
<td>ran_state</td>
<td>B7</td>
</tr>
<tr>
<td>rank</td>
<td>B8</td>
</tr>
<tr>
<td>ratint</td>
<td>B3</td>
</tr>
</tbody>
</table>
Appendix C3. Index of Programs and Dependencies

ratlsq
  [func] .................................................. B5 (p. 1081)
  svdcmp .................................................. pythag
  svbksh ..................................................
  ratval .................................................. B5 (p. 1072)
rc .......................................................... B6 (p. 1134)
rd .......................................................... B6 (p. 1130)
realft —— four1 —— fourrow ................................ B12 (p. 1243)
recur1 ..................................................... B5 (p. 1073)
recur2 ..................................................... B5 (p. 1074)
relax ....................................................... B19 (p. 1338)
relax2 ..................................................... B19 (p. 1341)
resid ....................................................... B19 (p. 1338)
rf ........................................................... B6 (p. 1128)
rj —— rc .................................................... B6 (p. 1131)
  rf ........................................................ B6 (p. 1130)
rk4 —— [derivs] ............................................. B16 (p. 1297)
rkck —— [derivs] ............................................ B16 (p. 1299)
rkdumb —— [derivs] ......................................... B16 (p. 1297)
rkqs —— rkck —— [derivs] ................................ B16 (p. 1298)
rlft2 —— four2 —— fourrow ................................ B12 (p. 1248)
rlft3 —— four3 —— fourrow_3d .............................. B12 (p. 1249)
rotate ....................................................... B2 (p. 1041)
rsolv ....................................................... B2 (p. 1040)
rstrct ...................................................... B19 (p. 1337)
rtbis —— [func] ............................................. B9 (p. 1184)
rtflsp —— [func] .......................................... B9 (p. 1185)
rtnewt —— [funcd] .......................................... B9 (p. 1189)
rtsafe —— [funcd] .......................................... B9 (p. 1190)
rtsec —— [func] ........................................... B9 (p. 1186)
rzextr ...................................................... B16 (p. 1306)
savgol —— ludcmp ........................................... B14 (p. 1283)
  lubksh ......................................................
scrsho —— [func] .......................................... B9 (p. 1182)
select ...................................................... B8 (p. 1177)
select_bypack ............................................. B8 (p. 1178)
select_heap —— sort ....................................... B8 (p. 1179)
select_inplace —— select ................................ B8 (p. 1178)
Appendix C3. Index of Programs and Dependencies

sfroid
  plgnr
  solvde
  difeq

shoot
  [load]
  odeint
    [derivs]
    rkqs
    rkck
      [derivs]
  [score]

shootf
  [load1]
  odeint
    [derivs]
    rkqs
    rkck
      [derivs]
  [score]
  [load2]

simplex

simpr
  ludcmp
  lubksb
  [derivs]

sinft
  realft
  four
  fourrow

slvsml

sncndn

snrm

sobseq

solvde
  difeq

sor

sort

sort2
  indexx

sort3
  indexx

sort_bypack

sort_byreshape

sort_heap

sort_pick

sort_radix

sort_shell

spctrm
  four
  fourrow

spear
  sort2

spheb
  bessjy
  beschb
  chebev
Appendix C3. Index of Programs and Dependencies

sphfpt — newt — fdjac — shootf (q.v.) . . . . B17 (p. 1322)
    — lnsrch — fmin — shootf (q.v)
    — lnsrch — shootf (q.v)
    — ludcmp — lubksb

sphoot — newt — fdjac — shoot (q.v) . . . . B17 (p. 1321)
    — lnsrch — fmin — shoot (q.v)
    — ludcmp — lubksb

spie2 — spline — tridag . . . . . . . . . . . . . B3 (p. 1050)
spinit2 — splint — locate . . . . . . . . . . . . B3 (p. 1051)
    — splint — tridag

spline — tridag . . . . . . . . . . . . . . . . . B3 (p. 1044)
spinit — locate . . . . . . . . . . . . . . . . . . B3 (p. 1045)
sprax . . . . . . . . . . . . . . . . . . . . . . . . B2 (p. 1032)
sprsdag . . . . . . . . . . . . . . . . . . . . . . . B2 (p. 1033)
sprsin . . . . . . . . . . . . . . . . . . . . . . . . B2 (p. 1031)
sparstp . . . . . . . . . . . . . . . . . . . . . . . . B2 (p. 1033)
sprstx . . . . . . . . . . . . . . . . . . . . . . . . B2 (p. 1032)
stifbs — jacobian . . . . . . . . . . . . . . . . . . B16 (p. 1311)
    — simpr — ludcmp — lubksb
    — pzextr

stiff — jacobian . . . . . . . . . . . . . . . . . . B16 (p. 1308)
    — ludcmp — lubksb

stoerm — [derivs] . . . . . . . . . . . . . . . . . . B16 (p. 1307)
svbksb . . . . . . . . . . . . . . . . . . . . . . . . B2 (p. 1022)
svdcmp — pythag . . . . . . . . . . . . . . . . . . B2 (p. 1023)
svdfit — [funcs] . . . . . . . . . . . . . . . . . . . B15 (p. 1290)
    — svdcmp — pythag
    — svdcmp — svbksb

svdvar . . . . . . . . . . . . . . . . . . . . . . . . B15 (p. 1290)
toeplz . . . . . . . . . . . . . . . . . . . . . . . . . B2 (p. 1038)
tptest — avevar . . . . . . . . . . . . . . . . . . . . B14 (p. 1271)
    — beta1 — gamma — betacf

tqli — pythag . . . . . . . . . . . . . . . . . . . . B11 (p. 1228)
trapzd — [func] . . . . . . . . . . . . . . . . . . . B4 (p. 1052)
tred2 . . . . . . . . . . . . . . . . . . . . . . . . . B11 (p. 1227)
tridag . . . . . . . . . . . . . . . . . . . . . . . . . B2 (p. 1018)
### Appendix C3. Index of Programs and Dependencies

<table>
<thead>
<tr>
<th>Program</th>
<th>Dependencies</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttest</td>
<td>avevar</td>
<td>B14</td>
</tr>
<tr>
<td></td>
<td>betai</td>
<td></td>
</tr>
<tr>
<td></td>
<td>gammln</td>
<td></td>
</tr>
<tr>
<td></td>
<td>betacf</td>
<td></td>
</tr>
<tr>
<td>tutest</td>
<td>avevar</td>
<td>B14</td>
</tr>
<tr>
<td></td>
<td>betai</td>
<td></td>
</tr>
<tr>
<td></td>
<td>gammln</td>
<td></td>
</tr>
<tr>
<td></td>
<td>betacf</td>
<td></td>
</tr>
<tr>
<td>twofft</td>
<td>four1</td>
<td>B12</td>
</tr>
<tr>
<td></td>
<td>fourrow</td>
<td></td>
</tr>
<tr>
<td>vander</td>
<td></td>
<td>B2</td>
</tr>
<tr>
<td>vegas</td>
<td>ran1</td>
<td>B7</td>
</tr>
<tr>
<td></td>
<td>ran_state</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[func]</td>
<td></td>
</tr>
<tr>
<td>voltra</td>
<td>[g]</td>
<td>B18</td>
</tr>
<tr>
<td></td>
<td>[ak]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ludcmp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lubksb</td>
<td></td>
</tr>
<tr>
<td>wt1</td>
<td>daub4</td>
<td>B13</td>
</tr>
<tr>
<td>wtn</td>
<td>daub4</td>
<td></td>
</tr>
<tr>
<td>wghts</td>
<td>kermom</td>
<td></td>
</tr>
<tr>
<td>zbrac</td>
<td>[func]</td>
<td>B9</td>
</tr>
<tr>
<td>zbrak</td>
<td>[func]</td>
<td></td>
</tr>
<tr>
<td>zbrent</td>
<td>[func]</td>
<td>B9</td>
</tr>
<tr>
<td>zriddr</td>
<td>[func]</td>
<td>B9</td>
</tr>
<tr>
<td>zroots</td>
<td>laguer</td>
<td>B9</td>
</tr>
<tr>
<td></td>
<td>indexx</td>
<td></td>
</tr>
</tbody>
</table>
General Index to Volumes 1 and 2

In this index, page numbers 1 through 934 refer to Volume 1, *Numerical Recipes in Fortran 77*, while page numbers 935 through 1446 refer to Volume 2, *Numerical Recipes in Fortran 90*. Front matter in Volume 1 is indicated by page numbers in the range 1/i through 1/xxxi, while front matter in Volume 2 is indicated 2/i through 2/xx.

Abstract data types 2/xiii, 1030
Accelerated convergence of series 160ff., 1070
Accuracy 19f.
achieved in minimization 392, 397, 404
achieved in root finding 346f.
contrasted with fidelity 832, 840
CPU different from memory 181
vs. stability 704, 729, 830, 844
Accuracy parameters 1362f.
Acknowledgments 1/xvi, 2/ix
Ada 2/x
Adams-Bashford-Moulton method 741
Adams’ stopping criterion 366
Monte Carlo 306ff., 1161ff.
Addition, multiple precision 907, 1353
Addition theorem, elliptic integrals 255
ADI (alternating direction implicit) method 847, 861ff., 906
Adjoint operator 867
Adobe Illustrator 1/xvi, 2/xx
Additive equation 826
AGM (arithmetic geometric mean) 906
Airy function 204, 234, 243f.
routine for 244f., 1121
Aitken’s delta squared process 160
Aitken’s interpolation algorithm 102
Algol 2/x, 2/xiv
Algorithms, non-numerical 881ff., 1343ff.
Aliasing 495, 569
see also Fourier transform
all() intrinsic function 945, 948
All-poles model 566
see also Maximum entropy method (MEM)
All-zeros model 566
see also Periodogram
Allocatable array 938, 941, 952ff., 1197, 1212, 1266, 1293, 1306, 1336
allocate statement 938f., 941, 953f., 1197, 1266, 1293, 1306, 1336
allocated() intrinsic function 938, 952ff., 1197, 1266, 1293
Allocation status 938, 952ff., 961, 1197, 1266
Alpha AXP 2/xix
Alternating-direction implicit method (ADI) 847, 861f., 906
Alternating series 160f., 1070
Alternative extended Simpson’s rule 128
American National Standards Institute (ANSI) 2/x, 2/xiii
Anoeba 403
see also Simplex, method of Nelder and Mead
Amplification factor 828, 830, 832, 840, 845f.
Amplitude error 831
Analog-to-digital converter 812, 886
Analyticity 195
Analyze/factorize/operate package 64, 824
Anderson-Darling statistic 621
Andrew’s sine 697
average 447
for continuous variables 437, 443ff., 1222
schedule 438
thermodynamic analogy 437
traveling salesman problem 438ff., 1219ff.
ANSI (American National Standards Institute) 2/x, 2/xiii
Antonov-Saleev variant of Sobol’ sequence 300, 1160
any() intrinsic function 945, 948
APL (computer language) 2/xx
Apple 1/xxiii
Macintosh 2/xix, 4, 886
Approximate inverse of matrix 49
Approximation of functions 99, 1043
by Chebyshev polynomials 185f., 513, 1076f.
by Padé approximant 194ff., 1080f.
by rational functions 197ff., 1081f.
by wavelets 594f., 782
see also Fitting
Argument
keyword 2/xiv, 947f., 1341
optional 2/xiv, 947f., 1092, 1228, 1230, 1256, 1272, 1275, 1340
Argument checking 994f., 1086, 1090, 1092, 1370f.
Arithmetic
  arbitrary precision 881, 906ff., 1352ff.
  floating point 881, 906ff., 1352ff.
  IEEE standard 276, 882, 1343
  rounding 882, 1343
Arithmetic coding 881, 902ff., 1349ff.
Arithmetic-geometric mean (AGM) method 906
Arithmetic-if statement 2/xi
Arithmetic progression 971f., 996, 1072, 1127, 1365, 1371f.
Array 953ff.
  allocatable 938, 941, 952ff., 1197, 1212, 1266, 1293, 1306, 1308, 1336
  allocated with pointer 941
  allocation 953
  array manipulation functions 950
  array sections 2/xii, 956, 1336
  associated pointer 953ff.
  assumed-shape 942
  automatic 938, 954, 1197, 1212, 1336
  centered subarray of 113
  conformable to a scalar 942ff., 965, 1094
  constructor 2/xii, 968, 971, 1022, 1052, 1055, 1057, 1059, 1061
  copying 991, 1034, 1327f., 1365f.
  cumulative product 997f., 1072, 1086, 1375
  cumulative sum 997, 1280ff., 1365, 1375
  deallocation 938, 953ff., 1197, 1266, 1293
  disassociated pointer 953
  extents 938, 949
  in Fortran 90 941
  increasing storage for 955, 1070, 1302
  index loss 967f.
  index table 1173ff.
  indices 942
  inquiry functions 948ff.
  intrinsic procedures 2/xiii, 948ff.
  of length 0 944
  of length 1 949
  location of first “true” 993, 1041, 1369
  location of maximum value 993, 1015, 1017, 1365, 1369
  location of minimum value 993, 1369f.
  manipulation functions 950, 1247
  masked swapping of elements in two arrays 1368
  operations on 942, 949, 964ff., 969, 1026, 1040, 1050, 1200, 1326
  outer product 949, 1076
  parallel features 941ff., 964ff., 985
  passing variable number of arguments to function 1022
  of pointers forbidden 956, 1337
  rank 938, 949
  reallocation 955, 992, 1070ff., 1365, 1368f.
  reduction functions 948ff.
  shape 938, 944, 949
  size 938
  skew sections 945, 985
  stride 944
  subscript bounds 942
  subscript triplet 944
  swapping elements of two arrays 991, 1015, 1365f.
  target 938
  three-dimensional, in Fortran 90 1248
  transformational functions 948ff.
  unary and binary functions 949
  undefined status 952ff., 961, 1266, 1293
  zero-length 944
Array section 2/xiii, 943ff., 960
  matches by shape 944
  pointer alias 939, 944f., 1286, 1333
  skew 2/xii, 945, 960, 985, 1284
  vs. eoshift 1078
array-copy() utility function 988, 991, 1034, 1153, 1278, 1328
arth() utility function 972, 974, 988, 996, 1072, 1086, 1127
  replaces do-list 968
Artificial viscosity 831, 837
Ascending transformation, elliptic integrals 256
ASCII character set 6, 888, 896, 902
Assembly language 269
assert() utility function 988, 994, 1086, 1090, 1249
assert_eq() utility function 988, 995, 1022
associated() intrinsic function 952f.
Associated Legendre polynomials 246ff., 764, 1122f., 1319
  recurrence relation for 247
  relation to Legendre polynomials 246
Association, measures of 604, 622ff., 1275
Assumed-shape array 942
Asymptotic series 161
  exponential integral 218
Attenuation factors 583, 1261
Autocorrelation 492
  in linear prediction 558
  use of FFT 538ff., 1254
  Wiener-Khinchin theorem 492, 566f.
AUTODIN-II polynomial 890
Automatic array 938, 954, 1197, 1212, 1336
  specifying size of 938, 954
Automatic deallocation 2/xv, 961
Autonomous differential equations 729f.
Autoregressive model (AR) see Maximum entropy method (MEM)
Average deviation of distribution 605, 1269
Averaging kernel, in Backus-Gilbert method 807
Backsubstitution 33ff., 39, 42, 92, 1017
  in band diagonal matrix 46, 1021
  in Cholesky decomposition 90, 1039
  complex equations 41
  direct for computing $A^{-1} \cdot B$ 40
  with QR decomposition 93, 1040
  relaxation solution of boundary value problems 755, 1316
  in singular value decomposition 56, 1022f.
Backtracking 419
  in quasi-Newton methods 376ff., 1195
Backus-Gilbert method 806ff.
Backus, John 2/x
Backward deflation 363
Index to Volumes 1 and 2

Bader-Deuflhard method 730, 735, 1310f.
Bairstow’s method 364, 370, 1193
Balancing 476f., 1230f.
Band diagonal matrix 42ff., 1019
LU decomposition 45, 1020
multiply by vector 44, 1019
storage 44, 1019
Band-pass filter 551, 554f.
wavelets 584, 592f.
Bandwidth limited function 495
Bank accounts, checksum for 894
Bar codes, checksum for 894
Bartlett window 547, 1254ff.
Basis functions in general linear least squares 665
Bayes’ Theorem 810
Bayesian
approach to inverse problems 799, 810f., 816f.
contrasted with frequentist 810
vs. historic maximum entropy method 816f.
views on straight line fitting 664
Bayes’ shuffle 270
Bernoulli number 132
Bessel functions 223ff., 234ff., 936, 1101ff.
asymptotic form 223f., 229ff.
complex 204
continued fraction 234, 239
double precision 223
fractional order 223, 234ff., 1115ff.
Miller’s algorithm 175, 228, 1106
modified 229ff.
modified, fractional order 239ff.
modified, normalization formula 232, 240
modified, routines for 230ff., 1109ff.
normalization formula 175
parallel computation of 1107ff.
recurrence relation 172, 224, 232, 234
reflection formulas 236
reflection formulas, modified functions 241
routines for 225ff., 236ff., 1101ff.
routines for modified functions 241ff., 1118
series for 160, 223
series for $K_r$ 241
series for $Y_r$ 235
spherical 234, 245, 1121f.
turning point 234
Wronskian 234, 239
Best-fit parameters 650, 656, 660, 698, 1285ff.
see also Fitting
Beta function 206ff., 1089
incomplete see Incomplete beta function
BFGS algorithm see Broyden-Fletcher-Goldfarb-
Shanno algorithm
Bias, of exponent 19
Bias, removal in linear prediction 563
Biconjugacy 77
Biconjugate gradient method
elliptic partial differential equations 824
preconditioning 78ff., 824, 1037
for sparse system 77, 599, 1034ff.
Bicubic interpolation 118f., 1049ff.
Bicubic spline 120f., 1050ff.
Big-endian 293
Bilinear interpolation 117
Binary constant, initialization 959
Binomial coefficients 206ff., 1087f.
recurrences for 209
Binomial probability function 208
cumulative 222f.
devises from 281, 285ff., 1155
Binormal distribution 631, 690
Biorthogonality 77
Bisecion 111, 359, 1045f.
compared to minimum bracketing 390ff.
minimum finding with derivatives 399
root finding 343, 346ff., 352ff., 390, 469,
1184f.
BISYNCH 890
Bit 18
manipulation functions see Bitwise logical functions
reversal in fast Fourier transform (FFT) 499ff., 525
bit_size() intrinsic function 951
Bitwise logical functions 2/xiii, 17, 287,
890f., 951
Block-by-block method 788
Block of statements 7
Bode’s rule 126
Boltzmann probability distribution 437
Boltzmann’s constant 437
Bootstrap method 686f.
Bordering method for Toeplitz matrix 85f.
Borwein and Borwein method for $\pi$ 906,
1357
Boundary 155f., 425f., 745
Boundary conditions
for differential equations 701f.
initial value problems 702
in multigrid method 868f.
partial differential equations 508, 819ff.,
848ff.
for spheroidal harmonics 764
two-point boundary value problems 702,
745ff., 1314ff.
Boundary value problems see Differential
equations; Elliptic partial differential
equations; Two-point boundary value
problems
Box-Muller algorithm for normal deviate 279f.,
215f.
Bracketing
of function minimum 343, 390ff., 402,
1201f.
of roots 341, 343ff., 353ff., 362, 364, 369,
390, 1183ff.
Branch cut, for hypergeometric function 203
Branching 9
Break iteration 14
Brenner, N.M. 500, 517
Brent's method
minimization 389, 395ff., 660ff., 1204ff., 1286
minimization, using derivative 389, 399, 1205
root finding 341, 349, 660ff., 1188ff., 1286
Broadcast (parallel capability) 965ff.
Broyden-Fletcher-Goldfarb-Shanno algorithm
390, 418ff., 1215
Broyden's method 373, 382f., 386, 1199ff.
singular Jacobian 386
btest() intrinsic function 951
Bubble sort 321, 1168
Bugs 4
in compilers 1/xvii
how to report 1/iv, 2/iv
Bulirsch-Stoer
algorithm for rational function interpolation
105ff., 1043
method (differential equations), stepsize control 719, 726
for second order equations 726, 1307
Burg's LP algorithm 561, 1256
Byte 18

C
programming language 13, 2/viii
and case construct 1010
Numerical Recipes in 1, 2/x, 2/xvii
C++ 1/xiv, 2/viii, 2/xvi, 7f.
class templates 1083, 1106
Calendar algorithms 1f., 13ff., 1010ff.
Calibration 653
Capital letters in programs 3, 937
Cards, sorting a hand of 321
Carlson's elliptic integrals 255ff., 1128ff.
case construct 2/xiv, 1010
trapping errors 1036
Cash-Karp parameters 710, 1299f.
Cauchy probability distribution see Lorentzian probability distribution
Cauchy problem for partial differential equations 818ff.
Cayley's representation of \(e^{-iHt}\) 844
CCITT (Comité Consultatif International Télégraphique et Téléphonique) 889f., 901
CCITT polynomial 889f.
ceil() intrinsic function 947
Center of mass 295ff.
Central limit theorem 652f.
Central tendency, measures of 604ff., 1269
Change of variable
in integration 137ff., 788, 1056ff.
in Monte Carlo integration 298
in probability distribution 279
Character functions 952
Character variables, in Fortran 90 1183
Characteristic polynomial
digital filter 554
eigensystems 449, 469
linear prediction 559
matrix with a specified 368, 1193
of recurrence relation 175

