A Fortran 90 Tutorial*

Zane Dodson
Computer Science Department
University of New Mexico
Revised by Brian T. Smith, Jan. 1997

August 10, 1997

Contents

*This work partially supported by the Numerical Algorithms Group Ltd. and the National Science Foundation (CDA-9017953).

1 Survey of the New Fortran Standard

Fortran 90 has many new features that make it a modern and robust language for numerical programming. In addition to providing many new language constructs, Fortran 90 contains Fortran 77 as a subset (except for four small inconsistencies). Consequently, all Fortran 77 programs can be compiled and should produce identical results. However, a few aspects of Fortran 77 have been labeled as obsolete and may be removed from the next standard, allowing a progression of the language without supporting all previously available features. This survey section is meant to highlight most of the new features in Fortran 90 and give some indication of their use. Examples and exercises begin in section 2.

1.1 New Source Form

One of the most visible features of the new Fortran standard is its free source form. Fortran statements may now appear anywhere on a source line, and columns are no longer reserved. Line continuation has also been improved and uses the '`&`' character at the end of source line to be continued. Trailing comments may be used and begin with a '`!`' character and continue to the end of the source line. The semicolon, '`;`', is used as a statement separator and allows multiple statements to be placed on a single source line. This new source form is illustrated in the following program segment.

```
tmp = x; x = y; y = tmp  ! Swap x and y.
print *, 'The values of x and y are ', &
   x, y
```

Blanks are now significant and the underscore character is permissible in an identifier. The number of significant characters in an identifier name has increased from 6 to 31. Fortran 77's fixed source form, line continuation, and comment specification is also acceptable in Fortran 90.

1.2 Array Processing

Array processing features are the most significant of the language enhancements offered by the new standard. Popular Fortran 77 extensions, such as whole array operations, masked assignment, array sections, and vector subscripts that are common in Cray Fortran and CM (Connection Machine) Fortran, are now part of the Fortran 90 standard. These types of constructs are important for parallel and vector computers.

Arithmetic and logical operations work elementally on arrays, so that, for example, `C = A + B` calculates the sum of two matrices, element-by-element, storing the result in `C`. The statement `Z = ( A > 0 )` creates a logical array with elements whose entries are `.true.` where the elements of `A` are greater than 0 and `.false.` elsewhere. Logical expressions of this kind may be used in a masked assignment statement, performing the specified assignment only where the mask is true.

To accompany this new notational convenience, most of the **intrinsic** functions act elementally on arrays, so that an expression such as `log(A)` applies the scalar log function to each element in `A` (in an unspecified order).

Intrinsic functions are defined by the language and are therefore present in any standard-conforming implementation.

Arrays may now be dynamically allocated at run-time using *pointers* or *allocatable* arrays.

```
program AllocatableMatrices
   real, allocatable :: A( :, : )
   integer n
   print *, 'Enter an integer for the array dimension: '
   read *, n

   ! Dynamically allocate an n x n matrix.
   allocate( A( n, n ) )
   call random_number( A )   ! Fill A with [0,1) random reals.

   ! Masked assignment.
   where ( A /= 0.0 )
      A = 1.0 / A
   elsewhere
      A = -1.0
   end where

   print *, 'A = '
   print *, A
end program AllocatableMatrices
```

Automatic arrays are also properly handled and are created on entry to a subprogram and destroyed upon exit. Lack of this feature in Fortran 77 was a severe limitation and often required that scratch variables be passed in the argument list. Functions may also be array-valued, as can be seen from such assignments as `A = log(B)`.

1.3 Modules

Common blocks in Fortran 77 were the only portable means of achieving global access of data throughout a collection of subprograms. This is unsafe, error-prone, and encourages bad programming practices in general. Fortran 90 provides a new program unit, a *module*, that replaces the common block and also provides many other features that allow modularization and data hiding, key concepts in developing large, maintainable numerical code.

Modules consist of a set of declarations and **module procedures** that are grouped under a single global name available for access in any other program unit via the `use` statement. **Interfaces** to the contained module procedures are explicit and permit compile time type-checking in all program units that use the

Module procedures are subprograms defined within a module.

An *interface* describes a subprogram, its attributes, and the attributes of its arguments, to the calling program or subprogram.

module. Visibility of items in a module may be restricted by using the `private` attribute. The `public` attribute is also available. Those identifiers not declared `private` in a module implicitly have the `public` attribute.

```
module TypicalModule
   private swap   ! Make swap visible only within this module.

   contains

   subroutine order( x, y )   ! Public by default.
      integer, intent( inout ) :: x, y

      if ( abs( x ) < abs( y ) )  call swap( x, y )
   end subroutine order


   subroutine swap( x, y )
      integer, intent( inout ) :: x, y
      integer tmp

      tmp = x; x = y; y = tmp ! Swap x and y.
   end subroutine swap
end module TypicalModule

program UseTypicalModule
   use TypicalModule

   ! Declare and initialize x and y.
   integer :: x = 10, y = 20

   print *, x, y
   call order( x, y )
   print *, x, y
end program UseTypicalModule
```

1.4 Derived Types and Generic Functions

Derived or user-defined types, similar to records or structures in other languages, are available in Fortran 90. Derived types are built from the intrinsic types or other derived types and allow the creation of data types that behave as if they were intrinsic types. **Generic functions**, another new feature in Fortran 90, help to make the support for derived data types complete. A generic function such as '`+`' may be extended to operate directly on a derived type, combining the notational convenience and clarity of operators with the abstraction of derived types. The module and program below illustrate the extension of the generic operator '`+`', to operate on a derived type, `interval`.

Generic functions are functions with the same name which behave differently based on the types of the arguments received. An example is the '`+`' operator, which operates on integers as well as floating point numbers.

```
module IntervalArithmetic
   type interval
      real a  ! Left endpoint
      real b  ! Right endpoint
   end type interval
```

```
   interface operator (+)
      module procedure addIntervals
   end interface

   contains

   function addIntervals( first, second )
      type( interval ) addIntervals
      type( interval ), intent( in ) :: first, second

      ! Numerically, the left and right endpoints of the interval
      !  sum should be rounded down and up, respectively, to
      !  ensure that numbers in the two intervals are also in the
      !  sum.  This has been omitted to simplify the example.

      addIntervals = interval( first%a + second%a, &
                               first%b + second%b )
   end function addIntervals
end module IntervalArithmetic

program Intervals
   use IntervalArithmetic

   type( interval ) :: x = interval( 1.0, 2.0 )
   type( interval ) :: y = interval( 3.0, 4.0 )
   type( interval ) z

   z = x + y
   print *, 'Interval sum: (', z%a, ',', z%b, ').'
end program Intervals
```

1.5 Pointers and Dynamic Data Structures

Fortran 90 contains 3 types of dynamic data: allocatable arrays, automatic data objects, and pointers. Allocatable arrays were described briefly in section 1.2 and apply only to arrays. Automatic data objects consist of those objects that are created on entry to a subprogram and destroyed upon exit. They are either arrays or scalar or array character entities where either the array bounds or character lengths are specified with an expression whose value can only be determined on entry to a subprogram. Pointers may be used with scalar or array quantities of any type and are used to construct dynamic structures such as linked lists and trees.

The following program illustrates how a dynamic data structure can be declared and manipulated.

```
program LinkedList
   type node
      real data
      type( node ), pointer :: next
   end type node
```

```
   type( node ), pointer :: list, current

   nullify( list )   ! Initialize list to point to no target.

   ! Place two elements in the list.
   allocate( list )   ! Reserve space for first node.
   call random_number( list%data )   ! Initialize data portion.
   allocate( list%next )   ! Reserve space for second node.
   call random_number( list%next%data )   ! Initialize data portion.
   nullify( list%next%next )   ! Initialize next to point to no target.

   ! Output the list.
   current => list   ! Assign target of list to target of current.
   do while ( associated( current ) )
      print *, current%data
      current => current%next
   end do

end program LinkedList
```

1.6 Parameterized Data Types

Portability of numerical code has long been difficult, primarily due to differences in the word sizes of the computers on which the code is run. Fortran 90 introduces parameterized types, increasing portability of software from machine to machine. This is done using *kind values*, constants associated with an intrinsic type such as `integer` or `real`. Parameterization of kind values allows precision changes by changing a single constant in the program. Several intrinsic functions are provided to select kind values based on the range and precision desired and inquire about a variable's precision characteristics in a portable way.

```
module Precision
   ! Define Q to be the kind number corresponding to at least 10
   ! decimal digits with a decimal exponent of at least 30.
   integer, parameter :: Q = selected_real_kind( 10, 30 )
end module Precision

module Swapping
   use Precision

   contains

   subroutine swap( x, y )
      real( Q ), intent( inout ) :: x, y
      real( Q ) tmp

      tmp = x; x = y; y = tmp
   end subroutine swap
end module Swapping

program Portable
   use Precision
```

```
   use Swapping

   ! Declare and initialize a and b using constants with kind value
   !  given by Q in the Precision module.
   real( Q ) :: a = 1.0_Q, b = 2.0_Q

   print *, a, b
   call swap( a, b )
   print *, a, b
end program Portable
```

1.7 Numerical Inquiry and Manipulation Functions

Fortran 90 introduces several intrinsic functions to inquire about machine dependent characteristics of an integer or real. For example, the inquiry function, `huge`, can be used to find the largest machine representable number for an integer or real value. The integer model used by these inquiry functions is[1]

$$i = s \sum_{k=0}^{q-1} d_k r^k$$

where

- i is the integer value
- s is the sign (+1 or -1)
- r is the radix ($r > 1$)
- q is the number of digits ($q > 0$)
- d_k is the kth digit, $0 \leq d_k < r$.

The floating-point model used by the inquiry functions is

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}$$

where

- x is the real value
- s is the sign (+1 or -1)
- b is the base ($b > 1$)
- e is the exponent
- p is the number of mantissa digits ($p > 1$)
- f_k is the kth digit, $0 \leq f_k < b$, $f_1 = 0 \Rightarrow f_k = 0 \, \forall \, k$.

Table 1 lists intrinsic functions that inquire about the numerical environment. Table 2 lists intrinsic functions that manipulate the numerical characteristics of variables in the real model. An important feature of all of these intrinsic functions is that they are generic and may be used to obtain information about any *kind* of integer or real supported by the Fortran 90 implementation.

[1] Information on the integer and floating-point models, as well as the following tables is taken from chapter 13 of [1].

Function	Description
`digits(x)`	q for an integer argument, p for a real argument
`epsilon(x)`	b^{1-p} for a real argument
`huge(x)`	Largest in the integer or real model
`minexponent(x)`	Minimum value of e in the real model
`maxexponent(x)`	Maximum value of e in the real model
`precision(x)`	Decimal precision (real or complex)
`radix(x)`	The base b of the integer or real model
`range(x)`	Decimal exponent range (real, complex, or integer)
`tiny(x)`	Smallest positive value in the real model

Table 1: Numeric Inquiry Functions

Function	Description
`exponent(x)`	Value of e in the real model
`fraction(x)`	Fractional part in the real model
`nearest(x)`	Nearest processor number in a given direction
`rrspacing(x)`	Reciprocal of relative spacing near argument
`set_exponent(x)`	Set the value of e to a specified value
`spacing(x)`	Model absolute spacing near the argument

Table 2: Numeric Manipulation Functions

2 Examples and Exercises

2.1 Getting Started

The Fortran 90 compiler is invoked, similar to any other compiler, by issuing a command with the name of the compiler, such as `f90 <prog>`, at the shell prompt, to invoke the NAG Fortran 90 compiler or `xlf90 <prog> to invoke the IBM Fortran 90 compiler`. The NAG Fortran 90 compiler has several naming conventions that are useful. If the filename's extension ends with `.f`, it is assumed to be Fortran 77 source using its fixed source form. Using `.f90` as the extension, it is assumed that the source file uses the free source form. These can be overridden using compiler switches and options. See the manual page for the `f90` command for more details.

As a first attempt at using the Fortran 90 compiler, try the following exercise.

Exercise 2.1 *Enter, compile, and run the* `Hello` *program given below. Call the file* `hello.f90`*. It can be compiled by issuing the command* `f90 hello` *at the shell prompt. Note the use of Fortran 90's free source form.*

```
program Hello
   print *, 'Hello, world'
end program Hello
```

2.2 Basic Fortran Programs

Declarations of variables and parameters in Fortran 77 have been changed slightly in the new standard to allow for more concise specification of an identifier's attributes. The Fortran 77 declaration,

```
integer N
parameter( N = 10 )
```

now can be written more concisely as

```
integer, parameter :: N = 10
```

As shown in the above declaration, constants and variables alike may be initialized at the time of declaration.[2]

Several new control constructs, many implemented as extensions to Fortran 77, are now officially part of Fortran 90. These include `do ... enddo`, `do while ( condition ) ... enddo`, and `select case ( case expr ) ... end case`. The `do while ( condition ) ... end do` construct is illustrated in next exercise. Note that a line number and the `continue` statement are no longer necessary to mark the end of a `do` loop.

Fortran 90 introduces synonyms for several relational operators, making them more natural and similar to relational operators in other languages. These

Fortran 77	Fortran 90 Synonym
.lt.	<
.le.	<=
.eq.	==
.gt.	>
.ge.	>=
.ne.	/=

Table 3: Fortran 90 Relational Operators

are summarized in Table 3. Either style is acceptable in Fortran 90 and the styles may be used interchangeably in either source form.

Non-advancing I/O is another new feature of Fortran 90 illustrated in the next exercise. It permits several `write` statements to output to a single line, something that was not possible in Fortran 77. The specification of the format for the output has also been improved, eliminating the need for line numbers and a separate `format` statement.

Comments may be placed anywhere on the source line, preceded with a '`!`' character. Trailing comments are also possible. All of these features are illustrated in the following exercise.

Exercise 2.2 *Write a Celsius-Fahrenheit conversion table using the following program as a guide. Note that this program outputs a Fahrenheit-Celsius table.*

```
program Fahrenheit_Celsius

! Output a table of Fahrenheit and equivalent Celsius temperatures
! from low to high in steps of step.

   real, parameter :: low = 0.0
   real, parameter :: high = 100.0
   real, parameter :: step = 10.0

   real f, c   ! Fahrenheit and Celsius temperatures, respectively.

   f = low

   do while ( f <= high )
      write( *, fmt = "(F8.3)", advance = "no" ) f
      c = 5.0 * ( f - 32 ) / 9.0
      write( *, fmt = "(F8.3)" ) c

      f = f + step  ! Advance the Fahrenheit temperature.
   end do
end program Fahrenheit_Celsius
```

[2] The double colon, '`::`', is necessary when more than one attribute is given for a variable, or the variable is to be initialized.

2.3 Internal Subprograms

In Fortran 77, all subprograms are external with the exception of statement functions. Internal subprograms are now possible under Fortran 90 and achieve an effect similar to Fortran 77's statement functions. They are visible only within the containing program and have an explicit interface, guarding against type mismatches in calls to the subprogram. Internal subprograms must be separated from the main program by the `contains` statement. An example illustrating an internal subprogram is given below.

```
program Triangle
   real a, b, c

   print *, 'Enter the lengths of the three sides of the triangle'
   read *, a, b, c
   print *, 'Triangle''s area:  ', triangleArea( a, b, c )

   contains

   function triangleArea( a, b, c )
      real triangleArea
      real, intent( in ) :: a, b, c

      real theta
      real height

      theta = acos( ( a**2 + b**2 - c**2 ) / ( 2.0 * a * b ) )
      height = a * sin( theta )

      triangleArea = 0.5 * b * height

   end function triangleArea

end program Triangle
```

Exercise 2.3 *Rewrite the Celsius-Fahrenheit program of the previous exercise to use an internal function to calculate the Fahrenheit temperature.*

2.4 Arrays

Arrays and array operations have undergone extensive change in the new Fortran standard. In Fortran 90, it is possible to treat an array as a single object. This permits array-valued expressions such as `C = A + B` without the need for `do` loops that are required in Fortran 77 to process the elements of the arrays one at a time. Although such statements are notationally convenient and offer a more natural form of expression, they are also important in utilizing the high computational speeds of parallel and vector computers.[3] Functions may now

[3]Another important feature of array-valued expressions is that there is no defined order in which the operations must be done. The operations may be performed in any order or simultaneously.

be array-valued, which was impossible to achieve in Fortran 77. Most intrinsic functions have been extended and act elementally on arrays, as do the intrinsic operators, such as '+' above.[4] Array sections are obtained using a syntax similar to *Matlab*. `A( :, i )` is the ith column of A. The ':' represents all elements in the extent of the particular dimension. `A( 2:4, 3:5 )` is the 3×3 array obtained from rows 2 through 4 and columns 3 through 5 of A. A *stride* may also be specified, achieving an effect similar to the *step* of a `do` loop. For example, `A( 2:10:2, 2:10 )` is the 5×9 array obtained from rows 2, 4, 6, 8, and 10 and columns 2 through 10 of A.

Passing arrays to subprograms is another area of improvement in the new standard. In Fortran 77, only the extents in the last dimension can be assumed in a subprogram. This often requires extending the argument list of a subprogram to include the extents of each dimension of the array. Fortran 90 supports *assumed-shape* arrays in dummy arguments in a subprogram. The extents can be determined by the subprogram through the use of the new intrinsic function `size`.[5] These array processing features of Fortran 90 are illustrated by the `MatrixVector` program below.

```
program MatrixVector
   implicit none
   integer, parameter :: N = 3

   real, dimension( N, N ) :: A
   real, dimension( N ) :: b, c

   ! Fill A and b with random entries.
   call random_number( A )
   call random_number( b )

   ! Compute the matrix-vector product, A*b.
   c = matrixVectorMultiply( A, b )

   print *, 'The matrix-vector product is ', c

   contains

   function matrixVectorMultiply( A, b ) result( c )
      implicit none

      ! Assume the shape of A and b.
```

[4] Most binary operations operate elementally on array operands, requiring that the two array operands be *conformable*, that is, the same size and shape. For example, the binary operator '*' forms an element-by-element product of two matrices. Scalars may also be used in such operations, as they are first broadcast to a conformable array before the elemental operation is performed.

[5] The rank of an array is fixed at compile-time and may not be assumed by a subprogram. The `size` function when called with only the array name as an argument returns the total number of elements in the array. When called with the array name and an integer denoting the dimension, it returns the number of elements along that dimension. For example, for an $N \times M$ dimensional array, `A`, `size(A,1)` returns `N` and `size(A,2)` returns `M`.

```
      real, dimension( :, : ), intent( in ) :: A
      real, dimension( : ), intent( in ) :: b
      real, dimension( size( b ) ) :: c

      integer N
      integer i

      N = size( b )

      c = 0.0

      do i = 1, N
         c = c + b( i ) * A( :, i )
      end do
   end function matrixVectorMultiply

end program MatrixVector
```

Exercise 2.4 *Derive a formula for matrix-matrix multiplication that forms the matrix product $C = AB$ by accessing A by columns. It is quite similar to the matrix-vector multiplication given in the previous example. Such an algorithm is column-oriented, and exploits Fortran's array storage conventions. Write and test a program that uses this column-oriented algorithm to obtain the matrix-matrix product. To check your results, use the intrinsic function,* `matmul`*. How is this column-oriented multiplication algorithm similar to the matrix-vector multiplication algorithm in the previous example?*

Exercise 2.5 *Similar to the previous exercise, derive a formula for matrix-matrix multiplication that forms the matrix product $C = AB$ by accessing B by rows. Such an algorithm is row-oriented, and is the worst matrix multiplication algorithm for Fortran. Write and test a program that uses this row-oriented algorithm to obtain the matrix-matrix product. To check your results, use the intrinsic function,* `matmul`.

Exercise 2.6 *The traditional approach to matrix multiplication forms the matrix product $C = AB$ utilizing a dot-product operation between a row of A and a column of B. Formulate such an algorithm and write and test a program that uses this algorithm to obtain the matrix-matrix product. Write a supporting* `dotProduct`[6] *function to form each element in the matrix product and call it from within your matrix multiplication function. You will want to make use of array sections to pass your* `dotProduct` *function a row of A and a column of B. Use the intrinsic function,* `matmul`*, or your previous algorithms, to test this dot product algorithm.*

2.5 Modules

Modules represent a tremendous improvement in program readability and main-

[6] Write your own function rather than using the intrinsic function `dot_product`.

tainability over Fortran 77's `common` block. However, modules offer much more functionality than the global access of data provided by the `common` block. A module can be used to group a set of related declarations and **module procedures**, under a single global name, providing a means of global access to constants, variables, user-defined types, and other functions and subroutines. The contents of a module may be made available to any program unit via the `use` statement. Data hiding and encapsulation are supported through the `private` and `public` attributes. Those items declared `private` are available only within the module and are hidden from any subprogram using the module. By default, all items within a module are `public`. The `contains` statement in a module marks the beginning of one or more module procedures, just as it was used to mark the beginning of one or more internal subprograms in section 2.3.