Characteristics of partial differential equations 818
Chebyshev acceleration in successive over-relaxation (SOR) 859f., 1332
Chebyshev approximation 84, 124, 183, 184ff., 1076ff.
Clenshaw-Curtis quadrature 190
Clenshaw's recurrence formula 187, 1076
coefficients for \(15f., 1076\)
contrasted with Padé approximation 195
derivative of approximated function 183, 189, 1077f.
economization of series 192f., 195, 1080
for error function 214, 1095
even function 188
and fast cosine transform 513
gamma functions 236, 1118
integral of approximated function 189, 1078
odd function 188
polynomial fits derived from 191, 1078
rational function 197ff., 1081f.
Remes exchange algorithm for filter 553
Chebyshev polynomials 184ff., 1076ff.
continuous orthonormality 184
discrete orthonormality 185
explicit formulas for 184
formula for \(x^k\) in terms of 193, 1080
Chi-square probability function 209ff., 215, 615, 654, 798, 1272
as boundary of confidence region 688f.
related to incomplete gamma function 215
Chi-square test 614f.
for binned data 614f., 1272
chi-by-eye 651
Chi-square fitting see Fitting; Least squares fitting
Chi-square probability function 209ff., 215, 615, 654, 798, 1272
as boundary of confidence region 688f.
related to incomplete gamma function 215
Chi-square test 614f.
for binned data 614f., 1272
chi-by-eye 651
and confidence limit estimation 688f.
for contingency table 623ff., 1275
degrees of freedom 615f.
for inverse problems 797
least squares fitting 653ff., 1285
nonlinear models 675ff., 1292
rule of thumb 655
for straight line fitting 655ff., 1285
for straight line fitting, errors in both coordinates 660, 1286f.
for two binned data sets 616, 1272
unequal size samples 617
Chip rate 290
Chirp signal 556
Cholesky decomposition 89f., 423, 455, 1038
backsubstitution 90, 1039
operation count 90
pivoting 90
solution of normal equations 668
Circulant 585
Class, data type 7
Clenshaw-Curtis quadrature 124, 190, 512f.
Clenshaw’s recurrence formula 176f., 191, 1078
for Chebyshev polynomials 187, 1076
stability 176f.
Clocking errors 891
CM computers (Thinking Machines Inc.) 964
CM Fortran 2/xv
cn function 261, 1137f.
Coarse-grid correction 864f.
Coarse-to-fine operator 864, 1337
Coding
arithmetic 902ff., 1349ff.
checksums 888, 1344
decoding a Huffman-encoded message 900, 1349
Huffman 896f., 1346ff.
run-length 901
variable length code 896, 1346ff.
Ziv-Lempel 896
see also Arithmetic coding; Huffman coding
Coefficients
binomial 208, 1087f.
for Gaussian quadrature 140ff., 1059ff.
for Gaussian quadrature, nonclassical weight function 151ff., 788f., 1064
for quadrature formulas 125ff., 789, 1328
Cohen, Malcolm 2/xiv
Column degeneracy 22
Column operations on matrix 29, 31f.
Column totals 624
Combinatorial minimization see Annealing
Comité Consultatif International Télégraphique et Téléphonique (CCITT) 889f., 901
Common block
obsolescent 2/xif.
superseded by internal subprogram 957, 1067
superseded by module 940, 953, 1298, 1320, 1322, 1324, 1330
Communication costs, in parallel processing 969, 981, 1250
Communication theory, use in adaptive integration 721
Communications protocol 888
Comparison function for rejection method 281
Compilers 964, 1364
CM Fortran 968
DEC (Digital Equipment Corp.) 2/viii
IBM (International Business Machines) 2/vii
Microsoft Fortran PowerStation 2/vii
NAG (Numerical Algorithms Group) 2/viii, 2/xiv
for parallel supercomputers 2/xiv
Complementary error function 1094f.
see Error function
Complete elliptic integral see Elliptic integrals
Complex arithmetic 171f.
avoidance of in path integration 203
 cubic equations 179f.
 for linear equations 41
 quadratic equations 178
Complex error function 252
Complex plane
fractal structure for Newton’s rule 360f.
path integration for function evaluation 201ff., 263, 1138
Complex systems of linear equations 41f.
Compression of data 596f.
Concordant pair for Kendall’s tau 637, 1281
Condition number 53, 78
Confidence level 687, 691ff.
see also
Confidence limits
bootstrap method 687f.
and chi-square 688f.
confidence region, confidence interval 687 on estimated model parameters 684ff.
by Monte Carlo simulation 684ff.
from singular value decomposition (SVD) 693f.
Confluent hypergeometric function 204, 239
Conformable arrays 942f., 1094
Conjugate directions 408f., 414ff., 1210
Conjugate gradient method
biconjugate 77, 1034
compared to variable metric method 418
elliptic partial differential equations 824
for minimization 390, 413ff., 804, 815, 1210, 1211
minimum residual method 78
preconditioner 78f., 1037
for sparse system 77ff., 599, 1034
and wavelets 599
Conservative differential equations 726, 1307
Constrained linear inversion method 799ff.
Constrained linear optimization see Linear programming
Constrained optimization 387
Constraints, deterministic 804ff.
Constraints, linear 423
CONTAINS statement 954, 957, 1067, 1134, 1202
Contingency coefficient C 625, 1275
Contingency table 622f., 638, 1275f.
statistics based on chi-square 623ff., 1275f.
statistics based on entropy 626ff., 1275f.
Continued fraction 163ff.
Bessel functions 234
convergence criterion 165
equivalence transformation 166
evaluation 163ff.
evaluation along with normalization condition 240
even and odd parts 166, 211, 216
even part 249, 251
exponential integral 216
Fresnel integral 248f.
incomplete beta function 219f., 1099f.
incomplete gamma function 211, 1092f.
Lentz’s method 165, 212
modified Lentz’s method 165
Pincherle’s theorem 175
ratio of Bessel functions 239
rational function approximation 164, 211, 219f.
recurrence for evaluating 164f.
and recurrence relation 175
sine and cosine integrals 250ff.
Steed’s method 164ff.
tangent function 164
typography for 163
Continuous variable (statistics) 623
Control structures 7ff., 2/xiv
had 15
named 959, 1219, 1305
Convergence
accelerated, for series 160ff., 1070
of algorithm for pi 906
criteria for 347, 392, 404, 483, 488, 679, 759
eigenvalues accelerated by shifting 470ff.
of golden section search 392f.
of Levenberg-Marquardt method 679
linear 346, 393
of QL method 470ff.
quadratic 49, 351, 356, 409ff., 419, 906
rate 346f., 353, 356
reccurence relation 175
of Ridders’ method 351
series vs. continued fraction 163f.
and spectral radius 856ff., 862
Conversion intrinsic functions 946ff.
Convex sets, use in inverse problems 804
Convolution
denoted by asterisk 492
finite impulse response (FIR) 531
of functions 492, 503f.
of large data sets 536f.
for multiple precision arithmetic 909, 1354
multiplication as 909, 1354
necessity for optimal filtering 535
overlap-add method 537
overlap-save method 536f.
and polynomial interpolation 113
relation to wavelet transform 585
theorem 492, 531ff., 546
treatment of end effects 533
use of FFT 523, 531ff., 1253
wraparound problem 533
Cooley-Tukey FFT algorithm 503, 1250
parallel version 1239ff.
Co-processor, floating point 886
Copyright rules 1/xx, 2/xix
Cornwell-Evans algorithm 816
Corporate promotion ladder 328
Corrected two-pass algorithm 607, 1269
Correction, in multigrid method 863
Correlation coefficient (linear) 630ff., 1276
linear correlation coefficient 630ff., 658, 1276
linear related to least square fitting 630, 658
nonparametric or rank statistical 633ff., 1277
among parameters in a fit 657, 667, 670
in random number generators 268
Spearman rank-order coefficient 634f., 1277
sum squared difference of ranks 634, 1277
Cosine function, recurrence 172
Cosine integral 248, 250ff., 1125ff.
continued fraction 250
routine for 251f., 1125
series 250
Cosine transform see Fast Fourier transform (FFT); Fourier transform
Coulomb wave function 204, 234
count() intrinsic function 948
Courant condition 829, 832ff., 836
multidimensional 846
Courant-Friedrichs-Lewy stability criterion see Courant condition
Covariance
a priori 700
in general linear least squares 667, 671, 1288ff.
matrix, by Cholesky decomposition 91, 667
matrix, of errors 796, 808
matrix, is inverse of Hessian matrix 679
matrix, when it is meaningful 690ff.
in nonlinear models 679, 681, 1292
relation to chi-square 690ff.
from singular value decomposition (SVD) 693f.
in straight line fitting 657
cpu_time() intrinsic function (Fortran 95) 961
CR method see Cyclic reduction (CR)
Cramer’s V 625, 1275
Crank-Nicolson method 840, 844, 846
Cray computers 964
CRC (cyclic redundancy check) 888ff., 1344ff.
CRC-12 890
CRC-16 polynomial 890
CRC-CCITT 890
CRTC (cyclic redundancy check) 888ff., 1344ff.
CRTC-12 890
CRTC-16 polynomial 890
Creativity, essay on 9
Critical (Nyquist) sampling 494, 543
Cross (denotes matrix outer product) 66
Crosstabulation analysis 623
see also Contingency table
Croft’s algorithm 36ff., 45, 1017
cshift() intrinsic function 950
communication bottleneck 969
Cubic equations 178ff., 360
Cubic spline interpolation 107ff., 1044f.
see also Spline
cumprod() utility function 974, 988, 997, 1072, 1086
cumsum() utility function 974, 989, 997, 1280, 1305
Cumulant, of a polynomial 977, 999, 1071f., 1192

Cumulative binomial distribution 222f.
Cumulative Poisson function 214
related to incomplete gamma function 214
Curvature matrix see Hessian matrix
cycle statement 859, 1219
Cycle, in multigrid method 865
Cyclic Jacobi method 459, 1225
Cyclic reduction (CR) 848f., 852ff.
linear recurrences 974
tridiagonal systems 976, 1018
Cyclic redundancy check (CRC) 888ff., 1344f.
Cyclic tridiagonal systems 67, 1030
D.C. (direct current) 492
Danielson-Lanczos lemma 498f., 525, 1235ff.
DAP Fortran 2/xi
Data
assigning keys to 889
continuous vs. binned 614
entropy 626ff., 896, 1275
essay on 603
fitting 650ff., 1285ff.
fractional 655
glitches in 653
iid (independent and identically distributed) 686
modeling 650ff., 1285ff.
serial port 892
smoothing 604, 644ff., 1283f.,
statistical tests 603ff., 1285ff.
unevenly or irregularly sampled 569, 574, 648f., 1258ff.
use of CRCs in manipulating 889
windowing 545ff., 1254
see also Statistical tests
Data compression 596f., 881
arithmetic coding 902f., 1349ff.
approximation for 252f.
cosine transform 513
Huffman coding 896f., 902, 1346ff.
linear predictive coding (LPC) 563ff.
lossless 896
Data Encryption Standard (DES) 290ff., 1144, 1147f., 1156f.
Data hiding 956ff., 1209, 1293, 1296
Data parallelism 941, 964ff., 985
DATA statement 959
for binary, octal, hexadecimal constants 959
repeat count feature 959
superseded by initialization expression 943, 959, 1127
Data type 18, 936
accuracy parameters 1362f.
character 1183
double 2/xii, 937, 1030, 1336, 1346
double precision, for array of arrays 956, 1336
double precision equations 1361
double precision, for Numerical Recipes 1361
double precision, storage allocation 955
DP (double precision) 1361
DPC (double precision complex) 1361
D (1 byte integer) 1361
I (2 byte integer) 1361
I (4 byte integer) 1361
intrinsic 937
LGT (default logical type) 1361
nrtype 190 1361f.
passing complex as real 1140
SP (single precision) 1361f.
SPC (single precision complex) 1361
user-defined 1346
DAUB4 584ff., 588, 590ff., 594, 1264f.
DAUB6 586
DAUB12 598
DAUB20 590ff., 1265
Daubechies wavelet coefficients 584ff., 588, 590f., 594, 598, 1264f.
Davidon-Fletcher-Powell algorithm 390, 418ff., 1215
Dawson’s integral 252ff., 600, 1127f.
Defect, in multigrid method 863
Deferred approach to the limit see Richardson’s deferred approach to the limit
Deflation
of matrix 471
of polynomials 362ff., 370ff., 977
Degeneracy of linear algebraic equations 22, 53, 57, 670
Degenerate kernel 785
Degenerate minimization principle 795
Degrees of freedom 615f., 654, 691
Dekker, T.J. 353
Demonstration programs 3, 936
Deprecated features
common block 2/xii, 940, 953, 957, 1067, 1298, 1320, 1322, 1324, 1330
dble() intrinsic function 947
DEQUATION statement 2/xii, 1161, 1286
statement function 1057, 1256
Derivatives
computation via Chebyshev approximation 183, 189, 1077f.
computation via Savitzky-Golay filters 183, 645
matrix of first partial see Jacobian determinant
matrix of second partial see Hessian matrix
Index to Volumes 1 and 2

use in optimization 388f., 399, 1205ff.
Derived data type see Data type, derived
DES see Data Encryption Standard
Descending transformation, elliptic integrals 256
Descent direction 376, 382, 419
Descriptive statistics 603ff., 1269ff.

see also Statistical tests
Design matrix 645, 665, 795, 801, 1082
Determinant 25, 41
Deviates, random see Random deviates
DFP algorithm see Davidon-Fletcher-Powell algorithm
diagadd() utility function 985, 989, 1004
diagmult() utility function 985, 989, 1004, 1294
Diagonal dominance 43, 679, 780, 856
Difference equations, finite see Finite difference equations (FDEs)
Difference operator 161
Differential equations 701ff., 1297ff.
accuracy vs. stability 704, 729
Adams-Bashforth-Moulton schemes 741
adaptive stepsize control 703, 708ff., 719, 726, 731, 737, 742f., 1298ff., 1303ff., 1308f., 1311ff.
agebraically difficult sets 763
backward Euler’s method 729
Bader-Deuflhard method for stiff 730, 735, 1301ff.
boundary conditions 701f., 745ff., 749, 751f., 771, 1314ff.
Bulirsch-Stoer method 202, 263, 702, 706, 716, 718ff., 740, 1138, 1303
Bulirsch-Stoer method for conservative equations 726, 1307
comparison of methods 702f., 739f., 743 conservative 726, 1307
danger of too small stepsize 714
eigenvalue problem 748, 764ff., 770ff., 1319ff.
embedded Runge-Kutta method 709ff., 731, 1298, 1308
equivalence of multistep and multivalue methods 743
Euler’s method 702, 704, 728f.
forward Euler’s method 728
free boundary problem 748, 776
high-order implicit methods 730ff., 1308ff.
iplicit differencing 729, 740, 1308
initial value problems 702
internal boundary conditions 775f.
internal singular points 775f.
interpolation on right-hand sides 111
Kaps-Rentrop method for stiff 730, 1308
local extrapolation 709
modified midpoint method 716f., 719, 1302f.
multistep methods 740ff.
multivalue methods 740
order of method 704f., 719
path integration for function evaluation 201ff., 263, 1138
predictor-corrector methods 702, 730, 740ff.
reduction to first-order sets 701, 745
relaxation method 746f., 753ff., 1316ff.
relaxation method, example of 764ff., 1319ff.

t.s. independent of \( x \) 729f.
Rosenbrock methods for stiff 730, 1308f.
Runge-Kutta method 702, 704ff., 708ff., 731, 740, 1297f., 1308
Runge-Kutta method, high-order 705, 1297
Runge-Kutta-Fehlberg method 709ff., 1298
scaling stepsize to required accuracy 709
second order 726, 1307
semi-implicit differencing 730
semi-implicit Euler method 730, 735f.
semi-implicit extrapolation method 730, 735f., 1311ff.
semi-implicit midpoint rule 735f., 1310ff.
shooting method 746, 749ff., 1314ff.
shooting method, example 770ff., 1321ff.
similarity to Volterra integral equations 786
singular points 718f., 751, 775ff., 1315f., 1323ff.
step doubling 708f.
stepsize control 703, 708ff., 719, 726, 731, 737, 742f., 1298, 1303f., 1308f.
stepsize control 703, 727ff., 1308f.
stepsize control 703, 727ff., 1308f.
stiff methods compared 739
Stoermer’s rule 726, 1307
see also Partial differential equations; Two-point boundary value problems
Diffusion equation 818, 838ff., 855
Crank-Nicolson method 840, 844, 846
Forward Time Centered Space (FTCS) 839ff., 855
implicit differencing 840
multidimensional 846
Digamma function 216
Digital filtering see Filter
Dihedral group \( D_6 \) 894
dim optional argument 948
Dimensional expansion 965ff.
Dimensions (units) 678
Diminishing increment sort 322, 1168
Dirac delta function 284, 780
Direct method see Periodogram
Direct methods for linear algebraic equations 26, 1014
Direct product see Product of matrices
Direction of largest decrease 410f.
Direction numbers, Sobol’s sequence 300
Dirichlet boundary conditions 820, 840, 850, 856, 858
Disclaimer of warranty 1/xx, 2/xvii
Discordant pair for Kendall’s tau 637, 1281
Discrete convolution theorem 531ff.
Discrete Fourier transform (DFT) 495ff., 1235ff.
  as approximate continuous transform 497
  see also Fast Fourier transform (FFT)
Discrete optimization 456ff., 1219ff.
Discriminant 178, 457
Diskettes
  are ANSI standard 3
  how to order 1/xxi, 2/xvii
Dispersion 831
DISPO see Savitzky-Golay filters
Dissipation, numerical 830
Divergent series 161
Divide and conquer algorithm 1226, 1229
Division
  complex 171
    multiple precision 910ff., 1356
    of polynomials 169, 362, 370, 1072
  dn function 261, 1137f.
Do-list, implied 968, 971, 1127
Do-loop 2/xiv
Do-until iteration 14
Do-while iteration 13
Dogleg step methods 386
Domain of integration 155f.
Dominant solution of recurrence relation 174
Dot (denotes matrix multiplication) 23
  dot_product() intrinsic function 945, 949,
    969, 1216
Double exponential error distribution 696
Double precision
  converting to 1362
    as refuge of scoundrels 882
  use in iterative improvement 47, 1022
Double root 341
Downhill simplex method see Simplex, method
  of Nelder and Mead
DP, defined 937
Driver programs 3
Dual viewpoint, in multigrid method 875
Duplication theorem, elliptic integrals 256
DWT (discrete wavelet transform) see Wavelet
  transform
Dynamical allocation of storage 2/xiii, 869,
  938, 941f., 953ff., 1327, 1336
  garbage collection 956
  increasing 955, 1070, 1302

Earl, D.M. 338
EBCDIC 890
Economization of power series 192f., 195,
  1080
Eigensystems 449ff., 1225ff.
  balancing matrix 476f., 1230f.
  bounds on eigenvalues 50
  calculation of few eigenvalues 454, 488
  canned routines 454f.
  characteristic polynomial 449, 469
  completeness 450
  defective 450, 476, 489
  deflation 471
  degenerate eigenvalues 449ff.
  elimination method 453, 478, 1231
  factorization method 453
  fast Givens reduction 463
  generalized eigenproblem 455
  Givens reduction 462f.
  Hermitian matrix 475
  Hessenberg matrix 453, 470, 476ff., 488,
    1233
  Householder transformation 453, 462ff.,
    469, 473, 475, 478, 1227f., 1231
  ill-conditioned eigenvalues 477
  implicit shifts 472ff., 1228f.
  and integral equations 779, 785
  invariance under similarity transform 452
  inverse iteration 455, 469, 476, 487ff.,
    1230
  Jacobi transformation 453, 456ff., 462,
    475, 489, 1225f.
  left eigenvalues 451
  list of tasks 454f.
  multiple eigenvalues 489
  nonlinear 455
  nonsymmetric matrix 476ff., 1230ff.
  operation count of balancing 476
  operation count of Givens reduction 463
  operation count of Householder reduction
    467
  operation count of inverse iteration 488
  operation count of Jacobi method 460
  operation count of QL method 470, 473
  operation count of QR method for Hessen-
    berg matrices 484
  operation count of reduction to Hessenberg
    form 479
  orthogonality 450
  parallel algorithms 1226, 1229
  polynomial roots and 368, 1193
  QL method 469ff., 475, 488f.
  QL method with implicit shifts 472ff.,
    1228f.
  QR method 52, 453, 456, 469ff., 1228
  QR method for Hessenberg matrices 480ff.,
    1232f.
  real, symmetric matrix 150, 467, 785,
    1225, 1228
  reduction to Hessenberg form 478f., 1231
  right eigenvalues 451
  shifting eigenvalues 449, 470f., 480
  special matrices 454
  termination criterion 484, 488
  tridiagonal matrix 453, 469ff., 488, 1228
  Eigenvalue and eigenvector, defined 449
  Eigenvalue problem for differential equations
    748, 764ff., 770ff., 1319ff.
  Eigenvalues and polynomial root finding 368,
    1193
  EISPACK 454, 475
  Electromagnetic potential 519
  ELEMENTAL attribute (Fortran 95) 961,
    1084
  Elemental functions 2/xiii, 2/xv, 940, 942,
    946f., 961, 986, 1015, 1083, 1097f.
  Elimination see Gaussian elimination
  Ellipse in confidence limit estimation 688
  Elliptic integrals 254ff., 906
    addition theorem 255
Carlson's forms and algorithms 255f., 1128ff.
Cauchy principal value 256f.
duplication theorem 256
Legendre 254ff., 260ff., 1135ff.
routines for 257ff., 1128ff.
symmetric form 255
Weierstrass 255
Elliptic partial differential equations 818, 1332ff.
alternating-direction implicit method (ADI) 861f., 906
analyze/factorize/operate package 824
biconjugate gradient method 824
boundary conditions 820
comparison of rapid methods 854
cyclic reduction 848f., 852ff.
Fourier analysis and cyclic reduction (FACR) 848ff., 854
Gauss-Seidel method 855, 864ff., 876, 1338, 1341
incomplete Cholesky conjugate gradient method (ICCG) 824
Jacobi's method 855f., 864
matrix methods 824
multigrid method 824, 862ff., 1009, 1334ff.
routine for 824, 848ff.
relaxation method 823, 854ff., 1332
strongly implicit procedure 824
successive over-relaxation (SOR) 857ff., 862, 866, 1332

elsewhere construct 943
Emacs, GNU l/xvi
Embedded Runge-Kutta method 709f., 731, 1298, 1308
Encapsulation, in programs 7
Encryption 290, 1156
enddo statement 12, 17
Entropy 896
of data 626ff., 811, 1275
EOM (end of message) 902
eoshift() intrinsic function 950
communication bottleneck 969
vector shift argument 1019f.
vs. array section 1078
epsilon() intrinsic function 951, 1189
Equality constraints 423
Equations
 cubic 178ff., 360
 normal (fitting) 645, 666ff., 800, 1288
 quadratic 20, 178
 see also Differential equations; Partial differential equations; Root finding
Equivalence classes 337f., 1180
EQUIVALEANCE statement 2/xif., 1161, 1286
Equivalence transformation 166
Error
checksums for preventing 891
clocking 891
double exponential distribution 696
local truncation 875
Lorentzian distribution 696f.
in multigrid method 863
nonnormal 653, 690, 694ff.
relative truncation 875
roundoff 180f., 881, 1362
series, advantage of an even 132f., 717, 1362
systematic vs. statistical 653, 1362
truncation 20f., 180, 399, 709, 881, 1362
varieties found by check digits 895
varieties of, in PDEs 831ff.
see also Roundoff error
Error function 213f., 601, 1094f.
approximation via sampling theorem 601
Chebyshev approximation 214, 1095
complex 252
for Fisher's z-transformation 632, 1276
relation to Dawson's integral 252, 1127
relation to Fresnel integrals 248
relation to incomplete gamma function 215
routine for 214, 1094
for significance of correlation 631, 1276
for sum squared difference of ranks 635, 1277
Error handling in programs 2/xii, 2/xvi, 3, 994f., 1036, 1370f.
Estimation of parameters see Fitting; Maximum likelihood estimate
Estimation of power spectrum 542ff., 565ff., 1254ff., 1258
Euler equation (fluid flow) 831
Euler-Maclaurin summation formula 132, 135
Euler's constant 216ff., 250
Euler's method for differential equations 702, 704, 728f.
Euler's transformation 160f., 1070
generalized form 162f.
Evaluation of functions see Function
Even and odd parts, of continued fraction 1364f.
Even parity 888
Exception handling in programs see Error handling in programs
exit statement 959, 1219
Explicit differencing 827
Exponent in floating point format 19, 882, 1343
exponent intrinsic function 1107
Exponential deviate 278, 1151f.
Exponential integral 215ff., 1096f.
asymptotic expansion 218
continued fraction 216
reurrence relation 172
related to incomplete gamma function 215
relation to cosine integral 250
routine for Ei(x) 218, 1097
routine for En(x) 217, 1096
series 216
Exponential probability distribution 570
Extended midpoint rule 124f., 1295, 135, 1054f.
Extended Simpson's rule 128, 788, 790
Extended Simpson's three-eighths rule 789
Extended trapezoidal rule 125, 127, 130ff., 135, 786, 1052ff., 1326
roundoff error 132
Extrapolation (so-called) 574, 1261
Extrapolation 99ff.
  in Bulirsch-Stoer method 718ff., 726, 1305ff.
  differential equations 702
  by linear prediction 557ff., 1256f.
  local 709
  maximum entropy method as type of 567
  polynomial 724, 726, 740, 1305ff.
  rational function 718ff., 726, 1306f.
  relation to interpolation 101
  for Romberg integration 134
see also Interpolation