Subprograms declared within a module are called *module procedures.*

The `matrixVectorMultiply` function in the previous example could be placed in a module and used in any program needing to perform matrix-vector multiplication. An example of such a module is given below. One significant advantage of module procedures is their explicit interface—the Fortran 90 compiler can detect type mismatches in calls to subprograms within a module.

```
module MatrixVectorOperations

   integer, parameter :: N = 3   ! A global constant.

   contains  ! Module procedure definitions appear below.

   function matrixVectorMultiply( A, b ) result( c )
      implicit none

      ! Assume the shape of A and b.
      real, dimension( :, : ), intent( in ) :: A
      real, dimension( : ), intent( in ) :: b
      real, dimension( size( b ) ) :: c

      integer N
      integer i

      N = size( b )

      c = 0.0

      do i = 1, N
         c = c + b( i ) * A( :, i )
      end do
   end function matrixVectorMultiply

end module MatrixVectorOperations

program MatrixVector
   use MatrixVectorOperations
   implicit none

   real, dimension( N, N ) :: A
```

```
   real, dimension( N ) :: b, c

   ! Fill A and b with random entries.
   call random_number( A )
   call random_number( b )

   ! Compute the matrix-vector product, A*b.
   c = matrixVectorMultiply( A, b )

   print *, 'The matrix-vector product is ', c

end program MatrixVector
```

Exercise 2.7 *Create a* `MatrixMatrixOperations` *module and place the column-oriented matrix multiplication function of exercise 2.4 in this module. Test it by using it with a main program.*

2.6 Interfaces and Generic Subprograms

Interfaces refer to the how much knowledge the compiler has about an procedure during compilation. If the interface is *explicit*, then the compiler can verify that the subprogram is being called correctly. If, however, the interface is *implicit*, then the Fortran 90 compiler has no information about the types and number of the subprogram's arguments or the return value of the result for a function. (Implicit typing is used in the calling program to determine the return result of a function if no declaration is given.) Consequently, no type-checking can be done to verify that a subprogram has been called correctly. Implicit interfaces are all that are available in Fortran 77.

Subprograms such as module procedures and internal functions have an explicit interface by default, and no explicit *interface block* is necessary. External subprograms have an implicit interface by default, and an interface block is necessary to specify an explicit interface of an external subprogram; as mentioned above, this allows type-checking of actual and formal arguments in a reference to a subprogram. Examples of interface blocks for two external functions, `f` and `g`, are given below.

```
interface
   function f( x )
      real f
      real, intent( in ) :: x
   end function f
   function g( y )
      integer g
      integer, intent( in ) :: y
   end function g
end interface
```

Interfaces are also necessary to define a *generic subprogram*. Generic subprograms should be familiar from Fortran 77 intrinsics such as `sin` or operators such as '+'. These intrinsic functions were special cases in Fortran 77 and were **overloaded** to work on a variety of argument types—`sin( x )` will properly compute the sine of its argument, whether `x` is single or double precision, real or complex. In Fortran 90, user-defined subprograms can be generic in the same sense. Generic functions and subroutines may be defined, similar to any other subprogram, although the interface must be explicit. The usual way to define such a generic function is to place it in a module as in the example below.

Overloading refers to using a generic name to specify a function whose behavior is dependent upon the types of its arguments.

```
module RationalArithmetic
   type rational
      integer n, d   ! Numerator and denominator.
   end type rational

   interface operator (*)
      module procedure integerRationalMultiply, &
                       rationalIntegerMultiply
   end interface

   contains

   function integerRationalMultiply( i, r )
      type( rational ) integerRationalMultiply
      integer, intent( in ) :: i
      type( rational ), intent( in ) :: r

      integerRationalMultiply = rational( i * r%n, r%d )
   end function integerRationalMultiply

   function rationalIntegerMultiply( r, i )
      type( rational ) rationalIntegerMultiply
      type( rational ), intent( in ) :: r
      integer, intent( in ) :: i

      rationalIntegerMultiply = rational( i * r%n, r%d )
   end function rationalIntegerMultiply

end module RationalArithmetic
```

This use of operators on derived types is a much more natural form of expression for many mathematical objects that can be modeled with user-defined types. The overloading of function names and operators is handled completely by the compiler. A reference to the '+' operator in the example above causes the compiler to insert a call to the appropriate function based on the types of arguments in the particular call. This function substitution can be completely determined at compile time and incurs no run-time overhead.

Exercise 2.8 *Why are two different functions needed to perform integer-rational multiplication? Write and test a program to use this module, using the overloaded operator '*'.*

2.7 Recursive Subprograms

Another area in which Fortran 77 has been extended is recursion. Although not possible in Fortran 77, Fortran 90 supports recursion. If a subprogram calls itself, directly or indirectly, the keyword `recursive` must appear in the subprogram statement.[7] Recursive functions must also declare a `result` variable to avoid ambiguity with array-valued functions that are directly recursive. The result variable is used to hold the function result for each function invocation; the function name is used to invoke the function itself. Consequently, the recursive function's name should never appear on the left side of an assignment statement. An example of a recursive factorial function is shown below.

```
recursive function factorial( n ) result( f )
   integer f
   integer, intent( in ) :: n

   if ( n <= 0 ) then
      f = 1
   else
      f = n * factorial( n-1 )
   end if
end function factorial
```

Exercise 2.9 *Enhance the* `RationalArithmetic` *module in the previous exercise to use a greatest common divisor function to cancel common divisors from the integer multiplier and the denominator of the rational number. Call the function* `gcd`*, and make it recursive, based on the following relationship, valid for* $0 \le m < n$.

$$\begin{aligned} \gcd(0,n) &= n; \\ \gcd(m,n) &= \gcd(n \bmod m, m), m > 0. \end{aligned}$$

Verify correct operation of the `gcd` *function by testing it using a main program.*

Exercise 2.10 *Using the* `gcd` *function written in the previous exercise, extend the '*`+`*' operator to operate correctly on rational numbers. Verify that your program is working correctly, that is, it forms the sum of two rational numbers whose numerator and denominator are relatively prime. An efficient implementation should cancel the greatest common divisor in a way that keeps the intermediate integer products as small as possible.*

Exercise 2.11 *Using the results of the previous exercise, you should be able to extend the '*`-`*' operator relatively easily. Write another set of functions to extend this operator and verify that your program works correctly as discussed in the previous exercise.*

[7] The `recursive` keyword is required for the benefit of the compiler and may help with the optimization of procedure calls.

Exercise 2.12 *Relational operators may also be extended to operate on user-defined types. Extend the '`.eq.`' operator to properly compare two rational numbers. Recall that '`==`' is a synonym for the '`.eq.`' operator.*

2.8 Dynamic Data Structures

Lack of dynamic data structures is another shortcoming of Fortran 77 that has been overcome in Fortran 90. Linked lists, trees, graphs, and other dynamic data structures that are allocated at runtime can be constructed using some of Fortran 90's new capabilities. The following program indicates the basic features of Fortran 90's pointers.

```
program TryPointers
   integer, pointer :: p, q
   integer, target :: n
   integer m

   n = 5

   p => n
   q => p

   allocate( p )
   p = 4

   m = p + q + n

   print *, "m = ", m
end program TryPointers
```

The program above illustrates many key concepts in the design of Fortran 90 pointers.

- The `pointer` attribute is used to identify variables that serve as descriptors for other data objects only. In the program above, `p` and `q` are "pointers to integers".

- The variable `n` is declared with the `target` attribute. This indicates that `n` may serve as a *target* to an integer pointer such as `p` or `q`. A target is an object that may be referenced via a pointer. The target of a pointer is very restrictive—only variables with the `target` or `pointer` attributes may be referenced via a pointer.[8]

- The statement `p => n` is called a *pointer assignment statement*, and is used to *associate* the target, `n`, with the pointer, `p`. This is possible because

[8] This restriction can aid the compiler in optimizing the code. The `target` attribute informs the compiler of all variables that can serve as pointer targets. All other variables cannot be referenced via a pointer and do not suffer from the side effects of pointer references.

n has been declared with the target attribute.[9] In effect, p is an alias for n and may be used just as n is used (p can be thought of as "pointing to" n).[10]

- The following statement, `q => p`, is different. In the previous pointer assignment statement, a pointer is associated with a non-pointer variable, n. However, this pointer assignment statement contains variables that are pointers on both the left and right hand sides. In this case, q is associated with the target of p, namely n, rather than p itself.[11]

- The `allocate` statement is used to dynamically allocate space for a target and associate the pointer with the target. Thus, `allocate( p )` reserves space for an integer target that is associated with p. At this time, the contents of the target are not defined. The previous target of p, n, is unaffected. Similarly, because q is not associated with p, the target of q is unaffected.

- The statement `p = 4` defines the target of p to contain the integer 4 and illustrates an important characteristic of Fortran 90's pointers—there is *no* dereference operator. Instead, the pointers are resolved to their targets before the assignment is made. This occurs in any expression involving pointers with the exception of the pointer assignment statement described above.

- As mentioned above, pointers are resolved to their targets in any expression not involving pointer assignment. Consequently, the statement `m = p + q + n` uses the values of the targets of p and q and the value of n to evaluate the expression. Because the values of the targets of p and q are 4 and 5, respectively, and n is 5, their sum, 14, is assigned to the the integer m.

A more complete example of a dynamic data structure is given below which manipulates a linked list of real numbers. Elements are allocated as needed and deallocated after use.

```
program LinkedList
   type node
      real data
      type( node ), pointer :: next
   end type node
```

[9] The other integer variable m declared in the program does not have the target attribute. Consequently, it may not be associated with a pointer.

[10] A Fortran 90 pointer has three possible states: associated, disassociated, and undefined. Initially, a pointer's association status is undefined. It may be associated using pointer assignment or the allocate statement. It may be disassociated using the nullify statement or by assigning it a disassociated pointer through pointer assignment.

[11] This immediately implies that pointers cannot point to other pointers.

```
   type( node ), pointer :: list, current, previous
   integer, parameter :: N = 10

   nullify( list )   ! Initialize list to point to no target.

   ! Add the 1st element as a special case.
   if ( N > 0 ) then
      allocate( list )
      nullify( list%next )
      call random_number( list%data )
   end if

   current => list

   ! Add N random numbers to the list.
   do i = 2, N
      allocate( current%next )
      nullify( current%next%next )
      call random_number( current%next%data )
      current => current%next
   end do

   ! Output the list, deallocating them after use.

   print *, 'List elements are:'

   current => list
   do while ( associated( current ) )
      print *, current%data
      previous => current
      current => current%next
      deallocate( previous )
   end do

end program LinkedList
```

Exercise 2.13 *Rewrite the* `do i = 2, N` *loop to allocate a node of the list using a temporary pointer. Initialize the* `data` *and* `next` *components of the node and link the node to the existing list as the last step in the loop.*

Exercise 2.14 *A linked list can be used to model an integer of arbitrary length. The following program implements unsigned extended integer arithmetic using this integer model, where the head of the linked list points to the least significant digit in the extended integer. For example, the extended integer 987654321 is represented as the list* $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to$ *where the last pointer in the integer is nullified to mark the end of the list.*

```
module ExtendedIntegers

   ! A kind value representing a single digit only.
   integer, parameter :: Q = selected_int_kind( 1 )
   private Q
```

```
type UnsignedExtendedInteger
   integer( Q ) digit
   type( UnsignedExtendedInteger ), pointer :: next
end type UnsignedExtendedInteger

type ExtendedInteger
   type( UnsignedExtendedInteger ), pointer :: number
end type ExtendedInteger

interface operator (+)
   module procedure addExtendedInteger
end interface

contains

function addExtendedInteger( m, n )
   type( ExtendedInteger ) addExtendedInteger
   type( ExtendedInteger ), intent( in ) :: m, n

   type( ExtendedInteger ) sum
   type( UnsignedExtendedInteger ), pointer :: current1, current2
   type( UnsignedExtendedInteger ), pointer :: current, previous
   integer( Q ) carry
   integer digitSum

   ! Allocate space for the number component of sum.
   allocate( sum%number )
   nullify( sum%number%next )

   ! Add the first two digits of m and n as a special case.

   ! Associate current1 and current2 with the first digits of the numbers
   !  of m and n.
   current1 => m%number
   current2 => n%number

   digitSum = current1%digit + current2%digit

   ! Check for carry
   if ( digitSum > 9 ) then
      digitSum = digitSum - 10
      carry = 1
   else
      carry = 0
   end if

   ! Assign the first digit into the sum.
   sum%number%digit = digitSum

   previous => sum%number
   current1 => current1%next
   current2 => current2%next

   ! Begin the general addition of m and n.  m and n should be
   !  have their ends marked with null pointers.
```

```
do while ( associated( current1 ) .and. associated( current2 ) )
   digitSum = current1%digit + current2%digit + carry

   ! Check for carry
   if ( digitSum > 9 ) then
      digitSum = digitSum - 10
      carry = 1
   else
      carry = 0
   end if

   ! Assign current digit into sum.
   allocate( previous%next )
   nullify( previous%next%next )
   previous%next%digit = digitSum

   previous => previous%next
   current1 => current1%next
   current2 => current2%next
end do

! current1, current2, or both have been exhausted.  Continue with current.
if ( associated( current1 ) ) then
   current => current1
else
   current => current2
end if

do while ( associated( current ) )
   digitSum = current%digit + carry

   ! Check for carry
   if ( digitSum > 9 ) then
      digitSum = digitSum - 10
      carry = 1
   else
      carry = 0
   end if

   ! Assign current digit into sum.
   allocate( previous%next )
   nullify( previous%next%next )
   previous%next%digit = digitSum

   previous => previous%next
   current => current%next
end do

! Check for a carry that has propagated to the most significant
!  place.
if ( carry /= 0 ) then
   ! Assign carry digit into sum.
   allocate( previous%next )
   nullify( previous%next%next )
   previous%next%digit = carry
```

```
   end if

   ! Prepare to return sum.
   addExtendedInteger = sum
end function addExtendedInteger

subroutine readExtendedInteger( n )
   type( ExtendedInteger ), intent( out ) :: n

   type( UnsignedExtendedInteger ), pointer :: new
   character ch
   integer( Q ) value


   nullify( n%number )   ! No list initially.


   do while (.true.)
      read(*, fmt="(A)", advance="no", eor=100) ch

      ! Determine the numerical value of the character.
      select case ( ch )
         case( '0' )
            value = 0
         case( '1' )
            value = 1
         case( '2' )
            value = 2
         case( '3' )
            value = 3
         case( '4' )
            value = 4
         case( '5' )
            value = 5
         case( '6' )
            value = 6
         case( '7' )
            value = 7
         case( '8' )
            value = 8
         case( '9' )
            value = 9
         case default
            print * , 'Input error in reading extended integer'
            stop
      end select

      ! Allocate a new node for the digit
      allocate( new )
      new%digit = value

      if ( .not. associated( n%number ) ) then
         ! List not yet started.
         nullify( new%next )
      else
```

```
         ! List already started.
         new%next => n%number
      end if

      n%number => new
   enddo

100 end subroutine readExtendedInteger

end module ExtendedIntegers
```

A subroutine to input unsigned extended integers is included in the module. Write a subroutine to output an extended integer based on the model of representation described above. Write a program to test your function and the `ExtendedIntegers` module.

2.9 Optional and Keyword Arguments

Optional arguments permit a subprogram to accept a default value for missing arguments in a call. In numerical computing, this is most useful for specifying a tolerance. If specified, the value may override the default tolerance; otherwise, the default tolerance is used. Because any argument in the argument list of a subprogram may be optional, there may be a problem matching actual and formal arguments in a call to a subprogram with optional arguments. In this case, *keyword arguments* must be used to establish unambiguously the correspondence between actual and formal arguments. The `RootFinders` module below illustrates some of these features.

```
module RootFinders
   ! Maximum error permitted in the approximation of a root.
   real, parameter :: DEFAULT_TOLERANCE = epsilon( 1.0 )

   ! Restrict the visibility of these functions to this module.
   private secant, newton

   contains

   ! Use the secant method to find a root of f if df, the
   !  derivative of f, is unavailable, otherwise, use Newton's
   !  method.  a and b are used as a starting interval for
   !  the secant method.  The average of a and b is used as
   !  the initial guess for Newton's method.

   function findRoot( a, b, f, df, tolerance )
      implicit none
      real findRoot
      real, intent( in ) :: a, b
      real, optional, intent( in ) :: tolerance
      interface
         function f( x )
            real f
```

```
         real, intent( in ) :: x
      end function f
      function df( x )
         real df
         real, intent( in ) :: x
      end function df
   end interface
   optional df

   real tol

   ! Initialize tol.
   if ( present( tolerance ) ) then
      tol = tolerance
   else
      tol = DEFAULT_TOLERANCE
   end if

   ! Select the root-finding method.
   if ( present( df ) ) then   ! Use Newton's method.
      findRoot = newton( (a+b)/2, f, df, tol )
   else   ! Use secant method.
      findRoot = secant( a, b, f, tol )
   end if
end function findRoot

recursive function secant( a, b, f, tol ) result( root )
   implicit none
   real root
   real, intent( in ) :: a, b, tol
   interface
      function f( x )
         real f
         real, intent( in ) :: x
      end function f
   end interface

   real c   ! The x-intercept of the secant line.
   real fa, fb, fc   ! f(a), f(b), and f(c), respectively.

   ! Initialize fa and fb.
   fa = f( a ); fb = f( b )

   ! Compute c, the x-intercept of the secant line given by
   !  the two points, (a, f(a)) and (b, f(b)).
   c = a - fa * ( ( b - a ) / ( fb - fa ) )

   ! Compute the value of the function at this point.
   fc = f( c )

   ! Check for a sufficient root at c.  This could cause an
   !  infinite loop if the round-off error in the evaluation
   !  of f( c ) exceeds the tolerance.

   if ( ( abs( fc ) <= tol ) .or. ( ( abs( c - b ) <= tol ) ) ) then   ! Root found.
```

```
         root = c
      else   ! Go again.
         ! Make sure the function is non-increasing in absolute
         !  value for each recursive call of secant.

         if ( abs( fa ) < abs( fb ) ) then   ! Use a and c.
            root = secant( a, c, f, tol )
         else   ! Use b and c.
            root = secant( b, c, f, tol )
         end if
      end if
   end function secant

   recursive function newton( guess, f, df, tol ) result( root )
      implicit none
      real root
      real, intent( in ) :: guess, tol
      interface
         function f( x )
            real f
            real, intent( in ) :: x
         end function f

         function df( x )
            real df
            real, intent( in ) :: x
         end function df
      end interface

      real fGuess, dfGuess   ! f(guess), df(guess), respectively.
      real newGuess

      ! Calculate df(guess) and f(guess).
      fGuess = f( guess ); dfGuess = df( guess )

      ! Check for a sufficient root at c.  This could cause an
      !  infinite loop if the round-off error in the evaluation
      !  of f( c ) exceeds the tolerance.

      if ( abs( fGuess ) <= tol ) then   ! Root found.
         root = guess
      else   ! Go again.
         newGuess = guess - fGuess / dfGuess
         root = newton( newGuess, f, df, tol )
      end if
   end function newton

end module RootFinders
```

The `findRoot` function defined above is quite convenient in its use of optional arguments. For example, `x = findRoot( a, b, g, dg )` uses the default tolerance and calls `newton`, due to the presence of the derivative, `dg`. `x = findRoot( a, b, g, dg, 1.0e-10 )` may be used to override this default tolerance. When the derivative of the function is not available, a call such as `x = findRoot( a, b, g )`, uses the default tolerance and calls `secant`, as no derivative is present. `x = findRoot( a, b, g, tolerance=1.0e-10 )` overrides the default tolerance and calls `secant`. Notice that overriding the default tolerance when the derivative is not passed requires using the keyword `tolerance`. If this keyword were not used, the fourth argument would be incorrectly paired with the formal argument `df`, resulting in a type mismatch. A test program for the `RootFinders` module is given below.

```
program Test
   use RootFinders
   implicit none
   real a, b
   real, parameter :: tol = 1.0e-6
   interface
      function f( x )
         real f
         real, intent( in ) :: x
      end function f

      function df( x )
         real df
         real, intent( in ) :: x
      end function df
   end interface

   print *, 'Enter left and right endpoints'
   read *, a, b
   print *, 'Newton:The root of f is ', findRoot( a, b, f, df )
   print *, 'Secant:The root of f is ', findRoot( a, b, f )

end program Test


function f( x )
   real f
   real, intent( in ) :: x

   f = x + exp( x )
end function f

function df( x )
   real df
   real, intent( in ) :: x

   df = 1 + exp( x )
end function df
```

Exercise 2.15 *Compile and run the test program and module given above. Then, rewrite* `secant` *and* `newton` *to perform their tasks non-recursively. Test your root finding functions with several functions. An efficient implementation should minimize the number of function evaluations necessary during the algorithm.*

Exercise 2.16 *Neither the secant method nor Newton's method are guaranteed to converge to a root of the function in a finite number of steps. Add another optional argument to* `findRoot`*, specifying the maximum number of iterations to perform. Make these changes to your iterative version of the* `RootFinders` *module. In adding this optional argument, consider its placement in the argument list so that the function may be conveniently called with or without including the argument in the call.*

2.10 Achieving Portability

Portability refers to the ease with which source code can be moved from machine to machine. A portable program requires little or no change to the source code when compiled and run on a different machine. Portability of numerical code is important for several reasons.

- Numerical libraries, such as Linpack or the NAG libraries, are available. If this code is not portable, it must be tailored to the particular implementation on which it is used.
- Access to high performance machines is sometimes limited. Often, a program is developed and tested on a small workstation, then ported to a high performance machine to run larger problems. Without portability, this is not possible.
- Many small workstations such as DEC, Sun, and IBM, are often available at a particular site on a local network. It is simply inconvenient to restrict code to a particular implementation.

Fortran 90 introduces several new mechanisms to aid in porting numerical code to other machines. The most difficult problem in porting numerical programs is in the portable selection of precision. Selecting the precision of a computation in a portable way was impossible with Fortran 77. However, with the introduction of *kind values* as well as intrinsic environmental inquiry functions for selecting and inquiring about precision, Fortran 90 programs should be much easier to port to other machines.