Extremization see Minimization

\textbf{F}-distribution probability function 222
F-test for differences of variances 611, 613, 1271
FACR see Fourier analysis and cyclic reduction (FACR)
Facsimile standard 901
Factorial
  double (denoted ‘!!’) 247
  evaluation of 159, 1072, 1086
  relation to gamma function 206
  routine for 207ff., 1086ff.
False position 347ff., 1185f.
Family tree 338
FAS (full approximation storage algorithm) 874, 1339ff.
Fast Fourier transform (FFT) 498ff., 881, 981, 1235f.
alternative algorithms 503ff.
  as approximation to continuous transform 497
  Bartlett window 547, 1254
  bit reversal 499f., 525
  and Clenshaw-Curtis quadrature 190
  column-parallel algorithm 981, 1237ff.
  communication bottleneck 969, 981, 1250
  convolution 503ff., 523, 531ff., 909, 1253, 1354
  convolution of large data sets 536ff.
  Cooley-Tukey algorithm 503, 1250
  Cooley-Tukey algorithm, parallel 1239ff.
  correlation 538ff., 1254
  cosine transform 190, 511ff., 851, 1245ff.
  cosine transform, second form 513, 852, 1246
  Danielson-Lanczos lemma 498f., 525
  data sets not a power of 2 503
  data smoothing 645
  data windowing 545ff., 1254
  decimation-in-frequency algorithm 503
  decimation-in-time algorithm 503
  discrete autocorrelation 539, 1254
  discrete convolution theorem 531ff.
  discrete correlation theorem 538
  at double frequency 575
  effect of caching 982
  endpoint corrections 578ff., 1261ff.
  external storage 525
  figures of merit for data windows 548
  filtering 551ff.
  FIR filter 553
  four-step framework 983, 1239
  Fourier integrals 577ff., 1261
  Fourier integrals, infinite range 583
  Hamming window 547
  Hann window 547
  history 498
  IIR filter 553ff.
  image processing 803, 805
  integrals using 124
  inverse of cosine transform 512ff.
  inverse of sine transform 511
  large data sets 525
  leakage 544
memory-local algorithm 528
  multidimensional 515ff., 1236f., 1241,
  1246, 1251
  for multiple precision arithmetic 906
  for multiple precision multiplication 909, 1354
  number-theoretic transforms 503ff.
  operation count 498
  optimal (Wiener) filtering 539ff., 558
  order of storage in 501
  parallel algorithms 981ff., 1235ff.
  partial differential equations 824, 848ff.
  Parzen window 547
periodicity of 497
  periodogram 543ff., 566
  power spectrum estimation 542ff., 1254ff.
  for quadrature 124
of real data in 2D and 3D 519ff., 1248ff.
  of real functions 504ff., 519ff., 1242ff.,
  1248ff.
related algorithms 503ff.
row-parallel algorithm 981, 1235ff.
  Sande-Tukey algorithm 503
  sine transform 508ff., 850, 1245
  Singleton’s algorithm 525
  six-step framework 983, 1240
  square window 546, 1254
  timing 982
  treatment of end effects in convolution 533
  treatment of end effects in correlation 538ff.
  Tukey’s trick for frequency doubling 575
  use in smoothing data 645
  used for Lomb periodogram 574, 1259
  variance of power spectrum estimate 544ff., 549
  virtual memory machine 528
Welch window 547, 1254
  Winograd algorithms 503
see also Discrete Fourier transform (DFT);
  Fourier transform; Spectral density
Faure sequence 300
Fax (facsimile) Group 3 standard 901
Feasible vector 424
FFFT see Fast Fourier transform (FFT)
Field, in data record 329
Figure-of-merit function 650
Filon’s method 583
Filter 551ff.
  acasual 552
  bilinear transformation method 554
  causal 552, 644

Index to Volumes 1 and 2

1457
Index to Volumes 1 and 2

characteristic polynomial 554
data smoothing 644f., 1283f.
digital 551ff.
DISPO 644
by fast Fourier transform (FFT) 523, 551ff.
fine impulse response (FIR) 531, 552
homogeneous modes of 554
infinite impulse response (IIR) 552ff., 566
Kalman 700
linear 552ff.
low-pass for smoothing 644ff., 1283f.
nonrecursive 552
optimal (Wiener) 535, 539ff., 558, 644
quadrature mirror 585, 593
realizable 552, 554ff.
recursive 552ff., 566
Remes exchange algorithm 553
Savitzky-Golay 183, 644ff., 1283f.
stability of 554f.
in the time domain 551ff.
Fine-to-coarse operator 864, 1337
Finite difference equations (FDEs) 753, 763, 774
alternating-direction implicit method (ADI) 847, 861f.
art not science 829
Cayley’s form for unitary operator 844
Courant condition 829, 832ff., 836
Courant condition (multidimensional) 846
Crank-Nicolson method 840, 844, 846
eigenmodes of 827f.
explicit vs. implicit schemes 827
forward Euler 826f.
Forward Time Centered Space (FTCS) 827ff., 839ff., 843, 855
implicit scheme 840
Lax method 828ff., 836
Lax method (multidimensional) 845f.
numerical derivatives 181
partial differential equations 821ff.
in relaxation methods 753ff.
staggered leapfrog method 833f.
upwind differencing 832f., 837
see also Partial differential equations
Finite element methods, partial differential equations 824
Finite impulse response (FIR) 531
Finkelstein, S. 1/xvi, 2/ix
FIR (finite impulse response) filter 552
Fisher’s z-transformation 631f., 1276
Fitting 650ff., 1285ff.
basis functions 665
by Chebyshev approximation 185f., 1076
chi-square 653f., 1285f.
confidence levels related to chi-square values 691ff.
confidence levels from singular value decomposition (SVD) 693f.
confidence limits on fitted parameters 684ff.
covariance matrix not always meaningful 651, 690
degeneracy of parameters 674
an exponential 674
freezing parameters in 668, 700
Gaussians, a sum of 682, 1294
general linear least squares 665ff., 1288, 1290ff.
Kalman filter 700
K–S test, caution regarding 621f.
least squares 651ff., 1285
Legendre polynomials 674, 1291f.
Levenberg-Marquardt method 678ff., 816, 1292f.
linear regression 655ff., 1285ff.
maximum likelihood estimation 652f., 694ff.
Monte Carlo simulation 622, 654, 684ff.
multidimensional 675
nonlinear models 675ff., 1292f.
nonlinear models, advanced methods 683
nonlinear problems that are linear 674
nonnormal errors 656, 690, 694ff.
polynomial 83, 114, 161, 645, 665, 674, 1078, 1291
by rational Chebyshev approximation 197ff., 1081f.
robust methods 694ff., 1294
of sharp spectral features 566
standard (probable) errors on fitted parameters 657f., 661, 671, 684ff., 1285f., 1288, 1290
straight line 655ff., 667f., 698, 1285ff., 1294ff.
straight line, errors in both coordinates 660ff., 1286ff.
see also Error; Least squares fitting; Maximum likelihood estimate; Robust estimation
Five-point difference star 867
Fixed point format 18
Fletcher-Powell algorithm see Davidson-Fletcher-Powell algorithm
Fletcher-Reeves algorithm 390, 414ff., 1214
Floating point co-processor 886
Floating point format 18ff., 882, 1343
care in numerical derivatives 181
IEEE 276, 882, 1343
floor() intrinsic function 948
Flux-conservative initial value problems 825ff.
FMG (full multigrid method) 863, 868, 1334ff.
FOR iteration 9f., 12
forall statement 2/xi, 2/xii
 access to associated index 968
skew array sections 985, 1007
Formats of numbers 18ff., 882, 1343
Fortran 9
arithmetic-if statement 2/xi
COMMON block 2/xif., 953, 957
decprecated features 2/xif., 947, 1057, 1161, 1256, 1286
dynamical allocation of storage 869, 1336
EQUIVALENCE statement 2/xif., 1161, 1286
evolution of 2/xivff.
exception handling 2/xii, 2/xvi
filenames 935
Fortran 2000 (planned) 2/xvi
Fortran 95 2/xv, 945, 947, 1084, 1100, 1364
HPF (High-Performance Fortran) 2/xvf.
Numerical Recipes in 2/x, 2/xvii, 1
obsolescent features 2/xif.
side effects 960
see also Fortran 90
Fortran D 2/xv
Fortran 77 1/xv
bit manipulation functions 17
hexadecimal constants 17
Fortran 8x 2/xi, 2/xiii
Fortran 90 3
abstract data types 2/xiii, 1030
all() intrinsic function 945, 948
allocatable array 938, 941, 953f., 1197,
1212, 1266, 1293, 1306, 1336
allocate statement 938f., 941, 953f., 1197,
1266, 1293, 1306, 1336
allocated() intrinsic function 938, 952ff.,
1197, 1266, 1293
any() intrinsic function 945, 948
array allocation and deallocation 953
array of arrays 2/xii, 956, 1336
array constructor 2/xii, 968, 971, 1022,
1052, 1055, 1127
array constructor with implied do-list 968,
971
array extents 938, 949
array features 941ff., 953ff.
array of length 0 944
array of length 1 949
array manipulation functions 950
array parallel operations 964f.
array rank 938, 949
array reallocation 955
array section 2/xif., 2/xii, 939, 941f.,
960, 1078, 1284, 1286, 1333
array shape 938, 949
array size 938, 942
array transpose 981f.
array unary and binary functions 949
associated() intrinsic function 952f.
associated pointer 953f.
assumed-shape array 942
automatic array 938, 974, 1197, 1212,
1336
backwards-compatibility 935, 946
bit manipulation functions 2/xii, 951
broadcasts 965f.
btest() intrinsic function 951
case construct 1010, 1036
case insensitive 937
ceiling() intrinsic function 947
character functions 952
character variables 1183
cmplx function 1125
communication bottlenecks 969, 981,
1250
compatibility with Fortran 77 935, 946
compilers 2/xii, 2/xiv, 1364
compiling 936
conformable arrays 942f., 1094
CONTAINS statement 954, 957, 985,
1067, 1134, 1202
control structure 2/xiv, 959, 9219, 1305
count() intrinsic function 948
cshl() intrinsic function 950, 969
cycle statement 959, 1219
data hiding 956f., 1209
data parallelism 964
DATA statement 959
data types 937, 1336, 1346, 1361
deallocate statement 938f., 953f., 1197,
1266, 1293
deallocating array 938, 953f., 1197, 1266,
1293
defined types 956
deprecated features 947, 1057, 1161,
1256, 1286
derived types 937, 955
dimensional expansion 965ff.
do-loop 2/xiv
dot_product() intrinsic function 945, 949,
969, 1216
dynamical allocation of storage 2/xiiii,
938, 941f., 953f., 1327, 1336
elemental functions 940, 942, 946f., 951,
1015, 1083, 1364
elsewhere construct 943
eoshift() intrinsic function 950, 969, 1019f.,
1078
epsilon() intrinsic function 951, 1189
evolution 2/xivff., 959, 987f.
example 936
exit statement 959, 1219
exponent() intrinsic function 1107
floor() intrinsic function 948
Fortran tip icon 1009
garbage collection 956
gather-scatter operations 2/xiiii, 969, 981,
984, 1002, 1032, 1034, 1250
generic interface 2/xiiii, 1083
generic procedures 939, 1015, 1083, 1094,
1096, 1364
global variables 955, 957, 1210
history 2/xivf.
huge() intrinsic function 951
iand() intrinsic function 951
 ide() intrinsic function 951
 ibits() intrinsic function 951
 ibset() intrinsic function 951
ieor() intrinsic function 951
IMPLICIT NONE statement 2/xiv, 936
implied do-list 968, 971, 1127
index loss 967f.
initalization expression 943, 959, 1012,
1127
inquiry functions 948
integer model 1144, 1149, 1156
INTENT attribute 1072, 1092
interface 939, 942, 1067, 1084, 1384
internal subprogram 2/xii, 2/xiv, 957,
1057, 1067, 1202f., 1256, 1302
interprocessor communication 969, 981,
1250
intrinsic data types 937
intrinsic functions 940, 942, 1015, 1083

Index to Volumes 1 and 2
1459
intrinsic procedures 939, 945ff., 987, 1016
intrinsic function 951
ISO (International Standards Organization) 2/xii, 2/xiiif.
kind() intrinsic function 951
KIND parameter 937, 946, 1125, 1144, 1192, 1254, 1261, 1284, 1361
keyword argument 2/xiv, 947f., 1341
kind() intrinsic function 951
KIND parameter 937, 946, 1125, 1144, 1192, 1254, 1261, 1284, 1361
language features 935ff.
lexical comparison 952
linear algebra 969f., 1000ff., 1018f., 1026, 1040, 1200, 1326
linear recurrence 971, 988
linking 936
literal constant 937, 1361
logo for tips 2/viii, 1009
mask 948, 967f., 1006f., 1038, 1102, 1200, 1226, 1305, 1333f., 1368, 1378, 1382
matmul() intrinsic function 945, 949, 969, 1026, 1040, 1050, 1076, 1200, 1216, 1290, 1326
maxexponent() intrinsic function 1107
maxloc() intrinsic function 949, 961
maxval() intrinsic function 945, 948, 961, 1016, 1273
memory leaks 953, 956, 1327
memory management 938, 953f., 957ff.
merge() intrinsic function 945, 950, 1010, 1094f., 1099f.
Metcalfe and Reid (M&R) 935
minloc() intrinsic function 949, 961
minval() intrinsic function 948, 961
missing language features 983ff., 987ff.
modulo() intrinsic function 946, 1156
module() intrinsic function 946, 1156
module constant 940, 1012, 1361
module control structure 959, 1219, 1305
nearest() intrinsic function 952, 1146
nested where construct forbidden 943
not() intrinsic function 951
nullify statement 953f., 1070, 1302
numeric representation functions 951
ONLY option 945, 951, 1067
operator overloading 2/xiiif.
operator, user-defined 2/xii
optional argument 2/xiv, 947f., 1092, 1228, 1230, 1256, 1272, 1275, 1340
outer product 969f.
overloading 940, 1083, 1102
pack() intrinsic function 945, 950, 964, 969, 991, 1170, 1176, 1178
pack, for selective evaluation 1087
Parallel extensions 2/xv, 959ff., 964, 981, 984, 987, 1002, 1032
parallel programming 963ff.
PARAMETER attribute 1012
pointer 2/xiiif., 938f., 941, 944f., 952ff., 1067, 1070, 1197, 1210, 1212, 1266, 1302, 1327, 1336
pointer to function (missing) 1067
PORTABILITY 963
present() intrinsic function 952
PRIVATE attribute 957, 1067
product() intrinsic function 948
programming conventions 937
PUBLIC attribute 957, 1067
quick start 936
radix() intrinsic function 1231
random_number() intrinsic function 1141, 1143
random_number() intrinsic function 1141
real() intrinsic function 947, 1125
RECURSIVE keyword 958, 1065, 1067
recursive procedure 2/xiv, 958, 1065, 1067
reduction functions 948
reshape() intrinsic function 950, 969, 1247
RESULT keyword 958, 1073
SAVE attribute 953f., 958ff., 1052, 1070, 1266, 1293
scale() intrinsic function 1107
scatter-with-combine (missing function) 984
scope 956ff.
scoping units 939
select case statement 2/xiv, 1010, 1036
shape() intrinsic function 938, 949
size() intrinsic function 938, 942, 945, 948
skip sections 985
sparse matrix representation 1030
specification statement 2/xiv
spread() intrinsic function 945, 950, 966ff., 969, 1000, 1094, 1290f.
statement functions deprecated 1057
stride (of an array) 944
structure constructor 2/xii
subscript triplet 944
sum() intrinsic function 945, 948, 966
tiny() intrinsic function 952
transformational functions 948
transpose() intrinsic function 950, 960, 969, 981, 1247
tricks 1009, 1072, 1146, 1274, 1278, 1280
truncation elemental functions 946
type checking 1140
undefined pointer 953
unpack() intrinsic function 950, 964, 969
USE statement 936, 939f., 954, 957, 1067, 1384
utility functions 987ff.
vector subscripts 2/xiiif., 969, 981, 984, 1002, 1032, 1250
visibility 956ff., 1209, 1293, 1296
WG5 technical committee 2/xii, 2/xiii, 2/xv
where construct 943, 985, 1060, 1291
X3J3 Committee 2/viii, 2/xiff., 2/xv, 947, 959, 964, 968, 990
zero-length array 944
see also Intrinsic procedures
see also Fortran

Fortran 95  947, 959ff.
allocatable variables  961
blocks  960
cpu_time() intrinsic function  961
elemental functions  2/xiii, 2/xv, 940, 961, 986, 1015, 1083f., 1097f.
forall statement  2/xii, 2/xv, 960, 964, 968, 986, 1007
initialization of derived data type  2/xv
initialization of pointer  2/xv, 961
minor changes from Fortran 90  961
modified intrinsic functions  961
nested where construct  2/xv, 960, 1100
pointer association status  961
pointers  961
PURE attribute  2/xv, 960f., 964, 986
SAVE attribute  961
side effects  960
and skew array section  945, 985

see also Fortran

Fortran 2000  2/xvi
Forward deflation  363
Forward difference operator  161
Forward Euler differencing  826f.
Forward Time Centered Space see FTCS
Four-step framework, for FFT  983, 1239
Fourier analysis and cyclic reduction (FACR)  848f., 854
Fourier integrals
attenuation factors  583, 1261
equality signs  578f., 1261
tail integration by parts  583
use of fast Fourier transform (FFT)  577ff., 1261ff.
Fourier transform  99, 400ff., 1235ff.
alizing  495, 569
approximation of Dawson’s integral  253
autocorrelation  492
basis functions compared  508f.
contrast with wavelet transform  584, 594
convolution  492, 503f., 531ff., 909, 1253, 1354
correlation  492, 538f., 1254
cosine transform  190, 511ff., 851, 1245f.
cosine transform, second form  513, 852, 1246
critical sampling  494, 543, 545
definition  490
discrete Fourier transform (DFT)  184, 495ff.
Gaussian function  600
image processing  803, 805
infinite range  583
inverse discrete Fourier transform  497
method for partial differential equations  848ff.
missing data  569
missing data, fast algorithm  574f., 1259
Nyquist frequency  494ff., 520, 543, 545, 569, 571
optimal (Wiener) filtering  539ff., 558
Parseval’s theorem  492, 498, 544
powered spectral density (PSD)  492f.
power spectrum estimation by FFT  542ff., 1254ff.
power spectrum estimation by maximum entropy method  568ff., 1258
properties of  491f.
sampling theorem  495, 543, 545, 600
scalings of  491
significance of a peak in  570
sine transform  508ff., 850, 1245
symmetries of  491
time sampling, fast algorithm  574f., 1259
time unevenly sampled data  569ff., 574, 1258
and wavelets  592f.
Wiener-Khinchin theorem  492, 558, 566f.
see also Fast Fourier transform (FFT);
Spectral density
Fractal region  360f.
Fractional step methods  847f.
Fredholm alternative  780
Fredholm equations  779f.
eigenvalue problems  780, 785
error estimate in solution  784
first kind  779
Fredholm alternative, second kind  785, 1325
homogeneous vs. inhomogeneous  779f.
il-conditioned  780
infinite range  789
inverse problems  780, 795ff.
kernel  779f.
nonlinear  781
Nyström method  782ff., 789, 1325
product Nyström method  789, 1328ff.
second kind  779f., 782ff., 1325, 1331
with singularities  788, 1328ff.
with singularities, worked example  792, 1328ff.
subtraction of singularity  789
symmetric kernel  785
see also Inverse problems
Frequency domain  490
Frequency spectrum see Fast Fourier transform (FFT)
Frequentist, contrasted with Bayesian  810
Fresnel integrals  248ff.
asymptotic form  249
continued fraction  248f.
routine for  249f., 1123
series  248
Friday the Thirteenth  14f., 1011f.
FTCS (forward time centered space)  827ff., 839ff., 843
stability of  827ff., 839ff., 855
Full approximation storage (FAS) algorithm  874, 1339ff.
Full moon  14f., 936, 1011f.
Full multigrid method (FMG)  863, 868, 1334ff.
Full Newton methods, nonlinear least squares  683
Full pivoting  29, 1014
Full weighting  867
Function
Airy  204, 243f., 1121
approximation 99ff., 184ff., 1043, 1076ff.
associated Legendre polynomial 246ff.,
764, 1122f., 1319
autocorrelation of 492
bandwidth limited 495
Bessel 172, 204, 223ff., 234, 1101ff.,
1115ff.
beta 209, 1089
binomial coefficients 208f., 1087f.
branch cuts of 202f.
display 215, 798
confluent hypergeometric 204, 239
correlation of 492
cosine integral 250ff., 1123f.
Coulomb wave 204, 234
cumulative binomial probability 222f.
cumulative Poisson 209ff.
Dawson’s integral 252ff., 600, 1127f.
digamma 216
eff 254ff., 906, 1128ff.
evaluation 159ff., 1070ff.
evaluation by path integration 201ff., 263,
1138
exponential integral 172, 215ff., 250,
1096f.
F-distribution probability 222
Fresnel integral 248ff., 1123
gamma 206, 1085
hypergeometric 202f., 263ff., 1138ff.
inclement beta 219ff., 610, 1098ff., 1269
incomplete gamma 209ff., 615, 654, 657f.,
1098ff., 1272, 1285
inverse hyperbolic 178, 255
inverse trigonometric 255
Jacobi elliptic 261, 1137f.
Kolmogorov-Smirnov probability 618f.,
640, 1274, 1281
Legendre polynomial 172, 246, 674, 1122,
1291
logarithm 255
modified Bessel 229ff., 1109ff.
modified Bessel, fractional order 239ff.,
1118ff.
overloading 1083
parallel evaluation 986, 1009, 1084, 1087,
1090, 1102, 1128, 1134
path integration to evaluate 201ff.
pathological 99f., 343
Poisson cumulant 214
representations of 490
routine for plotting a 342, 1182
sine and cosine integrals 248, 250ff.,
1125f.
sn, dn, cn 261, 1137f.
spherical harmonics 246ff., 1122
spheroidal harmonic 764ff., 770ff., 1319ff.,
1323ff.
Student’s probability 221f.
variable number of arguments 1022
Weber 204

Functional iteration, for implicit equations
740ff.
FWHM (full width at half maximum) 548ff.

Gamma deviate 282f., 1153f.
Gamma function 206ff., 1085
incomplete see Incomplete gamma function
Garbage collection 956
Gather-scatter operations 2xuif., 984, 1002,
1032, 1034
communication bottleneck 969, 981, 1250
many-to-one 984, 1002, 1032, 1034
Gauss-Chebyshev integration 141, 144, 512f.
Gauss-Hermite integration 144, 789
abscissas and weights 147, 1062
normalization 147
Gauss-Jacobi integration 144
abscissas and weights 148, 1063
Gauss-Jordan elimination 27ff., 33, 64, 1014f.
operation count 34, 39
solution of normal equations 667, 1288
storage requirements 30
Gauss-Kronrod quadrature 154
Gauss-Laguerre integration 144, 789, 1060
Gauss-Legendre integration 145f., 1059
see also Gaussian integration
Gauss-Lobatto quadrature 154, 190, 512
Gauss-Radau quadrature 154
Gauss-Seidel method (relaxation) 855, 857,
864ff., 1338
nonlinear 876, 1341
Gauss transformation 256
Gaussian (normal) distribution 267, 652, 798
central limit theorem 652f.
deviates from 279ff., 571, 1152
kurtosis of 606
multivariate 690
semi-invariants of 608
tails compared to Poisson 653
two-dimensional (binormal) 631

Gaussian function

Hardy’s theorem on Fourier transforms 600
see also Gaussian (normal) distribution
calculation of abscissas and weights 142ff.,
1009, 1059ff.
error estimate in solution 784
extensions of 153f.
Golub-Welsch algorithm for weights and
abscissas 150, 1064
for integral equations 781, 783, 1325
from known recurrence relation 150, 1064
nonclassical weight function 151ff., 788ff., 1064f., 1328f.
and orthogonal polynomials 142, 1009, 1061
parallel calculation of formulas 1009, 1061
preassigned nodes 153ff., 151ff., 788f., 1064f., 1328f.

Gear's method (stiff ODEs) 730
Geiger counter 266
Generalized eigenvalue problems 455
Generalized minimum residual method (GMRES) 78
Generic interface see Interface, generic
Generic procedures 939, 1094, 1096, 1328f.
elemental 940, 942, 946f., 1015, 1083

Geometric progression 972, 996f., 1365, 1372ff.
geop() utility function 972, 974, 989, 996, 1127
Geophysics, use of Backus-Gilbert method 809
Gerchberg-Saxton algorithm 805
get_diag() utility function 985, 989, 1005, 1226
Gilbert and Sullivan 714
Givens reduction 462f., 473
fast 463
operation count 463
Glassman, A.J. 180
Global optimization 387f., 436ff., 650, 1219ff.
continuous variables 443f., 1222
Global variables 940, 953f., 1094, 1096, 1364
communicated via internal subprogram 954, 957f., 1067, 1222
pointer method 954, 1197, 1212, 1266, 1287, 1302

Globally convergent minimization 415f., 1215
root finding 373, 376ff., 395, 1196, 1314f.

GMRES (generalized minimum residual method) 78
 GNU Emacs 1/xvi
Godunov's method 837
Golden mean (golden ratio) 21, 249, 392ff., 399
Golden section search 341, 389ff., 395, 1202ff.
Golub-Welsch algorithm, for Gaussian quadrature 150, 1064
Goodness-of-fit 650, 654, 657f., 662, 690, 1285
GOTO statements, danger of 9, 959
Gram-Schmidt
biortogonalization 415f.
orthogonalization 94, 450f., 1039
SVD as alternative to 58
Graphics, function plotting 342, 1182f.
Gravitational potential 519

Gray code 300, 881, 886ff., 1344
Greenbaum, A. 79
Gregorian calendar 13, 16, 1011, 1013
Grid square 116f.
Group, dihedral 894, 1345
Guard digits 882, 1343

Half weighting 867, 1337
Halton's quasi-random sequence 300
Hamming window 547
Hamming's motto 341
Hann window 547
Harmonic analysis see Fourier transform
Hashing 293, 1144, 1148, 1156
for random number seeds 1147f.
HDLC checksum 890
Heap (data structure) 327f., 336, 897, 1179
Heapsort 320, 327f., 336, 1171ff., 1179
Helmholtz equation 852
Hermite polynomials 144, 147
approximation of roots 1062
Hermitian matrix 450f., 475
Hertz (unit of frequency) 490
Hessenberg matrix 94, 453, 470, 476ff., 488, 1231
see also Matrix
Hessian matrix 382, 408, 415f., 419f., 676ff., 803, 815
is inverse of covariance matrix 597, 779
second derivatives in 676
Hexadecimal constants 17f., 276, 293
initialization 959
Hierarchically band diagonal matrix 598
Hierarchy of program structure 6ff.
High-order not same as high-accuracy 100f., 124, 389, 399, 705, 709, 741
High-pass filter 551
High-Performance Fortran (HPF) 2/xvi, 964, 981, 984
scatter-with-add 1032
Hilbert matrix 83
Home page, Numerical Recipes 1/xx, 2/xvii
Homogeneous linear equations 53
Hook step methods 386
Hotelling's method for matrix inverse 49, 598
Householder transformation 52, 453, 462ff., 469, 473, 475, 478, 481ff., 1227f.
operation count 467
in QR decomposition 92, 1039
HPF see High-Performance Fortran
Huffman coding 564, 881, 896ff., 902, 1346ff.
huge() intrinsic function 951
Hyperbolic functions, explicit formulas for inverse 178
Hyperbolic partial differential equations 818
advective equation 826
flux-conservative initial value problems 825ff.
Hypergeometric function 202f., 263ff.
routine for 264f., 1138
Hypothesis, null 603

I2B, defined 937
I4B, defined 937
iand() intrinsic function 951
ibclr() intrinsic function 951
ibits() intrinsic function 951
IBM 1/xxiii, 2/xxix
bad random number generator 268
Fortran 90 compiler 2/viii
PC 4, 276, 293, 886
PC-RT 4
radix base for floating point arithmetic 476
IBM checksum 894
ibset() intrinsic function 951
ICCG (incomplete Cholesky conjugate gradient method) 824
ICF (intrinsic correlation function) model 817
Identity (unit) matrix 25
IEEE floating point format 276, 882f., 1343
ieor() intrinsic function 951
if statement, arithmetic 2/xi
if structure 12f.
ifirstloc() utility function 989, 993, 1041, 1346
IIR (infinite impulse response) filter 552ff., 566
Ill-conditioned integral equations 780
Image processing 519, 803
cosine transform 513
fast Fourier transform (FFT) 519, 523, 803
as an inverse problem 803
maximum entropy method (MEM) 809ff.
from modulus of Fourier transform 805
wavelet transform 596f., 1267f.
imaxloc() utility function 989, 993, 1041, 1346
iminloc() utility function 989, 993, 1046, 1346
Implicit
function theorem 340
pivoting 30, 1014
shifts in QL method 472ff.
Implicit differencing 827
for diffusion equation 840
for stiff equations 729, 740, 1308
IMPLICIT NONE statement 2/xiv, 936
Implied do-list 968, 971, 1127
Importance sampling, in Monte Carlo 306f.
Improper integrals 135ff., 1055
Impulse response function 531, 540, 552
IMSL 1/xxiii, 2/xx, 26, 64, 205, 364, 369, 454
In-place selection 335, 1178f.
Included file, superseded by module 940
Incomplete beta function 219ff., 1098ff.
for F-test 613, 1271
routine for 220ff., 1097
for Student’s t 610, 613, 1269
Incomplete Cholesky conjugate gradient method
(ICCG) 824
Incomplete gamma function 209ff., 1089ff.
for chi-square 615, 654, 657f., 1272, 1285
deviates from 282f., 1153
in mode estimation 610
routine for 211f., 1089
Increment of linear congruential generator 268
Indentation of blocks 9
Index 934ff., 1446ff.
this entry 1464
Index loss 967f., 1038
Index table 320, 329f., 1173ff., 1176
Inequality constraints 423
Inheritance 8
Initial value problems 702, 818f.
see also Differential equations;
Partial differential equations
Initialization of derived data type 2/xxv
Initialization expression 943, 959, 1012, 1127
Injection operator 864, 1337
Instability see Stability
Integer model, in Fortran 90 1144, 1149, 1156
Integer programming 436
Integral equations 779ff.
adaptive stepsize control 788
block-by-block method 788
correspondence with linear algebraic equations 779ff.
degenerate kernel 785
eigenvalue problems 780, 785
ingraduate in solution 784
Fredholm 779f., 782ff., 1325, 1331
Fredholm alternative 780
homogeneous, second kind 785, 1325
ill-conditioned 780
infinite range 789
inverse problems 780, 795ff.
kernel 779
nonlinear 781, 787
Nyström method 782ff., 789, 1325
product Nyström method 789, 1328ff.
with singularities 788ff., 1328ff.
with singularities, worked example 792, 1328ff.
subtraction of singularity 789
symmetric kernel 785
unstable quadrature 787f.
V olterra 780f., 786ff., 1326f.
wavelets 782
see also Inverse problems
Integral operator, wavelet approximation of 597, 782
Integration of functions 123ff., 1052ff.
cosine integrals 250, 1125
Fourier integrals 577ff., 1261
Fourier integrals, infinite range 583
Fresnel integrals 248, 1123
Gauss-Hermite 147f., 1062
Gauss-Jacobi 148, 1063
Gauss-Laguerre 146, 1060
Gauss-Legendre 145, 1059
integrals that are elliptic integrals 254
path integration 201ff.
sine integrals 250, 1125
see also Quadrature
Integro-differential equations 782
INTENT attribute 1072, 1092
Interface (Fortran 90) 939, 942, 1067
for communication between program parts 957, 1209, 1293, 1296
explicit 939, 942, 1067, 1384
generic 2/xiii, 940, 1015, 1083, 1094, 1096
implicit 939
for Numerical Recipes 1384ff.
Interface block 939, 1084, 1384
Interface, in programs 2, 8
Intermediate value theorem 343
Internal subprogram (Fortran 90) 2/xiv, 954, 957, 1067, 1202f., 1226
nesting of 2/xii
resembles C macro 1302
supersedes statement function 1057, 1256
International Standards Organization (ISO) 2/xf., 2/xiii
Internet, availability of code over 1/xx, 2/xvii
Interpolation 99ff.
Aitken's algorithm 102
avoid 2-stage method 100
avoid in Fourier analysis 569
bicubic 118f., 1049f.
bilinear 117
cautions on high-order 100
coefficients of polynomial 100, 113ff., 191, 575, 1047f., 1078
for computing Fourier integrals 578
error estimates for 100
of functions with poles 104ff., 1043f.
inverse quadratic 353, 395ff., 1204
multidimensional 101f., 116ff., 1049ff.
in multidimensional method 866, 1337
Neville's algorithm 102f., 182, 1043
Nystrom 783, 1326
offset arrays 104, 113
operation count for 100
operator 864, 1337
order of 100
and ordinary differential equations 101
oscillations of polynomial 100, 116, 389, 399
parabolic, for minimum finding 395, 1204
polynomial 99, 102ff., 182, 1043
rational Chebyshev approximation 197ff., 1081
rational function 99, 104ff., 194ff., 225, 718ff., 726, 1043f., 1080, 1306
reverse (extrapolation) 574, 1261
spline 100, 107ff., 120f., 1044ff., 1050f.
trigonometric 99
see also Fitting
Interprocessor communication 969, 981
Interval variable (statistics) 623
Intrinsic correlation function (ICF) model 817
Intrinsic data types 937
Intrinsic procedures
array inquiry 938, 942, 948ff.
array manipulation 950
array reduction 948
array unary and binary functions 949
backwards-compatibility 946
bit manipulation 2/xiii, 951
character 952
cmplx 1254
conversion elemental 946
elemental 940, 942, 946f., 951, 1083, 1364
generic 939, 1083f., 1364
lexical comparison 952
numeric inquiry 2/xiv, 1107, 1231, 1343
numerical 946, 951f.
numerical representation 951
pack used for sorting 1171
random number 1143
real 1254
top 120 945
truncation 946f.
see also Fortran 90
Inverse hyperbolic function 178, 255
Inverse iteration see Eigensystems
Inverse problems 779, 795ff.
Backus-Gilbert method 806ff.
Bayesian approach 799, 810f., 816f.
central idea 799
constrained linear inversion method 799ff.
data inversion 807
deterministic constraints 804ff.
in geophysics 809
Gerchberg-Saxton algorithm 805
incomplete Fourier coefficients 813
and integral equations 780
linear regularization 799ff.
maximum entropy method (MEM) 810, 815f.
MEM demystified 814
principal solution 797
regularization 796ff.
regularizing operator 798
stabilizing functional 798
Tikhonov-Miller regularization 799ff.
trade-off curve 795
trade-off curve, Backus-Gilbert method 809
two-dimensional regularization 803
use of conjugate gradient minimization 804, 815
use of convex sets 804
use of Fourier transform 803, 805
Van Cittert's method 804
Inverse quadratic interpolation 353, 395ff., 1204
Inverse response kernel, in Backus-Gilbert method 807
Inverse trigonometric function 255
ior() intrinsic function 951
ISBN (International Standard Book Number)
checksum 894
ishft() intrinsic function 951
ishftc() intrinsic function 951
ISO (International Standards Organization) 2/xf., 2/xiii
Iterated integrals 155
Iteration 9f.
functional 740f.
to improve solution of linear algebraic equations 47ff., 195, 1022
for linear algebraic equations 26
required for two-point boundary value problems 745
in root finding 340f.
Iteration matrix 856
ITPACK 71
Iverson, John 2/xi

Jacobi matrix, for Gaussian quadrature 150, 1064
Jacobi polynomials, approximation of roots 1064
Jacobi transformation (or rotation) 94, 453, 456ff., 462, 475, 489, 1041, 1225
Jacobian determinant 279, 774
Jacobian elliptic functions 261, 1137f.
Jacobian matrix 374, 376, 379, 382, 731, 1197f., 1309
singular in Newton’s rule 386
Jacobi’s method (relaxation) 855ff., 864
Jenkins-Traub method 369
Julian Day 1, 13, 16, 936, 1010ff.
Jump transposition errors 895

K-S test see Kolmogorov-Smirnov test
Kalman filter 700
Kanji 2/xii
Kaps-Rentrop method 730, 1308
Kendall’s tau 634, 637ff., 1279
Kennedy, Ken 2/xv
Kepler’s equation 1061
Kermit checksum 889
Kernel 779
averaging, in Backus-Gilbert method 807
degenerate 785
finite rank 785
inverse response 807
separable 785
singular 788f., 1328
symmetric 785
Keys used in sorting 329, 889
Keyword argument 2/xiv, 947f., 1341
kind() intrinsic function 951
and cmplx() intrinsic function 1125, 1192, 1254
and real() intrinsic function 1125, 1192, 1254
default 937
for Numerical Recipes 1361
for random numbers 1144
and real() intrinsic function 1125
Kolmogorov-Smirnov test 614, 617ff., 694, 1273f.
two-dimensional 640, 1281f.
variants 620ff., 640, 1281
Kuiper’s statistic 621
Kurtosis 606, 608, 1269

L-estimate 694
Labels, statement 9
Lag 492, 538, 553
Lagged Fibonacci generator 1142, 1148ff.
Lagrange multiplier 795
Lagrange’s formula for polynomial interpolation 84, 102f., 575, 578
Laguerre polynomials, approximation of roots 1061
Laguerre’s method 341, 365ff., 1191f.
Lanczos lemma 498f.
Lanczos method for gamma function 206, 1085
Landen transformation 256
LAPACK 26, 1230
Laplace’s equation see also Poisson equation
Las Vegas 625
Latin square or hypercube 305f.
contrasted to general minimization problems 684ff.
degeneracies in 671f., 674
Fourier components 696
as M-estimate for normal errors 696
as maximum likelihood estimator 652
as method for smoothing data 645, 1283
Fourier components 1258
freezing parameters in 668, 700
general linear case 665ff., 1288, 1290f.
Levenberg-Marquardt method 678ff., 816, 1292f.
Lomb periodogram 570, 1258
multidimensional 675
nonlinear 386, 675ff., 816, 1292
nonlinear, advanced methods 683
normal equations 645, 666f., 800, 1288
normal equations often singular 670, 674
optimal (Wiener) filtering 540f.
QR method in 94, 668
for rational Chebyshev approximation 199f., 1081f.
relation to linear correlation 630, 658
Savitzky-Golay filter as 645, 1283
singular value decomposition (SVD) 25f., 518f., 199f., 670ff., 1081, 1290
skewed by outliers 653
for spectral analysis 570, 1258
standard (probable) errors on fitted parameters 667, 671
weighted 652
see also Fitting
L’Ecuyer’s long period random generator 271, 273
Least squares fitting
standard (probable) errors on fitted parameters 1288, 1290
weighted 1285
Left eigenvalues or eigenvectors 451
Legal matters 1/xx, 2/xxii
Legendre elliptic integral see Elliptic integrals
Legendre polynomials 246, 1122
fitting data to 674, 1291ff.
recurrence relation 172
shifted monic 151
see also Associated Legendre polynomials;
Spherical harmonics
Lehmer-Schur algorithm 369
Lemarie’s wavelet 593
Lentz’s method for continued fraction 165, 212
Lepage, P. 309
Leptokurtic distribution 606
Levenberg-Marquardt algorithm 386, 678ff., 816, 1292
advanced implementation 683
Levinson’s method 86, 1038
Lewis, H.W. 275
Lexical comparison functions 952
LGT, defined 937
License information 1/xx, 2/xviiff.
Limbo 356
Limit cycle, in Laguerre’s method 365
Line minimization see Minimization, along a ray
Line search see Minimization, along a ray
Linear algebra, intrinsic functions for parallelization 969ff., 1026, 1040, 1200, 1326
Linear algebraic equations 22ff., 1014
band diagonal 43ff., 1019
biconjugate gradient method 77, 1034ff.
Cholesky decomposition 89f., 423, 455, 668, 1038ff.
complex 41
computing $A^{-1} \cdot B$ 40
conjugate gradient method 77ff., 599, 1034
 cyclic tridiagonal 67, 1030
direct methods 26, 64, 1014, 1030
Fortran 90 vs. library routines 1016
Gauss-Jordan elimination 27ff., 1014
Gaussian elimination 33f., 1014ff.
Hilbert matrix 83
Hotelling’s method 49, 598
and integral equations 779ff., 783, 1325
iterative improvement 47ff., 195, 1022
iterative methods 26, 77ff., 1034
large sets of 23
least squares solution 53ff., 57f., 199f., 671, 1081, 1290
LU decomposition 34ff., 195, 386, 732, 783, 786, 801, 1016, 1022, 1325ff.
nonsingular 23
overdetermined 25f., 199, 670, 797
partitioned 70
QR decomposition 91f., 382, 386, 668, 1039ff., 1199
row vs. column elimination 31f.
Schulz’s method 49, 598
Sherman-Morrison formula 65ff., 83
singular 22, 53, 58, 199, 670
singular value decomposition (SVD) 51ff., 199f.,
670ff., 797, 1022, 1081, 1290
sparse 23, 43, 63ff., 732, 804, 1020ff., 1030
summary of tasks 25f.
Toeplitz 82, 85ff., 195, 1038
Vandermonde 82ff., 114, 1037, 1047
wavelet solution 597ff., 782
Woodbury formula 68ff., 83
see also Eigensystems
Linear congruential random number generator 267ff., 1142
choice of constants for 274ff.
 Linear constraints 423
Linear convergence 346, 393
Linear correlation (statistics) 630ff., 1276
 Linear dependency
constructing orthonormal basis 58, 94
of directions in $N$-dimensional space 409
in linear algebraic equations 22f.
Linear equations see Differential equations; Integral equations; Linear algebraic equations
Linear inversion method, constrained 799ff.
Linear prediction 557ff.
characteristic polynomial 559
coefficients 557ff., 1256
compared to maximum entropy method 558
compared with regularization 801
contrasted to polynomial extrapolation 560
related to optimal filtering 558
removal of bias in 563
stability 559f., 1257
Linear predictive coding (LPC) 563ff.
Linear programming 387, 423ff., 1216ff.
artificial variables 429
auxiliary objective function 430
basic variables 426
composite simplex algorithm 435
constraints 423
convergence criteria 432
decompose feasible vector 429
dual problem 435
equality constraints 423
feasible basis vector 426
feasible vector 424
fundamental theorem 426
inequality constraints 423
left-hand variables 426
nonbasic variables 426
normal form 426
 objective function 424
optimal feasible vector 424
pivot element 428f.
primal-dual algorithm 435
primal problem 435
reduction to normal form 429ff.
restricted normal form 426ff.
revised simplex method 435
right-hand variables 426
simplex method 402, 423ff., 431ff., 1216ff.
slack variables 429
tableau 427
vertex of simplex 426
Linear recurrence see Recurrence relation
Linear regression 655ff., 660ff., 1285ff. see also Fitting
Linear regularization 799ff.
LINPACK 26
Literal constant 937, 1361
Little-endian 293
Local extrapolation 709
Local extremum 387f., 437
Localization of roots see Bracketing
Logarithmic function 255
Lomb periodogram method of spectral analysis 569f., 1258f.
fast algorithm 574f., 1259
Loops 9f.
Lorentzian probability distribution 282, 696f.
Low-pass filter 551, 644f., 1283f.
Lower subscript 944
lower_triangle() utility function 989, 1007, 1200
LP coefficients see Linear prediction
LPC (linear predictive coding) 563ff.
LU decomposition 34ff., 47f., 51, 55, 64, 97, 374, 667, 732, 1016, 1022
for A−1 · B 40
backsubstitution 39, 1017
band diagonal matrix 43ff., 1020
complex equations 41f.
Croat's algorithm 36ff., 45, 1017
for integral equations 783, 786, 1325f.
for inverse iteration of eigenvectors 488
for inverse problems 801
for matrix determinant 41
for matrix inverse 40, 1016
for nonlinear sets of equations 374, 386, 1196
operation count 36, 39
outer product Gaussian elimination 1017
for Padé approximant 195, 1080
pivoting 37f., 1017
repeated backsubstitution 40, 46
solution of linear algebraic equations 40, 1017
solution of normal equations 667
for Tridiagonal matrix
Lucifer 290
M&R (Metcalfe and Reid) 935
M-estimates 694ff.
how to compute 697f.
local 695ff. see also Maximum likelihood estimate
Machine accuracy 19f., 881f., 1189, 1343
Macintosh, see Apple Macintosh
Machlly’s procedure 364, 371
Magic in MEM image restoration 814
in Padé approximant 195
Mantissa in floating point format 19, 882, 909, 1343
Marginals 624
Marquardt method (least squares fitting) 678ff., 816, 1292f.
Marsaglia shift register 1142, 1148ff.
Marsaglia, G. 1142, 1149
Matrix 23ff.
add vector to diagonal 1004, 1234, 1366, 1381
approximation of 58f., 598f.
band diagonal 42ff., 64, 1019
band triangular 64
banded 26, 454
bidirectional 52
block diagonal 64, 754
block triangular 64
block tridiagonal 64
bordered 64
classification polynomial 449, 469
Cholesky decomposition 89f., 423, 455, 668, 1038f.
column augmented 28, 1014
column 41
complex 41
condition number 53, 78
create unit matrix 1006, 1382
curvature 677
cyclic banded 64
cyclic tridiagonal 67, 1030
defective 450, 476, 489
derivatives see Hessian matrix; Jacobian determinant
design (fitting) 645, 665, 801, 1082
determinant of 25, 41
diagonal of sparse matrix 1033f.
diagonalization 452ff., 1225ff.
elementary row and column operations 28f.
finite differencing of partial differential equations 821ff.
gem diagonal 985, 1005, 1226ff., 1366, 1381f.
Hermitian 450, 454, 475
Hermitian conjugate 450
Hessian see Hessian matrix
hierarchically band diagonal 598
Hilbert 83
identity 25
ill-conditioned 53, 56, 114
indexed storage of 71f., 1030
and integral equations 779, 783, 1325
inverse 25, 27, 34, 40, 65ff., 70, 95ff., 1014, 1016f.
inverse, approximate 49
inverse by Hotelling’s method 49, 598
inverse by Schulte’s method 49, 598
inverse multiplied by a matrix 40
iteration for inverse 49, 598
Jacobi transformation 453, 456ff., 462, 1225f.
Jacobian 731, 1309
logical dimension 24
lower triangular 34f., 89, 781, 1016
lower triangular mask 1007, 1200, 1382
multiplication denoted by dot 23
multiplication, intrinsic function 949, 969, 1026, 1040, 1050, 1200, 1382
norm 50
nullity 53
nullspace 25, 53f., 449, 795
orthogonal 91, 450, 463ff., 587
orthogonal transformation 452, 463ff., 469, 1227
orthonormal basis 58, 94
outer product denoted by cross 66, 420
partitioning for determinant 70
partitioning for inverse 70
pattern multiply of sparse 74
physical dimension 24
positive definite 26, 89f., 668, 1038
QR decomposition 91f., 382, 386, 668, 1039, 1199
range 53
rank 53
residual 49
row and column indices 23
row vs. column operations 31f.
self-adjoint 450
set diagonal elements 1005, 1200, 1366, 1382
similarity transform 452ff., 456, 476, 478, 482
singular 53f., 58, 449
singular value decomposition 26, 51ff., 797
sparse 23, 63ff., 71, 598, 732, 754, 804, 1030ff.
special forms 26
splitting in relaxation method 856f.
spread 808
square root of 423, 455
symmetric 26, 89, 450, 454, 462ff., 668, 785, 1038, 1225, 1227
threshold multiply of sparse 74, 1031
Toeplitz 82, 85ff., 195, 1038
transpose() intrinsic function 950
transpose of sparse 73f., 1033
triangular 453
tridiagonal 26, 42f., 64, 109, 150, 453f., 462ff., 469ff., 488, 839f., 853, 861f., 1018f., 1227ff.
tridiagonal with fringes 822
unitary 450
updating 94, 382, 386, 1041, 1199
upper triangular 34f., 91, 1016
upper triangular mask 1006, 1226, 1305, 1382
Vandermonde 82ff., 114, 1037, 1047
see also Eigensystems
Matrix equations see Linear algebraic equations
Matterhorn 606
maxexponent() intrinsic function 1107
Maximization see Minimization
Maximum entropy method (MEM) 565ff., 1258
algorithms for image restoration 815f.
Bayesian 816f.
Cornwell-Evans algorithm 816
demystified 814
historic vs. Bayesian 816f.
image restoration 809ff.
intrinsic correlation function (ICF) model 817
for inverse problems 809ff.
operation count 567
see also Linear prediction
Maximum likelihood estimate (M-estimates) 690, 694ff.
and Bayes’ Theorem 811
chi-square test 690
defined 652
how to compute 697f.
mean absolute deviation 696, 698, 1294
relation to least squares 652
maxloc() intrinsic function 949, 992f., 1015
modified in Fortran 95 961
maxval() intrinsic function 945, 948, 961, 1016, 1273
Maxwell’s equations 825f.
Mean(s)
of distribution 604f., 608f., 1269
statistical differences between two 609ff., 1269f.
Mean absolute deviation of distribution 605, 696, 1294
related to median 698
Measurement errors 650
Median 320
calculating 333
doF distribution 605, 608f.
as L-estimate 694
role in robust straight line fitting 698
by selection 698, 1294
Median-of-three, in Quicksort 324
MEM see Maximum entropy method (MEM)
Memory leak 953, 956, 1071, 1327
Memory management 938, 941f., 953ff., 1327, 1336
merge construct 945, 950, 1099f.
for conditional scalar expression 1010, 1094f.
contrasted with where 1023
parallelization 1011
Merge-with-dummy-values idiom 1090
Merit function 650
in general linear least squares 665
for inverse problems 797
nonlinear models 675
for straight line fitting 656, 698
for straight line fitting, errors in both coordinates 660, 1286
Mesh-drift instability 834f.
Mesokurtic distribution 606
Metcalf, Michael 2/viii
see also M&R
Method of regularization 799ff.
Metropolis algorithm 437f., 1219
Microsoft 1/xxii, 2/xix
Microsoft Fortran PowerStation 2/viii
Midpoint method see Modified midpoint method; Semi-implicit midpoint rule
Mikado, or Town of Titipu 714
Miller’s algorithm 175, 228, 1106
MIMD machines (Multiple Instruction Multiple Data) 964, 985, 1071, 1084
Minimal solution of recurrence relation 174
Minimax polynomial 186, 197, 1076
Minimax rational function 198
Minimization 387ff.
along a ray 77, 376f., 389, 406ff., 412f., 415f., 418, 1195f., 1211, 1213
bracketing of minimum 390ff., 395, 1202ff.
Brent’s method 389, 395ff., 399, 660f., 1204ff., 1286
Broyden-Fletcher-Goldfarb-Shanno algorithm 390, 418ff., 1215
chi-square 653ff., 675ff., 1285, 1292
choice of methods 388f.
combinatorial 436f., 1219
conjugate gradient method 390, 413ff., 804, 815, 1210, 1214
convergence rate 393, 409
Davison-Fletcher-Powell algorithm 390, 418ff., 1215
degenerate 795
direction-set methods 389, 406ff., 1210ff.
downhill simplex method 389, 402ff., 444, 697f., 1208, 1222ff.
finding best-fit parameters 650
Fletcher-Reeves algorithm 390, 414ff., 1214
functional 795
global 387f., 443f., 650, 1219, 1222
globally convergent multidimensional 418, 1215
golden section search 390ff., 395, 1202ff.
multidimensional 388f., 402ff., 1208ff., 1214
in nonlinear model fitting 675f., 1292
Polak-Ribiere algorithm 389, 414ff., 1214
Powell’s method 389, 402, 406ff., 1210ff.
Powell's method 376, 390, 418ff., 1215
quasi-Newton methods 376, 390, 418ff., 1215
and root finding 375
scaling of variables 420
by searching smaller subspaces 815
steepest descent method 414, 804
termination criterion 392, 404
use in finding double roots 341
use for sparse linear systems 77ff.
using derivatives 389f., 399f., 1205f.
variable metric methods 390, 418ff., 1215
see also Linear programming
Minimum residual method, for sparse system 78
minloc() intrinsic function 949, 992f.
modified in Fortran 95 961
MINPACK 683
minval() intrinsic function 948, 961
MIPS 886
Missing data problem 569
Mississippi River 438f., 447
MMP (massively multiprocessor) machines 965ff., 974, 981, 984, 1016ff., 1021, 1045, 1226ff., 1250
Mode of distribution 605, 609
Modeling of data see Fitting
Model-trust region 386, 683
Modes, homogeneous, of recursive filters 554
Modified Bessel functions see Bessel functions
Modified Lentz’s method, for continued fractions 165
Modified midpoint method 716ff., 720, 1302f.
Modified moments 152
Modula-2 7
Modular arithmetic, without overflow 269, 271, 275
Modular programming 2/xiii, 7f., 956ff., 1209, 1293, 1296, 1346
MODULE facility 2/xiii, 936ff., 939f., 957, 1067, 1298, 1320, 1322, 1324, 1330, 1346
initializing random number generator 1144ff.
in nr.f90, 936, 941f., 1362, 1384ff.
in nttype.f90, 936f., 1361f.
in nrutil.f90, 936, 937, 1362, 1364ff.
sparse matrix 1031
undefined variables on exit 953, 1266
Module subprogram 940
modal() intrinsic function 946, 1156
Modulus of linear congruential generator 268
Moments of distribution 604ff., 1269
filter that preserves 645
modified problem of 151f.
problem of 83
and quadrature formulas 791, 1328
semi-invariants 608
Monic polynomial 142f.
Monotonicity constraint, in upwind differencing 837
Monte Carlo 155f., 267
adaptive 306ff., 1161f.
bootstrap method 686f.
comparison of sampling methods 309
exploration of binary tree 290
importance sampling 306f.
integration 124, 155ff., 295ff., 306ff., 1161
integration, recursive 314ff., 1164ff.
integration, using Sobol’ sequence 304
integration, VEGAS algorithm 309ff., 1161
and Kolmogorov-Smirnov statistic 622, 640
partial differential equations 824
quasi-random sequences in 299f.
quick and dirty 686f.
recursive 306ff., 314ff., 1161, 1164ff.
significance of Lomb periodogram 570
simulation of data 654, 684ff., 690
stratified sampling 308f., 314, 1164
Moon, calculate phases of 1f., 14f., 936, 1010f.
Mother functions 584
Mother Nature 684, 686
Moving average (MA) model 566
Moving window averaging 644
Mozart 9
MS 1/xxii, 2/xix
Muller’s method 364, 372
Multidimensional
confidence levels of fitting 688f.
data, use of binning 623
Fourier transform 515ff., 1241, 1246, 1251
Fourier transform, real data 519ff., 1248f.
initial value problems 844ff.
interpolation 116ff., 1049ff.
Kolmogorov-Smirnov test 640, 1281
least squares fitting 675
minimization 402ff., 406ff., 413ff., 1208ff., 1214f., 1222ff.
Monte Carlo integration 295ff., 306ff., 1161ff.
normal (Gaussian) distribution 690
optimization 388f.
partial differential equations 844ff.
search using quasi-random sequence 300
secant method 373, 382f., 1199f.
wavelet transform 595, 1267f.
Multigrid method 824, 862ff., 1334ff.
avoid SOR 866
boundary conditions 868f.
choice of operators 868
coarse-to-fine operator 864, 1337
course-grid correction 864f.
cycle 865
dual viewpoint 875
fine-to-coarse operator 864, 1337
full approximation storage (FAS) algorithm 874, 1339ff.
full multigrid method (FMG) 863, 868, 1334ff.
full weighting 867
Gauss-Seidel relaxation 865f., 1338
half weighting 867, 1337
importance of adjoint operator 867
interpolation operator 864, 1337
line relaxation 866
local truncation error 875
Newton’s rule 874, 876, 1339, 1341
nonlinear equations 874ff., 1339ff.
nonlinear Gauss-Seidel relaxation 876, 1341
odd-even ordering 866, 869, 1338
operation count 862
prolongation operator 864, 1337
recursive nature 865, 1009, 1336
relative truncation error 875
relaxation as smoothing operator 865
restriction operator 864, 1337
speeding up FMG algorithm 873
stopping criterion 875f.
straight injection 867
symbol of operator 866f.
use of Richardson extrapolation 869
V-cycle 865, 1336
W-cycle 865, 1336
zebra relaxation 866
Multiple precision arithmetic 906ff., 1352ff.
Multiple roots 341, 362
Multiplication, complex 171
Multiplication, multiple precision 907, 909, 1353f.
Multiplier of linear congruential generator 268
Multistep and multivalue methods (ODEs) 740ff.
see also Differential Equations; Predictor-corrector methods
Multivariate normal distribution 690
Murphy’s Law 407
Musical scores 5f.