Kind values are integer constants that can be used to further specify the characteristics of an intrinsic type, such as `integer` or `real`. For example, `real( 2 )` selects a `real` type with kind value 2. Unfortunately, kind values are processor-dependent and are therefore not standardized. However, there are several portable ways to select kind values based on the precision desired. Two such functions are `selected_int_kind` and `selected_real_kind`. The following `Precision` module illustrates a portable means of selecting precision parametrically.

```
module Precision
   integer, parameter :: Q = selected_real_kind( 10, 10 )
end module Precision
```

The `selected_real_kind` function above selects the kind value corresponding to a real number with at least 10 decimal digits of precision and a decimal exponent range of at least 10 in magnitude. The `selected_int_kind` function is similar, and an expression such as `selected_int_kind( 10 )` selects the kind value corresponding to a integer number with magnitude in the range $(10^{-10}, 10^{10})$. Other environmental inquiry and manipulation functions are described in tables 1 and 2 in section 1. Advanced uses of these functions and other portability issues are described in section 3.

Exercise 2.17 *In the* `RootFinders` *module, add parameterized types to all variables, constants, and functions that were previously declared as* `real`*. Use the* `Precision` *module given above, so that the precision of the entire* `RootFinders` *module may be changed by changing a parameter in the* `Precision` *module.* [Hint: The `DEFAULT_TOLERANCE` in the `RootFinders` module should use the epsilon corresponding to the parameterized real type. Constants of a particular kind value may be specified by using the kind value as a suffix, preceded by the underscore, '_', character. Using the notation of the `Precision` module, the `DEFAULT_TOLERANCE` should be assigned the value `epsilon( 1.0_Q )`.[12]]

[12]Note that `epsilon` is an inquiry function which uses the type of its argument rather than the specific value.

3 Advanced Numerical Experiments

3.1 Interval Arithmetic

Interval arithmetic is a technique used to determine upper bounds for the absolute error in an algorithm, properly considering all roundoff errors in the calculation. It is based on the fact that the real number system modelled by a computer is effectively viewed as an interval with machine representable endpoints in which the exact result lies. All real numbers that enter into a numerical calculation, initial, intermediate, and final, are most often unknown. At best, an interval is known that contains the exact answer. Extending the arithmetic operations used in a numerical algorithm to operate on intervals produces intervals that are guaranteed to contain the exact solution. This type of analysis can be readily implemented in Fortran 90 with its support for derived types and generic functions and operators. It also illustrates several advanced numerical manipulation functions new to Fortran 90 that simplify the implementation and increase portability.

In section 1.4, the following module was used to illustrate generic functions applied to a derived type.

```
module IntervalArithmetic
   type interval
      real a  ! Left endpoint
      real b  ! Right endpoint
   end type interval

   interface operator (+)
      module procedure addIntervals
   end interface

   contains

   function addIntervals( first, second )
      type( interval ) addIntervals
      type( interval ), intent( in ) :: first, second

      ! Numerically, the left and right endpoints of the interval
      !  sum should be rounded down and up, respectively, to
      !  ensure that numbers in the two intervals are also in the
      !  sum.  This has been omitted to simplify the example.

      addIntervals = interval( first%a + second%a, &
                               first%b + second%b )
    end function addIntervals
end module IntervalArithmetic
```

As pointed out in the comments preceding the calculation of the interval sum, this implementation is simplistic and could give incorrect results if used in a rounding error analysis. An accurate approach is explained below.

Let M be the set of all machine representable reals and let $\oplus$ denote interval addition. Then, the interval sum, $[c_1, c_2] = [a_1, a_2] \oplus [b_1, b_2]$, must be the smallest interval containing the exact sums, $a_1 + b_1$ and $a_2 + b_2$, with machine representable endpoints, c_1 and c_2. More precisely,

$$\begin{aligned} c_1 &= \max \{s \in M | s \le a_1 + b_1\} \\ c_2 &= \min \{s \in M | s \ge a_2 + b_2\} \end{aligned}$$

where $c_1, c_2, a_1, a_2, b_1, b_2 \in M$, and the '+' operator represents exact addition without roundoff or constraints of finite precision.[13]

Exercise 3.1 *The remaining exercises in this section will focus on correcting the behavior of interval addition in the* `IntervalArithmetic` *module. Two necessary supporting functions are* `roundSumDown` *and* `roundSumUp`*, for properly rounding the left and right endpoints, respectively. Specifically, these functions should calculate the left and right endpoints of the interval sum using the procedure described in the text above. That is,*

$$\begin{aligned} \texttt{roundSumDown}(x, y) &= \max \{s \in M | s \le x + y\} \\ \texttt{roundSumUp}(x, y) &= \min \{s \in M | s \ge x + y\}. \end{aligned}$$

For ease of exposition, let x be the smaller number in magnitude, so that $|x| \le |y|$ and let $x + y$, $x \oplus_d y$, and $x \oplus_s y$ denote exact, double precision, and single precision addition operators, respectively.

Consider calculating the sum $x + y$ using both double and single precision, forming $x \oplus_d y$ and $x \oplus_s y$. What information can be obtained regarding their accuracy if these two sums are compared? How might this be used to properly round the single precision sum? Let `rSum` *be the single precision sum and* `dSum` *be the double precision sum. State a value to be returned by* `roundSumDown` *and* `roundSumUp` *for the following two cases:* `rSum` $<$ `dSum` *and* `rSum` $>$ `dSum`*. The case* `rSum` $=$ `dSum` *is treated in the next exercise.* [Hint: Fortran 90's intrinsic function `nearest` should be helpful.]

Exercise 3.2 *Continuing with the previous exercise, consider the case where* `rSum` $=$ `dSum`*. What two possibilities exist in this case?* [Hint: The most obvious possibility is that both `rSum` and `dSum` represent the exact sum, $x+y$. The other possibility occurs when two numbers such as u and v are added and u is small relative to v.] *What should be returned by the functions* `roundSumDown` *and* `roundSumUp` *for each possibility in this case? Be sure that the interval being calculated is not larger than necessary by examining the sign of x and y in your analysis.*

Exercise 3.3 *From the previous exercises, you should now be able to write the two supporting functions* `roundSumDown` *and* `roundSumUp`*. Correct the* `IntervalArithmetic` *module using these two functions and write a small program*

[13] Similar definitions of other arithmetic operations on intervals can be stated.

to test your new `IntervalArithmetic` *module. Comment about the portability of your implementation. How have Fortran 90's environmental inquiry and manipulation functions helped make your implementation portable?*

3.2 Subtleties in Solving a Quadratic Equation

Solving a quadratic equation at first glance seems to be a trivial calculation using the quadratic formula,

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

However, a closer look reveals a number of numerical subtleties that, if overlooked, could lead to grossly inaccurate approximations to the roots.

To see one of the problems associated with the numerical aspects of the quadratic formula, work through the following exercise.

Exercise 3.4 *Solve the quadratic equation,*

$$x^2 - 12.4x + 0.494 = 0$$

by evaluating the quadratic formula using three-digit decimal arithmetic and unbiased rounding[14]*. The exact roots rounded to 6 digits are* 0.0399675 *and* 12.3600.[15]

The above exercise illustrates an important numerical problem called *cancellation* or *loss of significance* which manifests itself when subtracting values of nearly equal magnitude.[16] Cancellation occurs when the digits necessary to accurately define the difference have been discarded by rounding in previous calculations due to the finite precision of machine arithmetic. Problems arise when this difference is an intermediate result which must be used to complete the calculation—most of the significant digits that remain after rounding are eliminated by subtraction. To complicate the situation, the digits that become significant after subtraction may be accurate to only a few places due to the previous rounding errors in the two values being subtracted. For example, suppose that a calculation contains the intermediate values, 0.37294328×10^1 and 0.37294300×10^1, both correct only to 6 significant figures, with the last two digits incorrect due to rounding errors in previous calculations. Assuming a computer with 8 digit decimal arithmetic, the computed difference in the two

[14]Three-digit decimal arithmetic using unbiased rounding truncates after the third digit if the fourth digit is 4 or less, rounds the third digit up if the last digit is 6 or more, and rounds the third digit to an even digit if the fourth digit is exactly 5.

[15]Using three-digit arithmetic and unbiased rounding, you should obtain roots 0.05 and 12.4. Note that the larger calculated root is correctly rounded while the smaller calculated root has no correct figures.

[16]In the previous exercise, this occurs in calculating the smaller root.

numbers is $0.37294328 \times 10^1 - 0.37294300 \times 10^1 = .28 \times 10^{-6}$. On the assumption that the last two digits are incorrect due to previous rounding errors, this difference contains no correct figures, all of which have been brought to significance after subtraction.

Such severe cancellation can usually be eliminated by algebraic reformulation. In the case of the quadratic equation, the cancellation observed in the previous exercise results from the subtraction performed between $-b$ and $\sqrt{b^2 - 4ac}$.[17] This cancellation occurs when $4ac$ is small relative to b^2, so that $\sqrt{b^2 - 4ac} \approx b$. This problem may be resolved by calculating the larger root (in absolute value) using the quadratic formula and obtaining the smaller root by another means. The larger root (in absolute value) can be obtained from the quadratic formula by choosing the sign of $\sqrt{b^2 - 4ac}$ so that no subtraction occurs. The smaller root (in absolute value) can be obtained by observing that the product of the roots of a quadratic equation must equal the constant term. So, for a general quadratic equation, $p(x) = ax^2 + bx + c$, the product of the roots, $r_1 r_2 = c/a$. Thus, the second root may be obtained by division, circumventing the cancellation problem in the previous exercise.

Exercise 3.5 *Write a program to solve a quadratic equation using the methods described above. You may find the Fortran 90 intrinsic function* `sign` *helpful in choosing the proper sign in the quadratic formula to calculate the larger root.*

As mentioned previously, the calculation of $\sqrt{b^2 - 4ac}$ is another possible source of severe cancellation. This is illustrated in the following exercise.

Exercise 3.6 *Solve the quadratic equation,*

$$x^2 - 12.4x + 38.4 = 0$$

by evaluating the quadratic formula using three-digit arithmetic and unbiased rounding. The exact roots rounded to 6 digits are 6.00000 *and* 6.40000*. What is the value of the expression* $b^2 - 4ac$ *in three-digit arithmetic? What does this indicate about the roots?*[18] *What is the source of the problem?*

As indicated in the previous exercise, cancellation may occur in computing $\sqrt{b^2 - 4ac}$ when $4ac \approx b^2$. This time, however, algebraic reformulation cannot be used to solve the problem. *Intermediate extended precision* is another technique used to combat the effects of severe cancellation. The quantity $\sqrt{b^2 - 4ac}$ is a prime candidate for the use of extended precision because the result of applying the square root function will bring the digits used in the double precision

[17] As will be shown shortly, $\sqrt{b^2 - 4ac}$ can be another source of cancellation. This is a potentially serious problem because the application of the square root function could bring the inaccurate low order digits to significance.