NAG 1/xxiii, 2/xx, 26, 64, 205, 454
Fortran 90 compiler 2/viii, 2/xiv
Named constant 940
initialization 1012
for Numerical Recipes 1361
Named control structure 959, 1219, 1305
National Science Foundation (U.S.) 1/xvii, 1/xix, 2/ix
Natural cubic spline 109, 1044f.
Navier-Stokes equation 830f.
nearest() intrinsic function 952, 1146
Needle, eye of (minimization) 952, 1146
Needle, eye of (minimization) 952, 1146
Negation, multiple precision 907, 1353f.
Negentropy 811, 896
Nelder-Mead minimization method 389, 402, 1208
Nested iteration 868
Neumann boundary conditions 820, 840, 851, 858
Neutrino 640
Neville’s algorithm 102ff., 105, 134, 182, 1043
Newton-Cotes formulas 125ff., 140
Newton-Raphson method see Newton’s rule
Newton’s rule 143f., 180, 341, 355ff., 136, 364, 469, 1059, 1189
with backtracking 376, 1196
cautions on use of numerical derivatives 356ff.
fractal domain of convergence 360f.
globally convergent multidimensional 373, 376ff., 382, 749f., 752, 1196, 1199, 1341f.
for matrix inverse 49, 598
in multidimensions 370, 372ff., 749f., 752, 754, 1194ff., 1314ff.
in nonlinear multigrid 874, 876, 1339, 1341
nonlinear Volterra equations 787
for reciprocal of number 911, 1355
safe 359, 1190
scaling of variables 381
singular Jacobian 386
solving stiff ODEs 740
for square root of number 912, 1356
Niederreiter sequence 300
NL2SOL 683
Noise
bursty 889
effect on maximum entropy method 567
equivalent bandwidth 548
fitting data which contains 647, 650
model, for optimal filtering 541
Nominal variable (statistics) 623
Nonexpansive projection operator 805
Non-interfering directions see Conjugate directions
Nonlinear eigenvalue problems 455
Nonlinear elliptic equations, multigrid method 874ff., 1339ff.
Nonlinear equations, in MEM inverse problems 813
Nonlinear equations, roots of 340ff.
Nonlinear instability 831
Nonlinear integral equations 781, 787
Nonlinear programming 436
Nonnegativity constraints 423
Nonparametric statistics 633ff., 1277ff.
Nonpolynomial complete (NP-complete) 438
Norm, of matrix 50
Normal (Gaussian) distribution 267, 652, 682, 798, 1294
central limit theorem 652f.
deviates from 279f., 571, 1152
kurtosis of 607
multivariate 690
semi-invariants of 608
tails compared to Poisson 653
two-dimensional (binormal) 631
Normal equations (fitting) 26, 645, 666ff., 795, 800, 1288
often are singular 670
Normalization
of Bessel functions 175
of floating-point representation 19, 882, 1434
of functions 142, 765
of modified Bessel functions 232
not() intrinsic function 951
Notch filter 551, 555f.
NP-complete problem 438
nr.f90 (module file) 936, 1362, 1339ff.
nrerror() utility function 989, 995
ntype.f90 (module file) 936f.
named constants 1361
nrutil.f90 (module file) 936, 1070, 1362, 1364ff.
table of contents 1364
Null hypothesis 603
nullify statement 953f., 1070, 1302
Nullity 53
Nullspace 25, 53f., 449, 795
Number-theoretic transforms 503f.
Numeric inquiry functions 2/xiv, 1107, 1231, 1434
Numerical derivatives 180ff., 645, 1075
Numerical integration see Quadrature
Numerical intrinsic functions 946, 951f.
Numerical Recipes
compatibility with First Edition 4
Example Book 3
Fortran 90 types 936ff., 1361
how to get programs 1/xx, 2/xvii
how to report bugs 1/iv, 2/iv
interface blocks (Fortran 90) 937, 941f., 1084, 134ff.
no warranty on 1/xx, 2/xvii
plan of two-volume edition 1/xiii
table of dependencies 921ff., 1434ff.
as trademark 1/xiiii, 2/xx
utility functions (Fortran 90) 936f., 945, 968, 970, 972ff., 977, 984, 987ff., 1015, 1071f., 1361ff.
Numerical Recipes Software 1/xx, 1/xxii, 2/xviii
, 2/xviii,
address and fax number 1/iv, 1/xxii, 2/iv, 2/xix
Web home page 1/xx, 2/xvii
Nyquist frequency 494ff., 520, 543, 545, 560ff.
Nyström method 782f., 789, 1325
product version 789, 1331
Object extensibility 8
Objective function 424
Object-oriented programming 2/xvi, 2, 8
Oblateness parameter 764
Obsolete features see Fortran, Obsolescent features
Octal constant, initialization 959
Odd-even ordering
allows parallelization 1333
in Gauss-Seidel relaxation 866, 869, 1338
in successive over-relaxation (SOR) 859, 1332
Odd parity 888
OEM information 1/xxii
One-sided power spectral density 492
ONLY option, for USE statement 941, 957, 1067
Operation count
balancing 476
Bessel function evaluation 228
bisection method 346
Cholesky decomposition 90
coefficients of interpolating polynomial 114f.
complex multiplication 97
cubic spline interpolation 109
evaluating polynomial 168
fast Fourier transform (FFT) 498
Gauss-Jordan elimination 34, 39
Gaussian elimination 34
Givens reduction 463
Householder reduction 467
interpolation 100
inverse iteration 488
iterative improvement 48
Jacobi transformation 460
Kendall’s tau 637
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear congruential generator</td>
<td>268</td>
</tr>
<tr>
<td>LU decomposition</td>
<td>36, 39</td>
</tr>
<tr>
<td>Matrix inversion</td>
<td>97</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>96</td>
</tr>
<tr>
<td>Maximum entropy method</td>
<td>567</td>
</tr>
<tr>
<td>Multidimensional minimization</td>
<td>413f.</td>
</tr>
<tr>
<td>Multigrid method</td>
<td>862</td>
</tr>
<tr>
<td>Multiplication</td>
<td>909</td>
</tr>
<tr>
<td>Polynomial evaluation</td>
<td>97f., 168</td>
</tr>
<tr>
<td>QL method</td>
<td>470, 473</td>
</tr>
<tr>
<td>QR decomposition</td>
<td>92</td>
</tr>
<tr>
<td>QR method for Hessenberg matrices</td>
<td>484</td>
</tr>
<tr>
<td>Reduction to Hessenberg form</td>
<td>479</td>
</tr>
<tr>
<td>Selection by partitioning</td>
<td>333</td>
</tr>
<tr>
<td>Sorting</td>
<td>320ff.</td>
</tr>
<tr>
<td>Spearman rank-order coefficient</td>
<td>638</td>
</tr>
<tr>
<td>Toeplitz matrix</td>
<td>83</td>
</tr>
<tr>
<td>Vandermonde matrix</td>
<td>83</td>
</tr>
<tr>
<td>Operator overloading</td>
<td>2/xiif., 7</td>
</tr>
<tr>
<td>Operator splitting</td>
<td>823, 847ff., 861</td>
</tr>
<tr>
<td>Operator, user-defined</td>
<td>2/xii</td>
</tr>
<tr>
<td>Optimal feasible vector</td>
<td>424</td>
</tr>
<tr>
<td>Optimal (Wiener) filtering</td>
<td>535, 539ff., 558, 644</td>
</tr>
<tr>
<td>Optimized with regularization</td>
<td>801</td>
</tr>
<tr>
<td>Optimization of code</td>
<td>2/xiii</td>
</tr>
<tr>
<td>Optimization see Minimization</td>
<td></td>
</tr>
<tr>
<td>Orthonormal see Orthonormal functions; Orthonormal polynomials</td>
<td></td>
</tr>
<tr>
<td>Orthogonal transformation</td>
<td>452, 463ff., 469, 584, 1227</td>
</tr>
<tr>
<td>Orthonormal basis, constructing</td>
<td>58, 94, 1039</td>
</tr>
<tr>
<td>Orthonormal functions</td>
<td>142, 246</td>
</tr>
<tr>
<td>Orthonormal polynomials</td>
<td></td>
</tr>
<tr>
<td>Chebyshev 144, 184ff., 1076ff.</td>
<td></td>
</tr>
<tr>
<td>construct for arbitrary weight</td>
<td>151ff., 1064</td>
</tr>
<tr>
<td>in Gauss-Hermite integration</td>
<td>147, 1062</td>
</tr>
<tr>
<td>and Gaussian quadrature</td>
<td>142, 1009, 1061</td>
</tr>
<tr>
<td>Gaussian weights from recurrence</td>
<td>150, 1064</td>
</tr>
<tr>
<td>Hermite 144, 1062</td>
<td></td>
</tr>
<tr>
<td>Jacobi 144, 1063</td>
<td></td>
</tr>
<tr>
<td>Laguerre 144, 1060</td>
<td></td>
</tr>
<tr>
<td>Legendre 144, 1059</td>
<td></td>
</tr>
<tr>
<td>weight function log_2</td>
<td>153</td>
</tr>
<tr>
<td>Orthonormality</td>
<td>51, 142, 463</td>
</tr>
<tr>
<td>Outer product Gaussian elimination</td>
<td>1017</td>
</tr>
<tr>
<td>Outer product of matrices (denoted by cross)</td>
<td>66, 420, 949, 969f., 989, 1000ff., 1017, 1026, 1040, 1076, 1200, 1216, 1275</td>
</tr>
<tr>
<td>outerand() utility function</td>
<td>989, 1002, 1015</td>
</tr>
<tr>
<td>outerdiff() utility function</td>
<td>989, 1001</td>
</tr>
<tr>
<td>outerdiv() utility function</td>
<td>989, 1001</td>
</tr>
<tr>
<td>outerprod() utility function</td>
<td>970, 989, 1000, 1017, 1026, 1040, 1076, 1200, 1216, 1275</td>
</tr>
<tr>
<td>outersum() utility function</td>
<td>989, 1001</td>
</tr>
<tr>
<td>Outgoing wave boundary conditions</td>
<td>820</td>
</tr>
<tr>
<td>Outlier</td>
<td>605, 653, 656, 694, 697</td>
</tr>
<tr>
<td>see also Robust estimation</td>
<td></td>
</tr>
<tr>
<td>Overcorrection</td>
<td>857</td>
</tr>
<tr>
<td>Overflow</td>
<td>882, 1343</td>
</tr>
<tr>
<td>how to avoid in modulo multiplication</td>
<td>269</td>
</tr>
<tr>
<td>in complex arithmetic</td>
<td>171</td>
</tr>
<tr>
<td>Overlap-add and overlap-save methods</td>
<td>536f.</td>
</tr>
<tr>
<td>Overloading</td>
<td></td>
</tr>
<tr>
<td>operator 2/xiif.</td>
<td></td>
</tr>
<tr>
<td>procedures 940, 1015, 1083, 1094, 1096</td>
<td></td>
</tr>
<tr>
<td>Overrelaxation parameter</td>
<td>857, 1332</td>
</tr>
<tr>
<td>choice of 858</td>
<td></td>
</tr>
<tr>
<td>Pack() intrinsic function</td>
<td>945, 950, 964, 991, 1031</td>
</tr>
<tr>
<td>communication bottleneck</td>
<td>969</td>
</tr>
<tr>
<td>for index table</td>
<td>1176</td>
</tr>
<tr>
<td>for partition-exchange</td>
<td>1170</td>
</tr>
<tr>
<td>for selection</td>
<td>1178</td>
</tr>
<tr>
<td>for selective evaluation</td>
<td>1087</td>
</tr>
<tr>
<td>Pack-unpack idiom</td>
<td>1087, 1134, 1153</td>
</tr>
<tr>
<td>Padé approximant</td>
<td>194ff., 1080f.</td>
</tr>
<tr>
<td>Padé approximation</td>
<td>105</td>
</tr>
<tr>
<td>Parabolic interpolation</td>
<td>395, 1204</td>
</tr>
<tr>
<td>Parabolic partial differential equations</td>
<td>818, 838ff.</td>
</tr>
<tr>
<td>Parallel axis theorem</td>
<td>308</td>
</tr>
<tr>
<td>Parallel programming</td>
<td>2/vx, 941, 958ff., 962ff., 965f., 968f., 987</td>
</tr>
<tr>
<td>array operations</td>
<td>964f.</td>
</tr>
<tr>
<td>array ranking</td>
<td>1278f.</td>
</tr>
<tr>
<td>band diagonal linear equations</td>
<td>1021</td>
</tr>
<tr>
<td>Bessel functions</td>
<td>1107ff.</td>
</tr>
<tr>
<td>broadcasts 965ff.</td>
<td></td>
</tr>
<tr>
<td>C and C++</td>
<td>2/viii</td>
</tr>
<tr>
<td>communication costs</td>
<td>969, 981, 1250</td>
</tr>
<tr>
<td>counting do-loops</td>
<td>1015</td>
</tr>
<tr>
<td>cyclic reduction</td>
<td>974</td>
</tr>
<tr>
<td>deflation 977ff.</td>
<td></td>
</tr>
<tr>
<td>design matrix</td>
<td>1082</td>
</tr>
<tr>
<td>dimensional expansion</td>
<td>965ff.</td>
</tr>
<tr>
<td>eigensystems</td>
<td>1226, 1229f.</td>
</tr>
<tr>
<td>fast Fourier transform (FFT)</td>
<td>981, 1235ff., 1250</td>
</tr>
<tr>
<td>in Fortran 90</td>
<td>963ff.</td>
</tr>
<tr>
<td>Fortran 90 tricks</td>
<td>1099, 1274, 1278, 1280</td>
</tr>
<tr>
<td>function evaluation</td>
<td>986, 1009, 1084ff., 1087, 1090, 1102, 1128, 1134</td>
</tr>
<tr>
<td>Gaussian quadrature</td>
<td>1009, 1061</td>
</tr>
<tr>
<td>geometric progressions</td>
<td>972</td>
</tr>
<tr>
<td>index loss</td>
<td>967f., 1038</td>
</tr>
<tr>
<td>index table</td>
<td>1176f.</td>
</tr>
<tr>
<td>interprocessor communication</td>
<td>981</td>
</tr>
<tr>
<td>Kendall’s tau</td>
<td>1280</td>
</tr>
<tr>
<td>linear algebra</td>
<td>969f., 1000ff., 1018f., 1026, 1040, 1200, 1326</td>
</tr>
<tr>
<td>linear recurrence</td>
<td>973ff., 1073ff.</td>
</tr>
<tr>
<td>logo 2/viii, 1009</td>
<td></td>
</tr>
<tr>
<td>masks 967f., 1006f., 1038, 1102, 1200, 1226, 1305, 1333ff., 1368, 1378, 1382</td>
<td></td>
</tr>
<tr>
<td>merge statement</td>
<td>1010</td>
</tr>
</tbody>
</table>
MIMD (multiple instruction, multiple data) 964, 985f., 1084
MMP (massively multiprocessor) machines 965ff., 974, 984, 1016ff., 1226ff., 1250
numt1.f90 (module file) 1364ff.
odd-even ordering 1333
one-dimensional FFT 982f.
parallel note icon 1009
partial differential equations 1333
in-place selection 1178f.
polynomial coefficients from roots 980
polynomial evaluation 972f., 977, 998
random numbers 1009, 1141ff.
recursive doubling 973f., 976f., 979, 988, 999, 1071ff.
scatter-with-combine 984, 1002f., 1032f.
second order recurrence 974f., 1074
SIMD (Single Instruction Multiple Data) 964, 985f., 1009, 1084f.
singular value decomposition (SVD) 1026
sorting 1167ff., 1171, 1176f.
special functions 1009
SSP (small-scale parallel) machines 965ff., 984, 1010ff., 1016ff., 1059f., 1226ff., 1250
subvector scaling 972, 974, 996, 1000
successive over-relaxation (SOR) 1333
supercomputers 2/viii, 962
SVD algorithm 1026
synthetic division 977ff., 999, 1048, 1071f., 1079, 1079f., 1192
tridiagonal systems 975f., 1018, 1229f.
vector reduction 972f., 977, 998
vs. serial programming 965, 987
PARAMETER attribute 1012
Parameters in fitting function 651, 684ff.
Parity bit 888
Park and Miller minimal standard random generator 269, 1142
Parkinson’s Law 328
Parseval’s Theorem 492, 544
discrete form 498
Partial differential equations 818ff., 1332ff.
advective equation 826
alternating-direction implicit method (ADI) 847, 861f.
amplification factor 828, 834
analyze/evaluate/operate package 824
artificial viscosity 831, 837
biconjugate gradient method 824
boundary conditions 819f.
boundary value problems 819, 848
Cauchy problem 818f.
cautions on high-order methods 844f.
Cayley’s form 844
characteristics 818
Chebyshev acceleration 859f., 1332
classification of 818f.
comparison of rapid methods 854
conejuate gradient method 824
Courant condition 829, 832ff., 836
Courant condition (multidimensional) 846
Crank-Nicolson method 840, 842, 844, 846
cylic reduction (CR) method 848f., 852f.
diffusion equation 818, 838ff., 846, 855
Dirichlet boundary conditions 508, 820, 840, 850, 856, 858
eigenfunction, defined 818
error, varieties of 831ff.
explicit vs. implicit differencing 827
FAOR method 854
finite difference method 821f.
finite element methods 824
flux-conservative initial value problems 825f.
forward Euler differencing 826f.
forward Time Centered Space (FTCS) 827ff., 839ff., 843, 855
Fourier analysis and cyclic reduction (FAOR) 848ff., 854
Gauss-Seidel method (relaxation) 855, 864ff., 876, 1338, 1341
Godunov’s method 837
Helmholtz equation 852
hyperbolic 818, 825f.
imPLICIT differencing 840
incomplete Cholesky conjugate gradient method (ICCG) 824
inhomogeneous boundary conditions 850ff.
initial value problems 818f.
initial value problems, recommendations on 838ff.
Jacobi’s method (relaxation) 855ff., 864
Laplace’s equation 818
Lax method 824f., 836, 845f.
Lax method (multidimensional) 845f.
matrix methods 824
mesh-drift instability 834f.
Monte Carlo methods 824
Multidimensional initial value problems 844ff.
multigrid method 824, 862ff., 1009, 1334ff.
Neumann boundary conditions 508, 820, 840, 851, 858
nonlinear diffusion equation 842
nonlinear instability 831
numerical dissipation or viscosity 830
operator splitting 823, 847f., 861
outgoing wave boundary conditions 820
parabolic 818, 838ff.
parallel computing 1333
periodic boundary conditions 850, 858
piecewise parabolic method (PPM) 837
Poisson equation 818, 852
rapid (Fourier) methods 808ff., 824, 848ff.
relaxation methods 823, 854ff., 1332f.
Schrödinger equation 842ff.
second-order accuracy 833ff., 840
shock 831, 837
sparse matrices from 64
spectral methods 825
spectral radius 856ff., 862
stability vs. accuracy 830
stability vs. efficiency 821
staggered grids 513, 852
staggered leapfrog method 833f.
strongly implicit procedure 824
successive over-relaxation (SOR) 857ff., 862, 866, 1332f.
time splitting 847ff., 861
two-step Lax-Wendroff method 835ff.
upwind differencing 832ff., 837
variational methods 824
varieties of error 831ff.
von Neumann stability analysis 827ff., 830, 833ff., 840
wave equation 818, 825f.
see also Elliptic partial differential equations; Finite difference equations (FDEs)

Partial pivoting 29
Partition-exchange 323, 333
and pack() intrinsic function 1170
Partitioned matrix, inverse of 70
Party tricks 95ff., 168
Parzen window 547
Pascal, Numerical Recipes in 2/x, 2/xvii, 1
Pass-the-buck idiom 1102, 1128
Path integration, for function evaluation 201ff., 263, 1138
Pattern multiply of sparse matrices 74
PBCG (preconditioned biconjugate gradient method) 78ff., 824
PC methods see Predictor-corrector methods
PCGPACK 71
PDEs see Partial differential equations
Pearson’s χ² 606ff., 1268
PECE method 741
Pentagon, symmetries of 895
Percentile 320
Period of linear congruential generator 268
Periodic boundary conditions 850, 858
Periodogram 543ff., 566, 1258ff.
Lomb’s normalized 569ff., 574ff., 1258ff.
variance of 544ff.
Perl (programming language) 1/xvi
Perlin (programming language) 1/xvi
Perron’s theorem, for convergence of recurrence relations 174ff.
Perturbation methods for matrix inversion 65ff.
Phase error 831
Phase-locked loop 700
Phi statistic 625
Pi, computation of 906ff., 1352ff., 1357f.
Piecewise parabolic method (PPM) 837
Pincherle’s theorem 175
Pivot element 29, 33, 757
in linear programming 428f.
Pivoting 27, 29ff., 46, 66, 90, 1014
full 29, 1014
implicit 30, 38, 1014, 1017
in LU decomposition 37f., 1017
partial 29, 33, 37f., 1017
and QR decomposition 92
in reduction to Hessenberg form 478
in relaxation method 757
as row and column operations 32
for tridiagonal systems 43
Pixel 519, 596, 803, 811
PL/I 2/x
Planck’s constant 842
Plane rotation see Givens reduction; Jacobi transformation (or rotation)
Platykurtic distribution 606
Plotting of functions 342, 1182f.
PDOCS (projection onto convex sets) 805
Poetry 5f.
Pointer (Fortran 90) 2/xiii, 938f., 944f., 953ff., 1197, 1212, 1266
as alias 939, 944f., 1286, 1333
allocating an array 941
allocating storage for derived type 955
for array of arrays 956, 1336
array of, forbidden 956, 1337
associated with target 938f., 944f., 952f., 1197
in Fortran 95 961
to function, forbidden 1067, 1210
initialization to null 2/xvi, 961
returning array of unknown size 955f., 1184, 1259, 1261, 1327
undefined status 952f., 961, 1070, 1266, 1302
Poisson equation 519, 818, 852
Poisson probability function cumulative 214
deviates from 281, 283ff., 571, 1154
semi-invariants of 608
tails compared to Gaussian 653
Poisson process 278, 282ff., 1153
Polak-Ribiere algorithm 390, 414ff., 1214
Poles see Complex plane, poles in
Polishing of roots 356, 363ff., 370ff., 1193
poly() utility function 973, 977, 989, 998, 1072, 1096, 1192, 1258, 1284
Polynmorphism 8
Polynomial interpolation 99, 102ff., 1043
Aitken’s algorithm 102
in Bulirsch-Stoer method 724, 726, 1305
coefficients for 113ff., 1047f.
Lagrange’s formula 84, 102f.
multidimensional 116ff., 1049ff.
Neville’s algorithm 102f., 1134, 182, 1043
pathology in determining coefficients for 116
in predictor-corrector method 740
smoothing filters 645
see also Interpolation
Polynomials 167ff.
algebraic manipulations 169, 1072
approximate roots of Hermite polynomials 1062
approximate roots of Jacobi polynomials 1064
approximate roots of Laguerre polynomials 1061
approximating modified Bessel functions 230
approximation from Chebyshev coefficients 191, 1078f.
AUTODIN-II 890
CTT 889f.
characteristic 368, 1193
characteristic, for digital filters 554, 559, 1257
<table>
<thead>
<tr>
<th>Characteristic, for eigenvalues of matrix</th>
<th>449, 469</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chebyshev</td>
<td>184ff., 1076ff.</td>
</tr>
<tr>
<td>coefficients from roots</td>
<td>980</td>
</tr>
<tr>
<td>CRC-16</td>
<td>890</td>
</tr>
<tr>
<td>cumulants of</td>
<td>977, 999, 1071ff., 1192, 1365, 1378f.</td>
</tr>
<tr>
<td>deflation</td>
<td>362ff., 370f., 977</td>
</tr>
<tr>
<td>derivatives of</td>
<td>167, 978, 1071</td>
</tr>
<tr>
<td>division</td>
<td>84, 169, 362, 370, 977, 1072</td>
</tr>
<tr>
<td>evaluation of</td>
<td>167, 972, 977, 998f., 1071, 1258, 1365, 1376ff.</td>
</tr>
<tr>
<td>extrapolation in Bulirsch-Stoer method</td>
<td>724, 726, 1305f.</td>
</tr>
<tr>
<td>extrapolation in Romberg integration</td>
<td>134</td>
</tr>
<tr>
<td>fitting</td>
<td>83, 114, 191, 645, 665, 674, 1078f., 1291</td>
</tr>
<tr>
<td>generator for CRC</td>
<td>889</td>
</tr>
<tr>
<td>ill-conditioned</td>
<td>362</td>
</tr>
<tr>
<td>masked evaluation of</td>
<td>1378</td>
</tr>
<tr>
<td>matrix method for roots</td>
<td>368, 1193</td>
</tr>
<tr>
<td>minimax</td>
<td>186, 198, 1076</td>
</tr>
<tr>
<td>monic</td>
<td>142f.</td>
</tr>
<tr>
<td>multiplication</td>
<td>169</td>
</tr>
<tr>
<td>operation count for</td>
<td>168</td>
</tr>
<tr>
<td>orthonormal</td>
<td>142, 184, 1009, 1061</td>
</tr>
<tr>
<td>parallel operations on</td>
<td>977ff., 998f., 1071f., 1192</td>
</tr>
<tr>
<td>primitive modulo 2</td>
<td>287ff., 301f., 889</td>
</tr>
<tr>
<td>roots of</td>
<td>178ff., 362ff., 368, 1191ff.</td>
</tr>
<tr>
<td>shifting of</td>
<td>192f., 978, 1079</td>
</tr>
<tr>
<td>stopping criterion in root finding</td>
<td>366</td>
</tr>
<tr>
<td>polyterm() utility function</td>
<td>974, 977, 989, 999, 1071f., 1192</td>
</tr>
<tr>
<td>Port, serial data</td>
<td>892</td>
</tr>
<tr>
<td>Portability</td>
<td>3, 963</td>
</tr>
<tr>
<td>Portable random number generator</td>
<td>see Random number generator</td>
</tr>
<tr>
<td>Positive definite matrix, testing for</td>
<td>90</td>
</tr>
<tr>
<td>Positivity constraints</td>
<td>423</td>
</tr>
<tr>
<td>Postal Service (U.S.), barcode</td>
<td>894</td>
</tr>
<tr>
<td>PostScript</td>
<td>1/xvi, 1/xxiii, 2/xx</td>
</tr>
<tr>
<td>Powell’s method</td>
<td>389, 402, 406ff., 1210ff.</td>
</tr>
<tr>
<td>Power (in a signal)</td>
<td>492f.</td>
</tr>
<tr>
<td>Power series</td>
<td>159ff., 167, 195</td>
</tr>
<tr>
<td>economization of</td>
<td>192f., 1061, 1080</td>
</tr>
<tr>
<td>Padé approximant of</td>
<td>194ff., 1080f.</td>
</tr>
<tr>
<td>Power spectral density</td>
<td>see Fourier transform; Spectral density</td>
</tr>
<tr>
<td>Power spectrum estimation</td>
<td>see Fourier transform; Spectral density</td>
</tr>
<tr>
<td>PowerStation, Microsoft Fortran</td>
<td>2/xix</td>
</tr>
<tr>
<td>PPM (piecewise parabolic method)</td>
<td>837</td>
</tr>
<tr>
<td>Precision</td>
<td>converting to double</td>
</tr>
<tr>
<td>floating point</td>
<td>882, 937, 1343, 1361ff.</td>
</tr>
<tr>
<td>multiple</td>
<td>906ff., 1352ff., 1362</td>
</tr>
<tr>
<td>Preconditioned biconjugate gradient method (PBCG)</td>
<td>78f.</td>
</tr>
<tr>
<td>Preconditioning, in conjugate gradient methods</td>
<td>824</td>
</tr>
<tr>
<td>Predictor-corrector methods</td>
<td>702, 730, 740ff.</td>
</tr>
<tr>
<td>Adams-Bashforth-Moulton schemes</td>
<td>741</td>
</tr>
<tr>
<td>adaptive order methods</td>
<td>744</td>
</tr>
<tr>
<td>compared to other methods</td>
<td>740</td>
</tr>
<tr>
<td>fallacy of multiple correction</td>
<td>741</td>
</tr>
<tr>
<td>with fixed number of iterations</td>
<td>741</td>
</tr>
<tr>
<td>functional iteration vs. Newton’s rule</td>
<td>742</td>
</tr>
<tr>
<td>multivalue compared with multistep</td>
<td>742ff.</td>
</tr>
<tr>
<td>starting and stopping</td>
<td>742, 744</td>
</tr>
<tr>
<td>stepsize control</td>
<td>742f.</td>
</tr>
<tr>
<td>present() intrinsic function</td>
<td>952</td>
</tr>
<tr>
<td>Prime numbers</td>
<td>915</td>
</tr>
<tr>
<td>Primitive polynomials modulo 2</td>
<td>287ff., 301f., 889</td>
</tr>
<tr>
<td>Principal directions</td>
<td>408f., 1210</td>
</tr>
<tr>
<td>Principal solution, of inverse problem</td>
<td>797</td>
</tr>
<tr>
<td>PRIVATE attribute</td>
<td>957, 1067</td>
</tr>
<tr>
<td>Prize, $1000 offered</td>
<td>272, 1141, 1150f.</td>
</tr>
<tr>
<td>Probability</td>
<td>see Random number generator; Statistical tests</td>
</tr>
<tr>
<td>Probability density, change of variables in</td>
<td>278f.</td>
</tr>
<tr>
<td>Procedure</td>
<td>see Program(s); Subprogram</td>
</tr>
<tr>
<td>Process loss</td>
<td>548</td>
</tr>
<tr>
<td>product() intrinsic function</td>
<td>948</td>
</tr>
<tr>
<td>Product Nystrom method</td>
<td>789, 1331</td>
</tr>
<tr>
<td>Program(s)</td>
<td>as black boxes</td>
</tr>
<tr>
<td>dependencies</td>
<td>921ff., 1434ff.</td>
</tr>
<tr>
<td>encapsulation</td>
<td>7</td>
</tr>
<tr>
<td>interfaces</td>
<td>2, 8</td>
</tr>
<tr>
<td>modularization</td>
<td>7f.</td>
</tr>
<tr>
<td>organization</td>
<td>5ff.</td>
</tr>
<tr>
<td>type declarations</td>
<td>2</td>
</tr>
<tr>
<td>typography of</td>
<td>2f., 12, 937</td>
</tr>
<tr>
<td>validation</td>
<td>3f.</td>
</tr>
<tr>
<td>Programming, serial vs. parallel</td>
<td>965, 987</td>
</tr>
<tr>
<td>Projection onto convex sets (POCS)</td>
<td>805</td>
</tr>
<tr>
<td>Projection operator, nonexpansive</td>
<td>805</td>
</tr>
<tr>
<td>Prolongation operator</td>
<td>864, 1337</td>
</tr>
<tr>
<td>Protocol, for communications</td>
<td>888</td>
</tr>
<tr>
<td>PSD (power spectral density)</td>
<td>see Fourier transform; Spectral density</td>
</tr>
<tr>
<td>Pseudo-random numbers</td>
<td>266ff., 1141ff.</td>
</tr>
<tr>
<td>PUBLIC attribute</td>
<td>957, 1067</td>
</tr>
<tr>
<td>Puns, particularly bad</td>
<td>167, 744, 747</td>
</tr>
<tr>
<td>PURE attribute</td>
<td>2/xv, 960f., 964, 986</td>
</tr>
<tr>
<td>put_diag() utility function</td>
<td>985, 990, 1005, 1200</td>
</tr>
<tr>
<td>Pyramidal algorithm</td>
<td>586, 1264</td>
</tr>
<tr>
<td>Pythagoreans</td>
<td>392</td>
</tr>
</tbody>
</table>

**Q**: see Eigensystems

**QR**: see Eigensystems

**QR decomposition**: 91f., 382, 386, 1039f., 1199

**backsolution**: 92, 1040
**and least squares**: 668
**operation count**: 92
**pivoting**: 92
**updating**: 94, 382, 386, 1041, 1199
**use for orthonormal basis**: 58, 94

**Quadratic convergence**: 49, 256, 351, 356, 409f., 419, 906
**equations**: 20, 178, 391, 457
Index to Volumes 1 and 2

1477

interpolation 353, 364
programming 436
Quadrature 123ff., 1052ff.
  adaptive 123, 190, 788
  alternative extended Simpson’s rule 128
  arbitrary weight function 151ff., 789,
    1064, 1328
  automatic 154
  Bode’s rule 126
  change of variable in 137ff., 788, 1056ff.
  by Chebyshev fitting 124, 189, 1078
  classical formulas for 124ff.
  Clenshaw-Curtis 124, 190, 512f.
  closed formulas 125, 127f.
  and computer science 881
  by cubic splines 124
  error estimate in solution 784
  extended midpoint rule 129f., 135, 1054f.
  extended rules 127ff., 134f., 786, 788ff.,
    1326, 1328
  extended Simpson’s rule 128
  Fourier integrals 577ff., 1261ff.
  Fourier integrals, infinite range 583
  Gauss-Chebyshev 144, 512f.
  Gauss-Hermite 144, 789, 1062
  Gauss-Jacobi 144, 1063
  Gauss-Kronrod 154
  Gauss-Laguerre 144, 789, 1060
  Gauss-Legendre 144, 783, 789, 1059,
    1325
  Gauss-Lobatto 154, 190, 512
  Gauss-Radau 154
  Gaussian integration 127, 140ff., 781,
    783, 788f., 1009, 1059f., 1325, 1328f.
  Gaussian integration, nonclassical weight
    function 151ff., 789f., 1064f., 1328f.
  for improper integrals 135ff., 789, 1055,
    1328
  for integral equations 781f., 786, 1325ff.
  Monte Carlo 124, 155ff., 295ff., 306ff.,
    1052ff., 1326ff.
  multidimensional 124, 155ff., 1052, 1065ff.
  multidimensional, by recursion 1052, 1065
  Newton-Cotes formulas 125ff., 140
    open formulas 125ff., 129f., 135
    related to differential equations 123
  Romberg integration 124, 134f., 137, 182,
    717, 788, 1054f., 1065, 1067
  semi-open formulas 130
  Simpson’s rule 126, 133, 136f., 583, 782,
    788ff., 1053
  Simpson’s three-eighths rule 126, 789f.
    singularity removal 137ff., 788, 1057ff.,
      1328ff.
    singularity removal, worked example 792,
      1328ff.
  trapezoidal rule 125, 127, 130ff., 134f.,
    579, 583, 782, 786, 1052ff., 1326f.
  using FFTs 124
  weight function log x 153
  see also Integration of functions
Quadrature mirror filter 585, 593
Quantum mechanics, Uncertainty Principle 600
Quartile value 320
Quasi-Newton methods for minimization 390,
  418ff., 1215
Quasi-random sequence 299ff., 881, 888
  Halton’s 300
  for Monte Carlo integration 304, 309, 318
  Sobol’s 300ff., 1160
see also Random number generator
Quicksort 320, 323ff., 330, 333, 1169f.
Quotient-difference algorithm 164
R-estimates 694
Radioactive decay 278
Radix base for floating point arithmetic 476,
  882, 907, 913, 1231, 1343, 1357
Radix conversion 902, 906, 913, 1357
radix() intrinsic function 1231
Radix sort 1172
Ramanujan’s identity for π 915
Random bits, generation of 287ff., 1159f.
Random deviates 266ff., 1141ff.
  binomial 285f., 1155
  exponential 278, 1151ff.
  gamma distribution 282f., 1153
  Gaussian 267, 279f., 571, 798, 1152f.
  normal 267, 279f., 571, 1152f.
  Poisson 283ff., 571, 1154f.
  quasi-random sequences 299ff., 881, 888,
    1160f.
  uniform 267ff., 1158f., 1166
Random number generator 266ff., 1141ff.
  bitwise operations 287
  Box-Muller algorithm 279, 1152
  Data Encryption Standard 290ff., 1144,
    1156ff.
  good choices for modulus, multiplier and
    increment 274ff.
  initializing 1144f.
  for integer-valued probability distribution
    283f., 1154
  integer vs. real implementation 273
  L’Ecuyer’s long period 271f.
  lagged Fibonacci generator 1142, 1148ff.
  linear congruential generator 267ff., 1142
  machine language 269
  Marsaglia shift register 1142, 1148ff.
  Minimal Standard, Park and Miller’s 269,
    1142
  nonrandomness of low-order bits 268f.
  parallel 1009
  perfect 272, 1141, 1150f.
  planes, numbers lie on 268
  portable 269ff., 1142
  primitive polynomials modulo 2 287ff.
  pseudo-DES 291, 1144, 1156ff.
  quasi-random sequences 299ff., 881, 888,
    1160f.
  quick and dirty 274
  quicker and dirtier 275
  in Quicksort 324
  random access to nth number 293
### Index to Volumes 1 and 2