[18] Using three-digit arithmetic and unbiased rounding, you should obtain a double root of 6.2, correct to only one significant figure.

calculation to significance. Converting this quantity to single precision preserves the significant digits that would have been inaccurate if the entire computation had been done in one precision only.[19]

Exercise 3.7 *Modify your quadratic equation solver to use double precision in calculating* $\sqrt{b^2 - 4ac}$. *Fortran 90 has two type conversion functions,* `real` *and* `dble`, *to convert numbers to single precision and double precision, respectively.*

[19] A Fortran 90 double precision number is required to contain at least twice as many digits in the mantissa as a single precision number. This guarantees that b^2 is representable in double precision. The product $4ac$ is also immune to roundoff error on a machine with a binary base. Thus, the two values b^2 and $4ac$ may be computed exactly in double precision.

4 Complete Example Programs

The following programming examples illustrate many other features of Fortran 90 that are not present in the examples and exercises of the previous sections. Comments about the language are also included so that the programs can be used to learn much of Fortran 90 by example.

4.1 Rational Arithmetic

```
! File typical.1.f90

! The suffix .f90 permits the use of the new free source form available in
! Fortran 90.

! A module is a new feature in Fortran 90.  Modules are primarily used to
! declare data, subroutines, and functions that are accessible in more than
! one program unit.  Modules are a safe alternative to common in Fortran 77,
! and provide much more functionality by including more than just data.
! The data, subroutine, and function names in a module are made available
! by using the module in the program unit using the USE statement together
! with the module name.

! Case is not significant in Fortran 90.

module TypicalModule
   integer, parameter :: N = 4     ! Declare an integer constant, N.
   integer, parameter :: M = 50    ! Declare an integer constant, M.

   private M   ! Make the integer constant visible only within this module.

   ! Define a derived type called rational that contains two integer
   ! components, n and d.  Derived types are similar to records in Pascal
   ! or structures in C.  Individual components are accessed via the ``%''
   ! operator (see the rationalAdd function below).

   type rational
      integer n, d
   end type rational


   ! Every subroutine or function has an interface which indicates the name,
   ! the arguments, their types, attributes, and the type and attributes of
   ! the function (for a function).

   ! There are two types of interfaces in Fortran 90, implicit interfaces
   ! and explicit interfaces.  Implicit interfaces are used in Fortran 77
   ! external procedures and assume that a procedure call is correct in the
   ! type and number of arguments passed.  Explicit interfaces, however, have
   ! the advantage that type-checking of actual and dummy arguments can be
   ! performed for procedure calls.  Incorrect calls or invocations are
   ! detected by the compiler.

   ! Interfaces of subroutines and functions within a module are always
```

```
! explicit.  However, an interface block is needed to define a generic
! procedure name or operator and the set of procedures to which the name
! applies.  The following interface block extends the binary ``+''
! operator.  Now, the function rationalAdd can be invoked using the
! binary operator ``+'' on two objects of type( rational ).

interface operator (+)
   module procedure rationalAdd
end interface

private gcd   ! Use the gcd function internal to this module only.


! The contains statement indicates the presence of one or more internal
! subprograms that are included in the module and is necessary to
! separate the specification statements of the module from the
! subprogram definitions.

contains

function rationalAdd( left, right )

   ! To declare a variable to be of some derived type, use the type
   ! statement with the derived type's name in parenthesis.

   type( rational ) rationalAdd

   ! As in Fortran 77, parameter passing is by reference.
   ! It may be necessary for the compiler to generate temporaries in
   ! some cases.  However, changes to these temporaries affect the actual
   ! argument.  Fortran 90 provides the intent attribute to further
   ! specify and document a variable's use in the program.  Possible
   ! intent specifications are in, out, and inout.  Variables violating
   ! their intent are caught by the compiler.

   type( rational ), intent(in) :: left, right

   integer k, m1, m2
   type( rational ) sum

   k = gcd( left%d, right%d )
   m1 = left%d / k
   m2 = right%d / k

   ! To assign a value to a variable of a derived type such as sum, use
   ! a structure constructor as indicated below.  A structure constructor
   ! consists of the derived type name together with the value to be
   ! assigned to each component of the derived type, in the order
   ! declared in the specification of the derived type.  (A derived type
   ! may also be assigned by assigning values to its individual
   ! components.)

   sum = rational( left%n * m2 + right%n * m1, left%d * m2 )

   k = gcd( sum%n, sum%d )
```

```
   rationalAdd = rational( sum%n / k, sum%d / k )  ! Reduce the rational.
end function rationalAdd


! Recursion is now permissible in Fortran 90.  The function gcd is
! declared recursively below.

! If a function calls itself, either directly or indirectly, the
! keyword recursive must appear in the function statement.  Recursive
! functions must also declare a result clause to be used rather than the
! function name.  This requirement is to avoid ambiguity with
! array-valued functions that are directly recursive.  The result
! variable is used to hold the function result for each function
! invocation; the function name is used to invoke the function itself.

recursive function gcd( a, b ) result( divisor )

   ! Note that the function itself is not declared when the result
   ! variable is present.  The type of the function is the type of
   ! the result variable.  Thus, only the result variable may be
   ! declared.

   integer divisor
   integer, intent(in) :: a, b

   integer m, n

   ! Multiple statements may be written on a single source line
   ! provided they are delimited with semicolons.

   m = abs(a); n = abs(b)
   if ( m > n ) call swap( m, n )  ! Insure that m <= n.

   ! When the function invokes itself recursively, the result variable
   ! should be used to store the result of the function.  The function
   ! name is used to invoke the function.  Thus, the function name should
   ! not appear on the left-hand side of an assignment statement.

   if ( m == 0 ) then
      divisor = n
   else
      divisor = gcd( mod( n, m ), m )
   end if

   ! Unlike internal subprograms, module procedures, such as gcd, may
   ! have internal subprograms defined within them (to one level only).
   ! As with module procedures, internal subprograms also have an
   ! explicit interface.  Thus, the swap subroutine is not declared in
   ! the gcd function above---its interface is explicit.


   contains

   ! The swap subroutine is internal to gcd and is not visible
   ! elsewhere, not even in this module.
```

```
      subroutine swap( x, y )
         integer, intent(inout) :: x, y

         integer tmp

         tmp = x; x = y; y = tmp
      end subroutine swap

   end function gcd

end module TypicalModule




! The above module can be in a different source file than the program and
! be compiled separately.  The explicit interfaces of the module procedures
! guarantee that the compiler will check the actual and dummy arguments, in
! both type and number, for each module procedure called in the program.

program Typical

   ! A module is accessed via the use statement.  With the use statement
   ! below, all public names (names not declared private in the module) are
   ! available for use in the program.  Use statements must immediately
   ! follow the program statement.  A program may use an unlimited
   ! number of modules.

   use TypicalModule

   ! Note that a variable may be initialized at the time of declaration.

   type( rational ) :: a = rational( 1, 2 ), b = rational( 3, 4 ), c


   ! The derived type rational and its associated function, ``+'', may
   ! now be used between a and b.

   c = a + b   ! This statement is equivalent to c = rationalAdd( a, b ).


   ! Input two rational numbers and output their sum.

   print *, 'Enter two rational numbers.'
   print *

   ! Nonadvancing output is possible using the advance specifier in the
   ! write statement.  This permits multiple write statements to output
   ! a continuing line of output.

   write( *, fmt = "(a)", advance = "no" ) 'Numerator of a   :  '
   read  *, a%n
```

```
   write( *, fmt = "(a)", advance = "no" ) 'Denominator of a :  '
   read  *, a%d

   write( *, fmt = "(a)", advance = "no" ) 'Numerator of b   :  '
   read  *, b%n

   write( *, fmt = "(a)", advance = "no" ) 'Denominator of b :  '
   read  *, b%d

   c = a + b

   print *
   print *, a%n, '/', a%d, '+', b%n, '/', b%d, '=', c%n, '/', c%d

end program Typical
```