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>random bits</td>
<td>287ff., 1159f.</td>
</tr>
<tr>
<td>recommendations</td>
<td>276f.</td>
</tr>
<tr>
<td>rejection method</td>
<td>281ff.</td>
</tr>
<tr>
<td>serial 1141f.</td>
<td></td>
</tr>
<tr>
<td>shuffling procedure 270, 272</td>
<td></td>
</tr>
<tr>
<td>in simulated annealing method</td>
<td>438</td>
</tr>
<tr>
<td>spectral test 274</td>
<td></td>
</tr>
<tr>
<td>state space 1143f.</td>
<td></td>
</tr>
<tr>
<td>state space exhaustion 1141</td>
<td></td>
</tr>
<tr>
<td>subtractive method 273, 1143</td>
<td></td>
</tr>
<tr>
<td>system-supplied 267f.</td>
<td></td>
</tr>
<tr>
<td>timings 276ff., 1151</td>
<td></td>
</tr>
<tr>
<td>trick for trigonometric functions 280</td>
<td></td>
</tr>
<tr>
<td>Random numbers</td>
<td>see Monte Carlo;</td>
</tr>
<tr>
<td>Random walk</td>
<td>Random deviates</td>
</tr>
<tr>
<td>random_number() intrinsic function</td>
<td>1141, 1143</td>
</tr>
<tr>
<td>random_seed() intrinsic function</td>
<td>1141</td>
</tr>
<tr>
<td>RANDU, infamous routine</td>
<td>268</td>
</tr>
<tr>
<td>Range 53f.</td>
<td></td>
</tr>
<tr>
<td>Rank (matrix) 53</td>
<td></td>
</tr>
<tr>
<td>kernel of finite 785</td>
<td></td>
</tr>
<tr>
<td>Rank (sorting) 320, 332, 1176</td>
<td></td>
</tr>
<tr>
<td>Rank (statistics) 633ff., 694f., 1277</td>
<td></td>
</tr>
<tr>
<td>Kendall’s tau 637ff., 1279</td>
<td></td>
</tr>
<tr>
<td>Spearman correlation coefficient 634f.,</td>
<td>1277ff.</td>
</tr>
<tr>
<td>sum squared differences of 634, 1277</td>
<td></td>
</tr>
<tr>
<td>Ratio variable (statistics) 623</td>
<td></td>
</tr>
<tr>
<td>Rational Chebyshev approximation</td>
<td>197ff., 1081f.</td>
</tr>
<tr>
<td>Rational function 99, 167ff., 194ff.,</td>
<td>1080f.</td>
</tr>
<tr>
<td>approximation for Bessel functions</td>
<td>225</td>
</tr>
<tr>
<td>approximation for continued fraction</td>
<td>164, 211, 219f.</td>
</tr>
<tr>
<td>Chebyshev approximation 197ff., 1081f.</td>
<td></td>
</tr>
<tr>
<td>evaluation of 170, 1072f.</td>
<td></td>
</tr>
<tr>
<td>extrapolation in Bulirsch-Stoer method</td>
<td>718ff., 726, 1306f.</td>
</tr>
<tr>
<td>interpolation and extrapolation using</td>
<td>99, 104ff., 194f.,</td>
</tr>
<tr>
<td>718ff., 726, 1306f.</td>
<td>726</td>
</tr>
<tr>
<td>as power spectrum estimate 566</td>
<td></td>
</tr>
<tr>
<td>interpolation and extrapolation using</td>
<td>1043f., 1080ff., 1306f</td>
</tr>
<tr>
<td>minimax 198</td>
<td></td>
</tr>
<tr>
<td>Re-entrant procedure 1052</td>
<td></td>
</tr>
<tr>
<td>real() intrinsic function, ambiguity of</td>
<td>947</td>
</tr>
<tr>
<td>Realizable (causal) 552, 554f.</td>
<td></td>
</tr>
<tr>
<td>relocate() utility function 955, 990,</td>
<td>1070, 1302</td>
</tr>
<tr>
<td>992, 1070, 1302</td>
<td></td>
</tr>
<tr>
<td>Rearranging see Sorting</td>
<td></td>
</tr>
<tr>
<td>Reciprocal, multiple precision 910f.,</td>
<td>1355f.</td>
</tr>
<tr>
<td>1355f.</td>
<td></td>
</tr>
<tr>
<td>Record, in data file 329</td>
<td></td>
</tr>
<tr>
<td>Recurrence relation 172ff., 971ff.</td>
<td></td>
</tr>
<tr>
<td>arithmetic progression 971f., 996</td>
<td></td>
</tr>
<tr>
<td>associated Legendre polynomials 247</td>
<td></td>
</tr>
<tr>
<td>Bessel function 172, 224, 227f., 234</td>
<td></td>
</tr>
<tr>
<td>binomial coefficients 209</td>
<td></td>
</tr>
<tr>
<td>Bulirsch-Stoer 105f.</td>
<td></td>
</tr>
<tr>
<td>characteristic polynomial of tridiagonal</td>
<td></td>
</tr>
<tr>
<td>matrix 469</td>
<td></td>
</tr>
<tr>
<td>Clenshaw’s recurrence formula 176f.</td>
<td>and continued fraction 175</td>
</tr>
<tr>
<td>continued fraction evaluation 164f.</td>
<td></td>
</tr>
<tr>
<td>convergence 175</td>
<td></td>
</tr>
<tr>
<td>cosine function 172, 500</td>
<td></td>
</tr>
<tr>
<td>cyclic reduction 974</td>
<td></td>
</tr>
<tr>
<td>dominant solution 174</td>
<td></td>
</tr>
<tr>
<td>exponential integrals 172</td>
<td></td>
</tr>
<tr>
<td>gamma function 206</td>
<td></td>
</tr>
<tr>
<td>generation of random bits 287f.</td>
<td></td>
</tr>
<tr>
<td>geometric progression 972, 996</td>
<td></td>
</tr>
<tr>
<td>Golden Mean 21</td>
<td></td>
</tr>
<tr>
<td>Legendre polynomials</td>
<td>172</td>
</tr>
<tr>
<td>minimal vs. dominant solution 174</td>
<td></td>
</tr>
<tr>
<td>modified Bessel function 232</td>
<td></td>
</tr>
<tr>
<td>Neville’s 103, 182</td>
<td></td>
</tr>
<tr>
<td>orthonormal polynomials 142</td>
<td></td>
</tr>
<tr>
<td>Perron’s theorems 174f.</td>
<td></td>
</tr>
<tr>
<td>Pincherle’s theorem 175</td>
<td></td>
</tr>
<tr>
<td>for polynomial cumulants 977, 999,</td>
<td></td>
</tr>
<tr>
<td>1071f.</td>
<td></td>
</tr>
<tr>
<td>polynomial interpolation 103, 183</td>
<td></td>
</tr>
<tr>
<td>primitive polynomials modulo 2 287f.</td>
<td></td>
</tr>
<tr>
<td>random number generator 268</td>
<td></td>
</tr>
<tr>
<td>rational function interpolation 105f.,</td>
<td></td>
</tr>
<tr>
<td>1043 recursive doubling 973, 977, 988,</td>
<td>999, 1071f., 1073</td>
</tr>
<tr>
<td>second order 974f., 1074</td>
<td></td>
</tr>
<tr>
<td>sequence of trig functions 173</td>
<td></td>
</tr>
<tr>
<td>sine function 172, 500</td>
<td></td>
</tr>
<tr>
<td>spherical harmonics 247</td>
<td></td>
</tr>
<tr>
<td>stability of 21, 173ff., 177, 224f.,</td>
<td></td>
</tr>
<tr>
<td>227f., 232, 247, 975</td>
<td></td>
</tr>
<tr>
<td>trig functions 572</td>
<td></td>
</tr>
<tr>
<td>weight of Gaussian quadrature 144f.</td>
<td></td>
</tr>
<tr>
<td>Recursion</td>
<td>in Fortran 90 958</td>
</tr>
<tr>
<td>in multigrid method 865, 1009, 1336</td>
<td></td>
</tr>
<tr>
<td>Recursive doubling 973f., 979</td>
<td></td>
</tr>
<tr>
<td>cumulants of polynomial 977, 999,</td>
<td></td>
</tr>
<tr>
<td>1071f.</td>
<td></td>
</tr>
<tr>
<td>linear recurrences 973, 988, 1073</td>
<td></td>
</tr>
<tr>
<td>tridiagonal systems 976</td>
<td></td>
</tr>
<tr>
<td>RECURSIVE keyword 958, 1065, 1067</td>
<td></td>
</tr>
<tr>
<td>Recursive Monte Carlo integration 306ff.,</td>
<td>1161</td>
</tr>
<tr>
<td>Recursive procedure 2/xiv, 958, 1065,</td>
<td></td>
</tr>
<tr>
<td>1067, 1166</td>
<td></td>
</tr>
<tr>
<td>as parallelization tool 958</td>
<td></td>
</tr>
<tr>
<td>base case 958</td>
<td></td>
</tr>
<tr>
<td>for multigrid method 1009, 1336</td>
<td></td>
</tr>
<tr>
<td>re-entrant 1052</td>
<td></td>
</tr>
<tr>
<td>Recursive stratified sampling 314ff., 1164f.</td>
<td></td>
</tr>
<tr>
<td>Red-black see Odd-even ordering</td>
<td></td>
</tr>
<tr>
<td>Reduction functions 948ff.</td>
<td></td>
</tr>
<tr>
<td>Reduction of variance in Monte Carlo</td>
<td></td>
</tr>
<tr>
<td>integration 299, 306ff.</td>
<td></td>
</tr>
<tr>
<td>References (explanation) 4f.</td>
<td></td>
</tr>
<tr>
<td>References (general bibliography) 916ff.,</td>
<td>1359f.</td>
</tr>
<tr>
<td>Reflection formula for gamma function 206</td>
<td></td>
</tr>
<tr>
<td>Regula falsi (false position) 347ff.,</td>
<td>1185f.</td>
</tr>
<tr>
<td>Regularity condition 775</td>
<td></td>
</tr>
<tr>
<td>Regularization</td>
<td>compared with optimal filtering 801</td>
</tr>
<tr>
<td>compared with constrained linear inversion method 799ff. of inverse problems 796f.</td>
<td>799ff.</td>
</tr>
<tr>
<td>linear 799ff.</td>
<td>813</td>
</tr>
</tbody>
</table>
Index to Volumes 1 and 2

Index to Volumes 1 and 2

1479

objective criterion 802
Tikhonov-Miller 799ff.
trade-off curve 799
two-dimensional 803
zeroth order 797
see also Inverse problems
Regularizing operator 798
Reid, John 2/xiv, 2/xvi
Rejection method for random number genera-
tor 281ff.
Relaxation method
for algebraically difficult sets 763
automated allocation of mesh points 774f.,
777
computation of spheroidal harmonics 764ff.,
1319ff.
for differential equations 746ff., 753ff.,
1316ff.
elliptic partial differential equations 823,
854ff., 1332f.
example 764ff., 1319ff.
Gauss-Seidel method 855, 864ff., 876,
1338, 1341
internal boundary conditions 775ff.
internal singular points 775ff.
Jacobi's method 855f., 864
successive over-relaxation (SOR) 857ff.,
862, 866, 1332f.
see also Multigrid method
Remes algorithms
exchange algorithm 553
for minimax rational function 199
reshape() intrinsic function 950
communication bottleneck 969
order keyword 1050, 1246
Residual 49, 54, 78
in multigrid method 863, 1338
Resolution function, in Backus-Gilbert method 807
Response function 531
Restriction operator 864, 1337
RESULT keyword 958, 1073
Reward, $1000 offered 272, 1141, 1150f.
Richardson's deferred approach to the limit
134, 137, 182, 702, 718ff., 726, 788,
869
see also Bulirsch-Stoer method
Richtmyer artificial viscosity 837
Ridders' method, for numerical derivatives
182, 1075
Ridders' method, root finding 341, 349, 351,
1187
Riemann shock problem 837
Right eigenvalues and eigenvectors 451
Rise/ fall time 548f.
Robust estimation 653, 694ff., 700, 1294
Andrew's sine 697
average deviation 605
double exponential errors 696
Kalman filtering 700
Lorentzian errors 696f.
mean absolute deviation 605
nonparametric correlation 633ff., 1277
Tukey's biweight 697
use of a priori covariances 700
see also Statistical tests
Romberg integration 124, 134ff., 137, 182,
717, 788, 1054f., 1065
Root finding 143, 340ff., 1009, 1059
advanced implementations of Newton's rule
386
Bairstow's method 364, 370, 1193
bisection 343, 346ff., 352f., 359, 390, 469,
698, 1184f.
bracketing of roots 341, 343ff., 353f.,
362, 364, 369, 1183f.
Brent's method 341, 349, 660f., 1188f.,
1286
Broyden's method 373, 382ff., 386, 1199
compared with multidimensional minimiza-
tion 375
complex analytic functions 364
in complex plane 204
convergence criteria 347, 374
deflation of polynomials 362ff., 370f.,
1192
without derivatives 354
double root 341
eigenvalue methods 368, 1193
false position 347ff., 1185f.
Jenkins-Traub method 369
Laguerre's method 341, 366f., 1191f.
Lehmer-Schur algorithm 369
Maehly's procedure 364, 371
matrix method 368, 1193
Muller's method 364, 372
multiple roots 341
Newton's rule 143f., 180, 341, 355ff.,
362, 364, 370, 372ff., 376, 469, 740,
749f., 754, 787, 874, 876, 911f., 1059,
1189, 1194, 1196, 1314ff., 1339, 1341,
1355f.
pathological cases 343, 356, 362, 372
polynomials 341, 362ff., 449, 1191f.
in relaxation method 754, 1316
Ridders' method 341, 349, 351, 1187
root-polishing 356, 363ff., 369ff., 1193
safe Newton's rule 359, 1190
secant method 347ff., 358, 364, 399,
1186f.
in shooting method 746, 749f., 1314f.
singular Jacobian in Newton's rule 386
stopping criterion for polynomials 366
use of minimum finding 341
using derivatives 355ff., 1189
zero suppression 372
see also Roots
Root polishing 356, 363ff., 369ff., 1193
Roots
Chebyshev polynomials 184
cubic root of unity 999f., 1379
Hermite polynomials, approximate 1062
Jacobi polynomials, approximate 1064
Laguerre polynomials, approximate 1061
multiple 341, 364ff., 1192
nonlinear equations 340ff.
polynomials 341, 362ff., 449, 1191f.
quadratic equations 178
Index to Volumes 1 and 2

- reflection in unit circle 560, 1257
- square, multiple precision 912, 1356
  - see also Root finding
- Rosenbrock method 730, 1308
  - compared with semi-implicit extrapolation 739
  - stepsize control 731, 1308f.
- Roundoff error 20, 881, 1362
  - bracketing a minimum 399
  - compile time vs. run time 1012
  - conjugate gradient method 824
  - eigenvalue systems 458, 467, 470, 473, 476, 479, 483
  - extended trapezoidal rule 132
  - general linear least squares 668, 672
  - graceful 883, 1343
  - hardware aspects 882, 1343
  - Householder reduction 466
  - IEEE standard 882f., 1343
  - interpolation 100
  - least squares fitting 658, 668
  - Levenberg-Marquardt method 679
  - linear algebraic equations 23, 27, 29, 47, 56, 84, 1022
  - linear predictive coding (LPC) 564
  - magnification of 20, 47, 1022
  - maximum entropy method (MEM) 567
  - measuring 881f., 1343
  - multidimensional minimization 418, 422
  - multiple roots 362
  - numerical derivatives 180f.
  - recurrence relations 173
  - reduction to Hessenberg form 479
  - series 164f.
  - straight line fitting 658
  - variance 607
- Row degeneracy 22
- Row-indexed sparse storage 71f., 1030
  - transpose 73f.
- Row operations on matrix 28, 31f.
- Row totals 624
- RST properties (reflexive, symmetric, transitive) 338
- Runge-Kutta method 702, 704ff., 731, 740, 1297f., 1308
  - Cash-Karp parameters 710, 1299f.
  - embedded 709f., 731, 1298, 1308
  - high-order 705
  - quality control 722
  - stepsize control 708ff.
- Run-length encoding 901
- Runge-Kutta method
  - high-order 1297
  - stepsize control 1298f.
- Rybicki, G.B. 84ff., 114, 145, 252, 522, 574, 600
  - S-box for Data Encryption Standard 1148
- Sampling
  - importance 306f.
  - Latin square or hypercube 305f.
  - recursive stratified 314ff., 1164
  - stratified 308f.
  - uneven or irregular 569, 648f., 1258
  - for numerical approximation 600ff.
  - Savitzky-Golay filters
    - for data smoothing 644ff., 1283f.
    - for numerical derivatives 183, 645
  - scale() intrinsic function 1107
  - scallop loss 548
  - Scatter-with-combine functions 984, 1002f., 1032, 1366, 1380f.
  - scatter_add() utility function 984, 990, 1002, 1032
  - scatter_max() utility function 984, 990, 1003
  - Schonfelder, Lawrie 2/xi
  - Schrage’s algorithm 269
  - Schrödinger equation 842ff.
  - Schultz’s method for matrix inverse 49, 598
  - Scope 956ff., 1209, 1293, 1296
- Scoping unit 939
- SDLC checksum 890
- Searching
  - with correlated values 111, 1046f.
  - an ordered table 110f., 1045f.
  - selection 333, 1177f.
  - Secant method 341, 347ff., 358, 364, 399, 1186f.
  - Broyden’s method 382f., 1199f.
  - multidimensional (Broyden’s) 373, 382f., 1199
  - Second Euler-Maclaurin summation formula 135f.
- Second order differential equations 726, 1307
- Seed of random number generator 267, 1146f.
- Selection 320, 333, 1177f.
  - find m largest elements 336, 1179f.
  - heap algorithm 336, 1179
  - for median 698, 1294
  - operation count 333
  - by packing 1178
  - parallel algorithms 1178
  - by partition-exchange 333, 1177f.
  - without rearrangement 335, 1178f.
  - timings 336
  - use to find median 609
  - Semi-implicit Euler method 730, 735f.
  - Semi-implicit extrapolation method 730, 735f., 1310f.
  - compared with Rosenbrock method 739
  - stepsize control 737, 1311f.
  - Semi-implicit midpoint rule 735f., 1310f.
  - Semi-invariants of a distribution 608
  - Sentinel, in Quicksort 324, 333
  - Separable kernel 785
  - Separation of variables 246
  - Serial computing
    - convergence of quadrature 1060
    - random numbers 1141
  - sorting 1167
  - Serial data port 892
Index to Volumes 1 and 2

Series 159ff.
  accelerating convergence of 159ff.
  alternating 160f., 1070
  asymptotic 161
  Bessel function \( \nu \) 241
  Bessel function \( \gamma_2 \) 235
  Bessel functions 160, 223
  cosine integral 250
  divergent 161
  economization 192f., 195, 1080
  Euler's transformation 160f., 1070
  exponential integral 216, 218
  divergent 161
  economization 192f., 195, 1080
  Euler's transformation 160f., 1070
  extended beta function 219
  extended gamma function 210, 1090f.
  Laurent 566
  relation to continued fractions 163f.
  roundoff error in 164f.
  sine and cosine integrals 250
  sine function 160
  Taylor 355f., 408, 702, 709, 754, 759
  transformation of 160ff., 1070
  van Wijngaarden's algorithm 161, 1070

Shaft encoder 886

Shakespeare 9

Shampine's Rosenbrock parameters 732, 1308

shape() intrinsic function 938, 949

Shell algorithm (Shell's sort) 321ff., 1168

Sherman-Morrison formula 65ff., 83, 382

Shifting of eigenvalues 449, 470ff., 480

Shock wave 831, 837

Shooting method
  computation of spheroidal harmonics 772, 1321ff.
  for difficult cases 753, 1315ff.
  example 770ff., 1321ff.
  interior fitting point 752, 1315ff., 1323ff.

Shuffling to improve random number generator 270, 272

Side effects
  prevented by data hiding 957, 1209, 1293, 1296
  and PURE subprograms 960

Sidetole fall-off 548

Sidetole level 548

sign() intrinsic function, modified in Fortran 95 961

Signal, bandwidth limited 495

Significance (numerical) 19

Significance (statistical) 609f.
  one- vs. two-sided 632
  peak in Lomb periodogram 570
  of 2-d K-S test 640, 1281
  two-tailed 613

SIMD machines (Single Instruction Multiple Data) 964, 985f., 1009, 1084f.

Similarity transform 452ff., 456, 476, 478, 482

Simplex
  defined 402


use in simulated annealing 444, 1222ff.

Simpson's rule 124ff., 128, 133, 136f., 583, 782, 788f., 1053f.

Simpson's three-eighths rule 126, 789f.

Simulated annealing see Annealing, method of simulated

Simulation see Monte Carlo

Sine function
  evaluated from \( \tan(\theta/2) \) 173
  recurrence 172
  series 160

Sine integral 248, 250ff., 1123, 1125ff.
  continued fraction 250
  series 250

see also Cosine integral

Sine transform see Fast Fourier transform (FFT); Fourier transform

Singleton's algorithm for FFT 525

Singular value decomposition (SVD) 23, 25, 51ff., 1022
  approximation of matrices 58f.
  backsubstitution 56, 1022f.
  and bases for nullspace and range 53
  confidence levels from 693f.
  covariance matrix 693f.
  fewer equations than unknowns 57
  for inverse problems 797
  and least squares 54ff., 199f., 668, 670ff., 1081, 1290f.
  in minimization 410
  more equations than unknowns 57f.
  parallel algorithms 1026
  and rational Chebyshev approximation 199f., 1081f.
  of square matrix 53ff., 1023
  use for ill-conditioned matrices 56, 58, 449
  use for orthonormal basis 58, 94

Singularities
  of hypergeometric function 203, 263
  in integral equations 788ff., 1328
  in integral equations, worked example 792, 1328ff.
  in integrands 135ff., 788, 1055, 1328ff.
  removal in numerical integration 137ff., 788, 1057ff., 1328ff.

Singularity, subtraction of the 789

SIPSOL 824

Six-step framework, for FFT 983, 1240

size() intrinsic function 938, 942, 945, 948

Skew array section 2/xii, 945, 960, 985, 1284

Skewness of distribution 606, 608, 1269

Smoothing
  of data 114, 644ff., 1283f.
  of data in integral equations 781
  importance in multigrid method 865

sn function 261, 1137f.

Snyder, N.L. 1/xvi

Sobol's quasi-random sequence 300ff., 1160f.

Sonata 9

Sonnet 9

Sorting 320ff., 1167ff.
  bubble sort 1168
bubble sort cautioned against 321
compared to selection 333
covariance matrix 669, 681, 1289
eigenvectors 461f., 1227
Heapsort 320, 327f., 336, 1171f., 1179
index table 320, 329f., 1170, 1173ff., 1176
operation count 320ff.
by packing 1171
parallel algorithms 1168, 1171f., 1176
Quicksort 320, 323ff., 330, 333, 1169f.
radix sort 1172
rank table 320, 332, 1176
ranking 329, 1176
by reshaping array slices 1168
Shell's method 321ff., 1168
straight insertion 321f., 461f., 1167, 1227
SP , defined 937
SPARC or SPARCstation 1/xxii, 2/xix, 4
Sparse linear equations 23, 63ff., 732, 1030
band diagonal 43, 1019ff.
biconjugate gradient method 77, 599,
1034
data type for 1030
indexed storage 71f., 1030
in inverse problems 804
minimum residual method 78
named patterns 64, 822
partial differential equations 822ff.
relaxation method for boundary value problems 754, 1316
row-indexed storage 71f., 1030
wavelet transform 584, 598
see also Matrix
Spearman rank-order coefficient 634f., 694f., 1277
Special functions see Function
Spectral analysis see Fourier transform; Periodogram
Spectral density 541
and data windowing 545ff.
figures of merit for data windows 548f.
normalization conventions 542f.
one-sided PSD 492
periodogram 543ff., 566, 1258ff.
power spectral density (PSD) 492f.
power spectral density per unit time 493
power spectrum estimation by FFT 542ff., 1254ff.
power spectrum estimation by MEM 565ff., 1258
two-sided PSD 493
variance reduction in spectral estimation 545
Spectral lines, how to smooth 644
Spectral methods for partial differential equations 825
Spectral radius 856ff., 862
Spectral test for random number generator 274
Spectrum see Fourier transform
Spherical Bessel functions 234
routine for 245, 1121
Spherical harmonics 246ff.
orthogonality 246
routine for 247f., 1122
stable recurrence for 247
table of 246
see also Associated Legendre polynomials
Spheroidal harmonics 764ff., 770ff., 1319ff.
boundary conditions 765
normalization 765
routine for 768ff., 1319ff., 1323ff.
Spline 100
cubic 107ff., 1044f.
gives tridiagonal system 109
natural 109, 1044f.
operation count 109
two-dimensional (bicubic) 120ff., 1050ff.
spread() intrinsic function 945, 950, 969,
1000, 1094, 1290ff.
and dimensional expansion 966ff.
Spread matrix 808
Spread spectrum 290
Square root, complex 172
Square root, multiple precision 912, 1356ff.
Square window 546, 1254ff.
SSP (small-scale parallel) machines 965ff., 972, 974, 984, 1011, 1016ff., 1021, 1059ff., 1226ff., 1250
Stability 20ff.
of Clenshaw’s recurrence 177
Courant condition 829, 832ff., 836, 846
diffusion equation 840
of Gauss-Jordan elimination 27, 29
of implicit differencing 729, 840
mesh-drift in PDEs 834ff.
nonlinear 831, 837
partial differential equations 820, 827ff.
of polynomial deflation 363
in quadrature solution of Volterra equation 787f.
of recurrence relations 173ff., 177, 224ff.,
227ff., 232, 247
and stiff differential equations 728ff.
von Neumann analysis for PDEs 827ff., 830, 833ff., 840
see also Accuracy
Stabilized Kolmogorov-Smirnov test 621
Stabilizing functional 798
Staggered leapfrog method 833f.
Standard (probable) errors 1288, 1290
Standard deviation
of a distribution 605, 1269
of Fisher’s z 632
of linear correlation coefficient 630
of sum squared difference of ranks 635, 1277
Standard (probable) errors 610, 656, 661, 667, 671, 684
Stars, as text separator 1009
Statement function, superseded by internal subprogram 1057, 1256
Statement labels 9
Statistical error 653
Statistical tests 603ff., 1269ff.
Anderson-Darling 621
average deviation 605, 1269
bootstrap method 686ff.
chi-square 614f., 623ff., 1272, 1275f.

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>contingency coefficient C</td>
<td>625, 1275</td>
</tr>
<tr>
<td>contingency tables</td>
<td>622ff., 638, 1275f.</td>
</tr>
<tr>
<td>correlation</td>
<td>603f.</td>
</tr>
<tr>
<td>Cramer’s V</td>
<td>625, 1275</td>
</tr>
<tr>
<td>difference of distributions</td>
<td>614ff., 1272</td>
</tr>
<tr>
<td>difference of means</td>
<td>609ff., 1269f.</td>
</tr>
<tr>
<td>difference of variances</td>
<td>611, 613, 1271</td>
</tr>
<tr>
<td>entropy measures of association</td>
<td>626ff., 1275f.</td>
</tr>
<tr>
<td>F-test</td>
<td>611, 613, 1271</td>
</tr>
<tr>
<td>Kendall’s tau</td>
<td>634, 637ff., 1279</td>
</tr>
<tr>
<td>Kolmogorov-Smirnov</td>
<td>614, 617ff., 640, 694, 1273f., 1281</td>
</tr>
<tr>
<td>Kuiper’s statistic</td>
<td>621</td>
</tr>
<tr>
<td>Kurtosis</td>
<td>606, 608, 1269</td>
</tr>
<tr>
<td>L-estimates</td>
<td>694</td>
</tr>
<tr>
<td>linear correlation coefficient</td>
<td>630ff., 1276f.</td>
</tr>
<tr>
<td>M-estimates</td>
<td>694ff.</td>
</tr>
<tr>
<td>mean</td>
<td>603ff., 608ff., 1269f.</td>
</tr>
<tr>
<td>measures of association</td>
<td>604, 622ff., 1275</td>
</tr>
<tr>
<td>measures of central tendency</td>
<td>604ff., 1269</td>
</tr>
<tr>
<td>median</td>
<td>605, 694</td>
</tr>
<tr>
<td>mode</td>
<td>605</td>
</tr>
<tr>
<td>moments</td>
<td>604ff., 608, 1269</td>
</tr>
<tr>
<td>nonparametric correlation</td>
<td>633ff., 1277</td>
</tr>
<tr>
<td>Pearson’s r</td>
<td>630ff., 1276f.</td>
</tr>
<tr>
<td>phi statistic</td>
<td>625</td>
</tr>
<tr>
<td>R-estimates</td>
<td>694</td>
</tr>
<tr>
<td>rank correlation</td>
<td>633ff., 1277</td>
</tr>
<tr>
<td>robust</td>
<td>605, 634, 694ff.</td>
</tr>
<tr>
<td>semi-invariants</td>
<td>608</td>
</tr>
<tr>
<td>for shift vs. for spread</td>
<td>620f.</td>
</tr>
<tr>
<td>significance</td>
<td>609ff., 1269ff.</td>
</tr>
<tr>
<td>significance, one- vs. two-sided</td>
<td>613, 632</td>
</tr>
<tr>
<td>skewness</td>
<td>606, 608, 1269</td>
</tr>
<tr>
<td>Spearman rank-order coefficient</td>
<td>634ff., 694ff., 1277</td>
</tr>
<tr>
<td>standard deviation</td>
<td>605, 1269f.</td>
</tr>
<tr>
<td>strength vs. significance</td>
<td>609ff., 622</td>
</tr>
<tr>
<td>Student’s t</td>
<td>610, 631, 1269</td>
</tr>
<tr>
<td>Student’s t, for correlation</td>
<td>631</td>
</tr>
<tr>
<td>Student’s t, for correlation</td>
<td>631</td>
</tr>
<tr>
<td>Student’s t, paired samples</td>
<td>612, 1271</td>
</tr>
<tr>
<td>Student’s t, Spearman rank-order coefficient</td>
<td>634, 1277</td>
</tr>
<tr>
<td>Student’s t, unequal variances</td>
<td>611, 1270</td>
</tr>
<tr>
<td>sum squared difference of ranks</td>
<td>635, 1277</td>
</tr>
<tr>
<td>Tukey’s trimean</td>
<td>694</td>
</tr>
<tr>
<td>two-dimensional</td>
<td>640, 1281ff.</td>
</tr>
<tr>
<td>variance</td>
<td>603ff., 607ff., 612ff., 1269ff.</td>
</tr>
<tr>
<td>Wilcoxon</td>
<td>694</td>
</tr>
<tr>
<td>see also Error; Robust estimation</td>
<td></td>
</tr>
<tr>
<td>Steak, without sizzle</td>
<td>809</td>
</tr>
<tr>
<td>Steed’s method</td>
<td></td>
</tr>
<tr>
<td>Bessel functions</td>
<td>234, 239</td>
</tr>
<tr>
<td>continued fractions</td>
<td>164f.</td>
</tr>
<tr>
<td>Steepest descent method</td>
<td>414</td>
</tr>
<tr>
<td>in inverse problems</td>
<td>804</td>
</tr>
<tr>
<td>Step</td>
<td>130, 708ff., 1052</td>
</tr>
<tr>
<td>tripling</td>
<td>136, 1055</td>
</tr>
<tr>
<td>Stieljes, procedure</td>
<td>151</td>
</tr>
<tr>
<td>Stiff equations</td>
<td>703, 727ff., 1308ff.</td>
</tr>
<tr>
<td>Kaps-Rentrop method</td>
<td>730, 1308</td>
</tr>
<tr>
<td>methods compared</td>
<td>739</td>
</tr>
<tr>
<td>predictor-corrector method</td>
<td>730</td>
</tr>
<tr>
<td>r.h.s. independent of e</td>
<td>729f.</td>
</tr>
<tr>
<td>Rosenbrock method</td>
<td>730, 1308</td>
</tr>
<tr>
<td>scaling of variables</td>
<td>730</td>
</tr>
<tr>
<td>semi-implicit extrapolation method</td>
<td>730, 1310ff.</td>
</tr>
<tr>
<td>semi-implicit midpoint rule</td>
<td>735ff., 1310ff.</td>
</tr>
<tr>
<td>Stiff functions</td>
<td>100, 399</td>
</tr>
<tr>
<td>Stirling’s approximation</td>
<td>206, 812</td>
</tr>
<tr>
<td>Stoermer’s rule</td>
<td>726, 1307</td>
</tr>
<tr>
<td>Stopping criterion, in multigrid method</td>
<td>875f.</td>
</tr>
<tr>
<td>Stopping criterion, in polynomial root finding</td>
<td>366</td>
</tr>
<tr>
<td>Storage</td>
<td></td>
</tr>
<tr>
<td>band diagonal matrix</td>
<td>44, 1019</td>
</tr>
<tr>
<td>sparse matrices</td>
<td>71f., 1030</td>
</tr>
<tr>
<td>Storage association</td>
<td>2/xiv</td>
</tr>
<tr>
<td>Straight injection</td>
<td>867</td>
</tr>
<tr>
<td>Straight insertion</td>
<td>321ff., 461ff., 1167, 1227</td>
</tr>
<tr>
<td>Straight line fitting</td>
<td>655ff., 667f., 1285ff.</td>
</tr>
<tr>
<td>errors in both coordinates</td>
<td>660ff., 1286ff.</td>
</tr>
<tr>
<td>robust estimation</td>
<td>698, 1294ff.</td>
</tr>
<tr>
<td>Strassen’s fast matrix algorithms</td>
<td>96f.</td>
</tr>
<tr>
<td>Stratified sampling, Monte Carlo</td>
<td>308ff., 314</td>
</tr>
<tr>
<td>Stopping criterion, in polynomial root finding</td>
<td>969</td>
</tr>
<tr>
<td>Strongly implicit procedure (SIPSOL)</td>
<td>824</td>
</tr>
<tr>
<td>Structure constructor</td>
<td>2/xii</td>
</tr>
<tr>
<td>Structured programming</td>
<td>5ff.</td>
</tr>
<tr>
<td>Student’s probability distribution</td>
<td>221f.</td>
</tr>
<tr>
<td>Student’s t-test</td>
<td></td>
</tr>
<tr>
<td>for correlation</td>
<td>631</td>
</tr>
<tr>
<td>for difference of means</td>
<td>610, 1269</td>
</tr>
<tr>
<td>for difference of means (paired samples)</td>
<td>612, 1271</td>
</tr>
<tr>
<td>for difference of means (unequal variances)</td>
<td>611, 1270</td>
</tr>
<tr>
<td>for difference of ranks</td>
<td>635, 1277</td>
</tr>
<tr>
<td>Spearman rank-order coefficient</td>
<td>634, 1277</td>
</tr>
<tr>
<td>Sturmian sequence</td>
<td>469</td>
</tr>
<tr>
<td>Sub-random sequences see Quasi-random sequence</td>
<td></td>
</tr>
<tr>
<td>Subprogram</td>
<td>938</td>
</tr>
<tr>
<td>Subtraction, multiple precision</td>
<td>907, 1353</td>
</tr>
<tr>
<td>Subtractive method for random number generator</td>
<td>273, 1143</td>
</tr>
<tr>
<td>Subvector scaling</td>
<td>972, 974, 996, 1000</td>
</tr>
<tr>
<td>Successive over-relaxation (SOR)</td>
<td>857ff., 862, 1332f.</td>
</tr>
<tr>
<td>bad in multigrid method</td>
<td>866</td>
</tr>
<tr>
<td>Chebyshev acceleration</td>
<td>859ff., 1332f.</td>
</tr>
<tr>
<td>choice of overrelaxation parameter</td>
<td>858</td>
</tr>
<tr>
<td>with logical mask</td>
<td>1333f.</td>
</tr>
<tr>
<td>parallelization</td>
<td>1333</td>
</tr>
<tr>
<td>sum() intrinsic function</td>
<td>945, 948, 966</td>
</tr>
<tr>
<td>Subtracted difference of ranks</td>
<td>634, 1277</td>
</tr>
</tbody>
</table>
Sums see Series
Sun 1/xxii, 2/xix, 886
SPARCstation 1/xxii, 2/xix, 4
Supernova 1987A 640
SVD see Singular value decomposition (SVD)
swap() utility function 987, 990f., 1015, 1210
Symbol, of operator 866f.
Synthetic division 84, 167, 362, 370
parallel algorithms 977ff., 999, 1048, 1071f., 1079, 1192
repeated 978f.
Systematic errors 653

Tableau (interpolation) 103, 183
Tangent function, continued fraction 163
Target, for pointer 938f., 945, 952f.
Taylor series 180, 355f., 408, 702, 709, 742, 754, 759
Test programs 3
Thermodynamics, analogy for simulated annealing 437
Thinking Machines, Inc. 964
Threshold multiply of sparse matrices 74, 1031
Tides 560f.
Tikhonov-Miller regularization 799ff.
Time domain 490
Time splitting 847f., 861
tiny() intrinsic function 952f., 1070, 1266, 1293, 1302
Tongue twisters 333
Torus 297f., 304
Trade-off curve 795, 809
Transformations 1/xxi, 2/xxi.

Transformational functions 948ff.
Transforms, number theoretic 503f.
Transport error 831f.
transpose() intrinsic function 950, 960, 969, 981, 1050, 1246
Transpose of sparse matrix 73f.
Trapezoidal rule 125, 127, 130ff., 134f., 579, 583, 782, 786, 1052, 1326f.
Traveling salesman problem 438ff., 1219ff.
Tridiagonal matrix 42, 63, 150, 453f., 488, 839f., 1018f.
in alternating-direction implicit method (ADI) 861f.
from cubic spline 109
cyclic 67, 1030
in cyclic reduction 853
eigenvalues 469ff., 1228
with fringes 822
from operator splitting 861f.
parallel algorithm 975, 1018, 1229f.
recursive splitting 1229f.
reduction of symmetric matrix to 462ff., 470, 1227f.
serial algorithm 1018f.
see also Matrix

Trigonometric
functions, linear sequences 173
functions, recurrence relation 172, 572
functions, tan(θ/2) as minimal 173
interpolation 99
solution of cubic equation 179ff.
Truncation error 20f., 399, 709, 881, 1362
in multigrid method 875
in numerical derivatives 180
Tukey’s biweight 697
Tukey’s trimean 694
Turbo Pascal (Borland) 8
Twin errors 895
Two-dimensional see Multidimensional
Two-dimensional K–S test 640, 1281ff.
Two-pass algorithm for variance 607, 1269
Two-point boundary value problems 702, 745ff., 1314ff.
automated allocation of mesh points 774ff., 777
boundary conditions 745ff., 749, 751f., 771, 1314f.
difficult cases 753, 1315f.
free boundary problem 748, 776
grid (mesh) points 746f., 754, 774ff., 777
internal boundary conditions 775ff.
internal singular points 775ff.
linear requires no iteration 751
multiple shooting 753
problems reducible to standard form 748
regularity condition 775
relaxation method 746f., 753ff., 1316ff.
relaxation method, example of 764ff., 1319
shooting to a fitting point 751ff., 1315f., 1323ff.
shooting method, example of 770ff., 1321ff.
singular endpoints 751, 764, 771, 1315f., 1319ff.
see also Elliptic partial differential equations

Two-sided exponential error distribution 696
Two-sided power spectral density 493
Two-step Lax-Wendroff method 835ff.
Two-volume edition, plan of 1/xxii
Two’s complement arithmetic 1144
Type declarations, explicit vs. implicit 2

U

bound() intrinsic function 949
ULTRIX 1/xxiii, 2/xix
Uncertainty coefficient 628
Uncertainty principle 600
Undefined status, of arrays and pointers 952f., 961, 1070, 1266, 1293, 1302
Underflow, in IEEE arithmetic 883, 1343
Underrelaxation 857
Uniform deviates see Random deviates, uniform
Index to Volumes 1 and 2

1485

Unitary (function) 843f.
Unitary (matrix) see Matrix
unitLmatri() utility function 985, 990, 1006, 1216, 1226, 1325
UNIX 1/xxiii, 2/viii, 2/xix, 4, 17, 276, 293, 886
Upper Hessenberg matrix see Hessenberg matrix
U.S. Postal Service barcode 894
unpack() intrinsic function 950, 964
Upwind differencing 832f., 837
USE statement 936, 939f., 954, 957, 1067, 1384
USES keyword in program listings 2
Utility functions 987ff., 1364ff.
add vector to matrix diagonal 1004, 1234, 1366, 1381
alphabetical listing 988ff.
argument checking 994f., 1370f.
array reallocation 992, 1070f., 1365, 1368f.
assertion of numerical equality 995, 1022, 1365, 1370f.
compared to intrinsics 990ff.
complex n
th root of unity 999f., 1379
create unit matrix 1006, 1382
cumulative product of an array 997f., 1072, 1086, 1375
cumulative sum of an array 997, 1280f., 1365, 1375
data types 1361
elemental functions 1364
error handling 994f., 1036, 1370f.
generic functions 1364
gerithmic progression 996f., 1365, 1372ff.
get diagonal of matrix 1005, 1226f., 1366, 1381f.
length of a vector 1008, 1383
linear recurrence 996
location in an array 992ff., 1015, 1017f.
location of first logical “true” 993, 1041, 1369
location of maximum array value 993, 1015, 1017, 1365, 1369
location of minimum array value 993, 1369f.
logarithm function 994, 1086, 1090, 1092, 1365, 1370
lower triangular mask 1007, 1200, 1382
masked polynomial evaluation 1378
masked swap of elements in two arrays 1368
moving data 990ff., 1015
multiply vector into matrix diagonal 1004f., 1366, 1381
nrutil.f90 (module file) 1364ff.
outer difference of vectors 1001, 1366, 1380
overloading 1364
partial cumulants of a polynomial 999, 1071, 1192f., 1365, 1378f.
polynomial evaluation 996, 998f., 1258, 1365, 1376ff.
scalar-with-add 1002f., 1032f., 1366, 1380f.
scalar-with-combine 1002f., 1032f., 1380f.
scalar-with-max 1002f., 1366, 1381
set diagonal elements of matrix 1005, 1200, 1382
skew operation on matrices 1004ff., 1381ff.
swap elements of two arrays 991, 1015, 1365ff.
upper triangular mask 1006, 1226, 1305, 1382

V-cycle 865, 1336
vabs() utility function 990, 1008, 1290
Validation of Numerical Recipes procedures 3f.
Van Cittert’s method 804
Van Wijngaarden-Dekker-Brent method see Brent’s method
Vandermonde matrix 82ff., 114, 1037, 1047
Variable length code 896, 1346ff.
Variable metric method 390, 418ff., 1215
vector PRODUCTS 1365ff.

Variance(s)
correlation 605
of distribution 603ff., 608, 611, 613, 1269
pooled 610
reduction of (in Monte Carlo) 299, 306ff.
statistical differences between two 609, 1271
two-pass algorithm for computing 607, 1269
see also Covariance

Variational methods, partial differential equations 824
VAX 275, 293
Vector(s)
length 1008, 1383
norms 1036
outer difference 1001, 1366, 1380
outer operations on vectors 1000ff., 1379f.
outer product of vectors 1000f., 1076, 1365f., 1379
outer quotient of vectors 1001, 1379
overloading 1364
partial cumulants of a polynomial 999, 1071, 1192f., 1365, 1378f.
polynomial evaluation 996, 998f., 1258, 1365, 1376ff.
scalar-with-add 1002f., 1032f., 1366, 1380f.
scalar-with-combine 1002f., 1032f., 1380f.
scalar-with-max 1002f., 1366, 1381
set diagonal elements of matrix 1005, 1200, 1382
skew operation on matrices 1004ff., 1381ff.
swap elements of two arrays 991, 1015, 1365ff.
upper triangular mask 1006, 1226, 1305, 1382
Index to Volumes 1 and 2

Viète’s formulas for cubic roots 179
Vienna Fortran 2/xv
Virus, computer 889
Viscosity
artificial 831, 837
numerical 830f., 837
Visibility 956ff., 1209, 1293, 1296
VMS 1/xxii, 2/xix
Volterra equations 780f., 1326
  adaptive stepsize control 788
  analogy with ODEs 786
  block-by-block method 788
  first kind 781, 786
  nonlinear 781, 787
  second kind 781, 786ff., 1326f.
  unstable quadrature 787f.
von Neuman, John 963, 965
von Neumann-Richtmyer artificial viscosity 837
von Neumann stability analysis for PDEs 827f., 830, 833f., 840
Vowellish (coding example) 896f., 902

W
  cycle 865, 1336
  Warranty, disclaimer of 1/xx, 2/xvii
Wave equation 246, 818, 825f.
Wavelet transform 584ff., 1264ff.
  appearance of wavelets 590ff.
  approximation condition of order $p$ 585
  coefficient values 586, 589, 1265
  contrasted with Fourier transform 584, 594
  Daubechies wavelet filter coefficients 584ff., 588, 590ff., 594, 598, 1264ff.
  detail information 585
  discrete wavelet transform (DWT) 586f., 1264
  DWT (discrete wavelet transform) 586f., 1264ff.
  eliminating wrap-around 587
  fast solution of linear equations 597ff.
  filters 592f.
  and Fourier domain 592f.
  image processing 596f.
  for integral equations 782
  inverse 587
  Lemarie’s wavelet 593
  of linear operator 597f.
  mother-function coefficient 587
  mother functions 584
  multidimensional 595, 1267f.
  nonsmoothness of wavelets 591
  pyramidal algorithm 586, 1264
  quadrature mirror filter 585
  smooth information 585
  truncation 594f.
  wavelet filter coefficient 584, 587
  wavelets 584, 590ff.
Wavelets see Wavelet transform
Weber function 204
Weighted Kolmogorov-Smirnov test 621
Weighted least-squares fitting see Least squares fitting
Weighting, full vs. half in multigrid 867
Weights for Gaussian quadrature 140ff., 788f., 1059ff., 1328f.
  nonclassical weight function 151ff., 788f., 1064f., 1328f.
  Welch window 547, 1254ff.
  WG5 (ISO/IEC JTC1/SC22/WG5 Committee) 2/xvii
  where construct 943, 1291
  contrasted with merge 1023
  for iteration of a vector 1060
  nested 2/xv, 943, 960, 1100
  not MIMD 985
While iteration 13
Wiener filtering 535, 539ff., 558, 644
  compared to regularization 801
  Wiener-Khinchin theorem 492, 558, 566f.
  Wilcoxon test 694
Window function
  Bartlett 547, 1254ff.
  flat-topped 549
  Hamming 547
  Hann 547
  Parzen 547
  square 544, 546, 1254ff.
  Welch 547, 1254ff.
Windowing for spectral estimation 1255f.
  Windows 95 2/xix
  Windows NT 2/xix
Winograd Fourier transform algorithms 503
Woodbury formula 68ff., 83
Wordlength 18
Workspace, reallocation in Fortran 90 1070f.
World Wide Web, Numerical Recipes site 1/xx, 2/xvii
Wraparound
  in integer arithmetic 1146, 1148
  order for storing spectrum 501
  problem in convolution 533
Wronskian of Bessel functions 234
X
  .25 protocol 890
  X3J3 Committee 2/viii, 2/xff., 2/xv, 947, 959, 964, 968, 990
  XMODEM checksum 889
  X-ray diffraction pattern, processing of 805
Y
  ale Sparse Matrix Package 64, 71
Z
  -transform 554, 559, 565
  Z-transformation, Fisher’s 631ff., 1276
  Zaman, A. 1149
  Zealots 814
  Zebra relaxation 866
  Zero contours 372
  Zero-length array 944
  Zeroth-order regularization 796ff.
  Zip code, barcode for 894
  Ziv-Lempel compression 896
  zroots_unity() utility function 974, 990, 999