4.2 Linear Equation Solvers

```
! File typical.2.f90

module LinearSolvers
   implicit none

   ! The default value for the smallest pivot that will be accepted
   ! using the LinearSolvers subroutines.  Pivots smaller than this
   ! threshold will cause premature termination of the linear equation
   ! solver and return false as the return value of the function.

   real, parameter :: DEFAULT_SMALLEST_PIVOT = 1.0e-6

contains

   ! Use Gaussian elimination to calculate the solution to the linear
   ! system, A x = b.  No partial pivoting is done.  If the threshold
   ! argument is present, it is used as the smallest allowable pivot
   ! encountered in the computation; otherwise, DEFAULT_SMALLEST_PIVOT,
   ! defined in this module, is used as the default threshold.  The status
   ! of the computation is a logical returned by the function indicating
   ! the existence of a unique solution (.true.), or the nonexistence of
   ! a unique solution or threshold passed (.false.).

   ! Note that this is an inappropriate method for some linear systems.
   ! In particular, the linear system, M x = b, where M = 10e-12 I, will
   ! cause this routine to fail due to the presence of small pivots.
   ! However, this system is perfectly conditioned, with solution x = b.

   function gaussianElimination( A, b, x, threshold )
      implicit none
      logical gaussianElimination
      real, dimension( :, : ), intent( in ) :: A   ! Assume the shape of A.
      real, dimension( : ), intent( in ) ::  b     ! Assume the shape of b.
      real, dimension( : ), intent( out ) :: x     ! Assume the shape of x.

      ! The optional attribute specifies that the indicated argument
```

```
! is not required to be present in a call to the function.  The
! presence of optional arguments, such as threshold, may be checked
! using the intrinsic logical function, present (see below).

real, optional, intent( in ) :: threshold

integer i, j   ! Local index variables.
integer N      ! Order of the linear system.
real m         ! Multiplier.
real :: smallestPivot = DEFAULT_SMALLEST_PIVOT

! Pointers to the appropriate rows of the matrix during the elmination.
real, dimension( : ), pointer :: pivotRow
real, dimension( : ), pointer :: currentRow

! Copies of the input arguments.  These copies are modified during
! the computation.
! The target attribute is used to indicate that the specified
! variable may be the target of a pointer.  Rows of ACopy are targets
! of pivotRow and currentRow, defined above.

real, dimension( size( A, 1 ), size( A, 2) ), target :: ACopy
real, dimension( size( b ) ) :: bCopy

! Status of the computation.  The return value of the function.
logical successful

! Change the smallestPivot if the threshold argument was included.
if ( present( threshold ) ) smallestPivot = abs( threshold )

! Setup the order of the system by using the intrinsic function size.
! size returns the number of elements in the specified dimension of
! an array or the total number of elements if the dimension is not
! specified.  Also assume that a unique solution exists initially.

N = size( b )
ACopy = A
bCopy = b
successful = .true.

! Begin the Gaussian elimination algorithm.
! Note the use of array sections in the following loops.  These
! eliminate the need for many do loops that are common in Fortran
! 77 code.
! Pointers are also used below and enhance the readability of the
! elimination process.

! Begin with the first row.
i = 1

! Reduce the system to upper triangular.
do while ( ( successful ) .and. ( i <= N-1 ) )

   ! The following statement is called pointer assignment and uses
   ! the pointer assignment operator '=>'.  This causes pivotRow
```

```
      ! to be an alias for the ith row of ACopy.  Note that this does
      ! not cause any movement of data.

      ! Assign the pivot row.
      pivotRow => ACopy( i, : )

      ! Verify that the current pivot is not smaller than smallestPivot.
      successful = abs( pivotRow( i ) ) >= smallestPivot

      if ( successful ) then

! Eliminate the entries in the pivot column below the pivot row.

do j = i+1, N
   ! Assign the current row.
   currentRow => ACopy( j, : )

           ! Calculate the multiplier.
           m = currentRow( i ) / pivotRow( i )

           ! Perform the elimination step on currentRow and right
           ! hand side, bCopy.
           currentRow = m * pivotRow - currentRow
           bCopy( j ) = m * bCopy( i ) - bCopy( j )
end do

      end if

      ! Move to the next row.
      i = i + 1

  end do

  ! Check the last pivot.
  pivotRow => ACopy( N, : )
  if ( successful ) successful = abs( pivotRow( N ) ) >= smallestPivot

  if ( successful ) then
     do i = N, 2, -1   ! Backward substitution.

        ! Determine the ith unknown, x( i ).
        x( i ) = bCopy( i ) / ACopy( i, i )

        ! Substitute the now known value of x( i ), reducing the order of
        ! the system by 1.
        bCopy = bCopy - x( i ) * ACopy( :, i )

     end do
  end if

  ! Determine the value of x( 1 ) as a special case.
  if ( successful ) x( 1 ) = bCopy( 1 ) / ACopy( 1, 1 )

  ! Prepare the return value of the function.
  gaussianElimination = successful
```

```
end function gaussianElimination


! The LU decomposition of a matrix may be represented in a compact form
! existing in a single matrix, M,  if the assignments M=L and M=U are
! done (in that order).  The diagonal entries in L are assumed to be
! unity so that no storage space is necessary.  Instead, the diagonal
! of M is used to hold the diagonal entries of U.  This is a common
! method of storing the LU decomposition of a matrix.

! The algorithm belows makes an additional assumption concerning the
! pivots or diagonal elements of U.  Computation terminates if one of
! these pivots is smaller than the given or default threshold.  In this
! case, the LU decomposition is not formed.  Note that this algorithm
! successfully terminates if such an LU can be computed.  In this case
! the coefficient matrix, A, is nonsingular.  (No attempt for recovery,
! such as permutation of rows, is done.)

! Compute the LU decomposition of A, storing the result in LU so that
! A is not overwritten.  If the threshold argument is present, it is used
! as the smallest allowable pivot encountered in the computation;
! otherwise, DEFAULT_SMALLEST_PIVOT, defined in this module, is used as
! the default threshold during the computation.  The status of the
! computation is a logical returned by the function indicating the
! success (.true.) or failure (.false.) of the factorization
! After the computation, LU will contain the multipliers below the main
! diagonal (L) and the result after elimination on and above the main
! diagonal (U), so that A = L * U.

function LUFactor ( A, LU, threshold )
   implicit none
   logical LUFactor
   real, dimension( :, : ), intent( in ) :: A
   real, dimension( :, : ), intent( out ) :: LU
   real, optional, intent( in ) :: threshold

   integer k, i
   integer N
   logical successful   ! Status of the computation.
   real :: smallestPivot = DEFAULT_SMALLEST_PIVOT

   ! Reassign the smallestPivot, set the order of the system, and
   ! copy A into LU as it will be written to during the factorization.

   if ( present( threshold ) ) smallestPivot = abs( threshold )
   N = size( A, 1 )
   LU = A

   ! Begin the LU factorization algorithm.
   ! The status of the computation is initially successful.
   successful = .true.

   k = 1   ! Begin with the first column.
   do while ( ( successful ) .and. ( k <= N-1 ) )
```

```
      ! Verify that the kth pivot is not smaller than smallestPivot.
      successful = abs( LU( k, k ) ) >= smallestPivot

      if ( successful ) then
         ! Calculate the multipliers (L) for the current column.
         LU( k+1:N, k ) = LU( k+1:N, k ) / LU( k, k )

         ! Perform elimination on the upper portion of the matrix (U).
         do i = k+1, N
            LU( i, k+1:N ) = LU( i, k+1:N ) - LU( i, k ) * LU( k, k+1:N )
         enddo

         k = k + 1   ! Move to the next column.
      end if

   enddo

   ! Prepare the return value of the function.
   LUFactor = successful

end function LUFactor


! Let A = L*U where LU represents the LU decomposition of A stored in the
! format produced by LUFactor, A, L, U in R**(NxN).
! Solve the linear system, A x = b, using the LU decomposition of A stored
! in LU.  Since LU is the LU decomposition of A, A is nonsingular.
! Consequently, the columns of A constitute a basis for R**N.   So, there
! must exist a unique solution to the linear system A x = b.
! LUSolve returns the solution to this linear system.

function LUSolve( LU, b ) result( x )
   implicit none
   real, dimension( :, : ), intent( in ) :: LU
   real, dimension( : ), intent( in ) :: b
   real, dimension( size( b ) ) :: x

   integer k
   integer N
   real, dimension( size( b ) ) :: bCopy

   ! Determine the order of the system and store a copy of b in bCopy
   ! as it is written during the computation.
   N = size( b )
   bCopy = b

   ! Assume LU is in the form of LU and solve the system in two steps.
   ! First, using forward elmination to solve L y = b, then using
   ! backward elmination to solve U x = y.  In both cases, the right
   ! hand side is overwritten with the solution as it is computed.

   ! Forward elimination.  Store the solution into the right hand side.
   do k = 1, N-1
      bCopy( k+1:N ) = bCopy( k+1:N ) - bCopy( k ) * LU( k+1:N, k )
```

```
   end do

   ! Backward elimination.  Store the solution into the right hand side.
   do k = N, 2, -1
      bCopy( k ) = bcopy( k ) / LU( k, k )
      bCopy( 1:k-1 ) = bCopy( 1:k-1 ) - bCopy( k ) * LU( 1:k-1, k )
   end do

   ! Solve for the 1st unknown as a special case.
   bCopy( 1 ) = bCopy( 1 ) / LU( 1, 1 )

   ! Assign a return value for the function via its result variable, x.
   x = bCopy

end function LUSolve


! Output A in Matlab format, using name in the Matlab assignment statement.
subroutine printMatrix( A, name )
   implicit none
   real, dimension( :, : ) :: A   ! Assume the shape of A.
   character name  ! Name for use in assignment, ie, name = ......

   integer n, m, i, j

   n = size( A, 1 )
   m = size( A, 2 )

   write( *, fmt="(a1,a5)", advance = "no" ) name, ' = [ '

   ! Output the matrix, except for the last row, which needs no `;'.
   do i = 1, n-1

      ! Output current row.
      do j = 1, m-1
         write( *, fmt="(f10.6,a2)", advance = "no" ) A( i, j ), ', '
      end do

      ! Output last element in row and end current row.
      write( *, fmt="(f10.6,a1)" ) A( i, m ), ';'

   end do

   ! Output the last row.
   do j = 1, m-1
      write( *, fmt="(f10.6,a2)", advance = "no" ) A( i, j ), ', '
   end do

   ! Output last element in row and end.
   write( *, fmt="(f10.6,a1)" ) A( i, m ), ']'

end subroutine printMatrix


! Output b in Matlab format, using name in the Matlab assignment statement.
```

```
   subroutine printVector( b, name )
      implicit none
      real, dimension( : ) :: b   ! Assume the shape of b.
      character name   ! Name for use in assignment, ie, name = ......

      integer n, i

      n = size( b )

      write( *, fmt="(a1,a5)", advance = "no" ) name, ' = [ '

      do i = 1, n-1
         write( *, fmt = "(f10.6,a2)", advance = "no" ) b( i ), ', '
      end do

      write( *, fmt = "(f10.6,a2)" ) b( n ), ']'''

   end subroutine printVector

end module LinearSolvers



! A program to solve linear systems using the LinearSolvers module.

program SolveLinearSystem

   ! Include the module for the various linear solvers.

   use LinearSolvers
   implicit none

   integer, parameter :: N =  5 ! Order of the linear system.
   real, parameter    :: TOO_SMALL = 1.0e-7  ! Threshold for pivots.

   ! Declare the necessary arrays and vectors to solve the linear system
   ! A x = b.

   real, dimension( N, N ) :: A   ! Coefficient matrix.
   real, dimension( N ) :: x, b   ! Vector of unknowns, and right hand side.
   real, dimension( N, N ) :: LU  ! Matrix for LU factorization of A.

   logical successful   ! Status of computations.


   ! The intrinsic subroutine, random_number, fills a real array or scalar,
   ! with uniformly distributed random variates in the interval [0,1).

   call random_number( A )   ! Initialize the coefficient matrix.
   call random_number( b )   ! Initialize the right-hand side.

   ! Output the matrix in Matlab format for ease of checking the solution.
   call printMatrix( A, 'A' )
   call printVector( b, 'b')
```

```
   ! Use Gaussian elmination to calcuate the solution of the linear system.
   ! The call below uses the default threshold specified in the
   ! LinearSolvers module by omitting the optional argument.

   successful = gaussianElimination( A, b, x )

   print *, '===================================='
   print *, 'Gaussian Elimination:'
   print *, '------------------------------------'
   if ( successful ) then
      call printVector( x, 'x' )
      print *, 'Infinity Norm of Difference = ',  &
 maxval( abs ( matmul( A, x ) - b ) )
   else
      print *, 'No unique solution or threshold passed.'
   end if

   ! Compute the LU decomposition of A.

   successful = LUFactor( A, LU )

   ! Calculate the solution of the linear system given the LU decomposition.
   ! Output the results.

   print *
   print *, '===================================='
   print *, 'LU Factorization:'
   print *, '------------------------------------'
   if ( successful ) then
      x = LUSolve(LU, b)

      print *, 'LU Decomposition:'
      call printMatrix( LU, 'M' )
      call printVector( x, 'x' )
      print *, 'Infinity Norm of Difference = ', &
 maxval( abs ( matmul( A, x ) - b ) )
   else
      print *, 'No unique solution or threshold passed.'
   end if

end program SolveLinearSystem
```

4.3 One-Dimensional Multigrid

```
! Full Multigrid V-cycle for a one dimensional pde, u''(x) = g(x),
!  u(0) = 1, u(1) = e on [0,1].

program FMV

   ! u contains the approximate solution to the current problem on all grids.
   ! f contains the right hand side for the current problem on all grids.
   ! gridLevelInfo contains the starting index (into u and
   !   f) of the current grid as well as the dimension of the current
```

```
!   grid.
! fullSize is the total static storage necessary to maintain
!   results for all grids during execution.
! nSweepsBefore is the number of relaxation sweeps to perform
!   before injection of the residual to the next coarser level.
! nSweepsAfter is the number of relaxation sweeps to perform
!   after correction to the solution on the current grid.
! g is the right hand side of the de.

implicit none

integer, parameter :: K = 15  ! The number of grid levels

integer, parameter :: fullSize =  2**(K+1) + K - 2

real u(fullSize), f(fullSize)
integer gridLevelInfo(K,2)

integer nSweepsBefore, nSweepsAfter
parameter (nSweepsBefore = 1)
parameter (nSweepsAfter  = 1)

integer i, n      ! Index variables
integer index     ! Current displacement into u and f
integer fineIndex, coarseIndex   ! Displacements into u and f
integer nFine, nCoarse ! Size of fine and coarse grids
integer level     ! Current grid level
real h            ! Current step size

interface
   function g( x )
      real g
      real, intent( in ) :: x
   end function g
end interface



! Initialization

h = 0.5   ! Begin on coarsest grid
index = 1

! Set up right hand side and displacements into the solution
!  and right hand side arrays, u and f.

do i = 1, K
   N = 2**i + 1  ! Size of current grid
   gridLevelInfo(i,1) = index
   gridLevelInfo(i,2) = N

   ! Initialize right hand side for all grids for the FMV
   !  cycle.  Note that this initialization is used to solve
   !  the problem Au = f on coarser and coarser grids.  It is
```

```
   !  not for residual correction.

   call setF( f( index:index+N-1 ), h, g )   ! Note f is a vector in R**N
   index = index + N
   h = h/2
enddo



! Set up the boundary conditions for the coarsest grid.
! These are copied by interpolate() into finer grids.

index = gridLevelInfo(1,1)
N     = gridLevelInfo(1,2)

call setCoarseBC( u( index:index+N-1 ) )


! Begin FMV-cycle

! Relax on coarsest grid.
! Note that this code is duplicated within the general loop.  Here,
!  it sets up the coarsest grid for the following loop.
! Notice that relax with one unknown is a direct solve on the
!  coarsest grid.

call relax( u( index:index+N-1 ), f( index:index+N-1 ), 1 )


do level = 2, K

   ! Retrieve intial guess from previous coarser grid using
   !  interpolation.  Note that this information has been
   !  calculated either by the direct solve above or the
   !  ascent phase of the V-cycle in the general case.

   fineIndex = gridLevelInfo(level,1)
   nFine     = gridLevelInfo(level,2)

   coarseIndex = gridLevelInfo(level-1,1)
   nCoarse     = gridLevelInfo(level-1,2)


   call interpolate( u( fineIndex:fineIndex+nFine-1 ), &
                     u( coarseIndex:coarseIndex+nCoarse-1 ) )

   ! Begin the descent phase of a single V-cycle.
   ! Descend to the coarsest grid for the current problem.

   do i = level, 2, -1
      fineIndex = gridLevelInfo(i,1)
      coarseIndex = gridLevelInfo(i-1,1)

      nFine   = gridLevelInfo(i,2)   ! Dimension of fine grid
      nCoarse = gridLevelInfo(i-1,2) ! Dimension of coarse grid
```

```
   ! Relax on current grid using the initial guess just
   !  interpolated from the previous coarser grid.

   call relax( u( fineIndex:fineIndex+nFine-1 ),  &
               f( fineIndex:fineIndex+nFine-1 ),  &
               nSweepsBefore)

   ! Inject residual of current equation into next coarser
   !  level.  Note that this overwrites the previous right
   !  hand side.  After the call to the subroutine
   !  injectResidual(), the right hand side contains
   !  the residual, r.  The solution to Ae = r, on this coarser
   !  grid yields the correction for the current problem at
   !  level i.  The correction is added in the ascent phase of
   !  the V-cycle.

   call injectResidual( u( fineIndex:fineIndex+nFine-1 ), &
                        f( fineIndex:fineIndex+nFine-1 ), &
                        f( coarseIndex:coarseIndex+nCoarse-1 ) )

   ! Force a zero initial guess for the residual equation,
   !  Ae = r for coarse level.

   call fillZeros( u( coarseIndex:coarseIndex+nCoarse-1 ) )



enddo  ! End downward phase of V-cycle


! Now at coarsest grid, N = 3.
! Solve the current problem exactly.
! Residual equations are solved yielding corrections to the
!  current equation on the next finer grid.
! Notice that relax with one unknown is a direct solve on the
!  coarsest grid.

index = gridLevelInfo(1,1)
N     = gridLevelInfo(1,2)

call relax( u( index:index+N-1 ), f( index:index+N-1 ), 1 )


! Begin ascent phase of V-cycle.
! Ascend to the finest grid (level) for the current
!  problem.

do i = 2, level
   coarseIndex = gridLevelInfo(i-1,1)
   fineIndex = gridLevelInfo(i,1)

   nCoarse = gridLevelInfo(i-1,2)
   nFine   = gridLevelInfo(i,2)
```

```
      ! Get correction for current problem from previous
      !  level using interpolation.
      ! Add to the current solution of the current problem.

      call correct( u( fineIndex:fineIndex+nFine-1 ), &
                    u( coarseIndex:coarseIndex+nCoarse-1 ) )

      ! Relax on the fine grid problem after the correction.
      call relax( u( fineIndex:fineIndex+nFine-1 ), &
                  f( fineIndex:fineIndex+nFine-1 ), &
                  nSweepsAfter)


   enddo  ! End upward phase of V-cycle

enddo  ! End nested iteration

! Out a summary of results.

index = gridLevelInfo( K,1 )
N = gridLevelInfo( K,2 )

call writeResults( u( index:index+N-1 ), K )


contains


! Set up the boundary conditions for the coarsest grid.
! These are copied by interpolate() into finer grids.

subroutine setCoarseBC( u )
implicit none

real u(0:)

integer N

N = size( u )

u( 0 ) = 1.0
u( N-1 ) = exp( 1.0 )

end subroutine setCoarseBC




! Set the right hand size, f, using the step size h and the right
!  hand side of the pde, g(x).

subroutine setF( f, h, g )
implicit none

real f(0:)
```

```
integer N

real h

interface
   function g( x )
      real g
      real, intent( in ) :: x
   end function g
end interface

integer i
real hSquared


hSquared = h * h
N = size( f )

do i = 1, N-2
   f(i) = hSquared * exp(i * h)   ! Note, x = i * h
end do

end subroutine setF




! Set the solution vector, u, to zero so that the residual equation
!  is solved with initial guess 0.

subroutine fillZeros( u )
implicit none

real u(0:)

u = 0.0

end subroutine fillZeros




! Relaxation method.
! The relaxation method currently used is Jacobi.
! Note that u(0) and u(N) are both 0.0.

subroutine relax( u, f, nSweeps )
implicit none

integer nSweeps
real u(0:), f(0:)

integer i, j
integer N
```

```
N = size( u )

do j = 1, nSweeps
   do i = 1, N-2
      u(i) = 0.5 * ( u(i-1) + u(i+1) - f(i) )
   end do
end do

end subroutine relax




subroutine interpolate( uFine, uCoarse )
implicit none

! This subroutine interpolates uCoarse to uFine.
! At even-numbered fine grid points, the values are transferred
!  directly from the coarse grid.
! At odd-numbered fine grid points, value is the average of the
!  two adjacent coarse grid points.

! uFine(0) and uFine(nFine-1) are boundary points.
! Note that 0 and nFine-1 are even numbered fine grid points.

real uFine(0:), uCoarse(0:)

integer i
integer nFine, nCoarse

nFine = size( uFine ); nCoarse = size( uCoarse )

do i = 0, nCoarse-1
   uFine(2*i) = uCoarse(i)
end do

do i = 1, nFine-2, 2
   uFine(i) = 0.5 * ( uFine(i-1) + uFine(i+1) )
end do

end subroutine interpolate




subroutine correct( uFine, uCoarse )
implicit none

! Correct the solution uFine by adding the interpolated correction
!  given by uCoarse.
! Note that correct() uses the same interpolation algorithm as
!  interpolate.  Code was duplicated to avoid extra storage and the
!  overhead of another subroutine call.
! Note that this subroutine and interpolate() could be combined if
!  uFine were initialized to zero before the call to interpolate
```

```
!  in the descent phase of each V-cycle.

real uFine(0:), uCoarse(0:)

integer i
integer nFine, nCoarse

nFine = size( uFine ); nCoarse = size( uCoarse )

do i = 0, nCoarse-2
   uFine(2*i) = uFine(2*i) + uCoarse(i)
   uFine(2*i+1) = uFine(2*i+1) + 0.5 * &
                  ( uCoarse(i) + uCoarse(i+1) )
end do

uFine(2*(nCoarse-1)) = uFine(2*(nCoarse-1)) + uCoarse(nCoarse-1)

end subroutine correct




subroutine injectResidual( uFine, fFine, fCoarse )
implicit none

! Calculate the residual on the fine grid and inject it down to the
!  coarse grid using a full weighting restriction operator.

real uFine(0:), fFine(0:)

real fCoarse(0:)

integer i         ! Index variable
integer nFine, nCoarse

nFine = size( uFine ); nCoarse = size( fCoarse )


! Calculate the (2*i+1)st, (2*i+2)nd, and (2*i+3)rd components of
!  the residual on the fine grid.
! Inject this component of the residual into the right hand side
!  for the residual equation on the coarse grid.

do i = 1, nCoarse-2
   fCoarse(i) = 0.25 * ( fFine(2*i-1) - &
    ( uFine(2*i-2) - 2 * uFine(2*i-1) + uFine(2*i) ) + 2 * &
    ( fFine(2*i) - ( uFine(2*i-1) - 2 * uFine(2*i) + &
      uFine(2*i+1) ) ) + fFine(2*i+1) - &
    ( uFine(2*i) - 2 * uFine(2*i+1) + uFine(2*i+2) ) )
end do

end subroutine injectResidual
```

```
   subroutine writeResults( u, K )
   implicit none

   real u(0:)
   integer K

   integer N

   N = size( u )

   print *, '----------------------------------------'
   print *, 'Summary of Results'
   print *, '------------------'
   print *, 'K =', K
   print *, 'N =', N
   print *, 'h =', 1.0/(N-1)
   print *, 'u(0)   =', u(0)
   print *, 'u(0.5) =', u( (N-1)/2 )
   print *, 'u(1)   =', u(N-1)
   print *, '----------------------------------------'

   end subroutine writeResults

end  ! End program FMV


   ! g(x) is the right hand side of the pde being solved where x is
   !   the independent variable.

   function g(x)
   real g
   real, intent( in ) :: x

   g = exp(x)

   end function g
```

References

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. Intertext Publications and McGraw-Hill, New York, 1992.

[2] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. PWS-Kent, Boston, fourth edition, 1989.

[3] S. D. Conte. *Elementary Numerical Analysis*. McGraw-Hill Series in Information Processing and Computers. McGraw-Hill, New York, 1965.

[4] T. M. R. Ellis. *Fortran 77 Programming*. Addison-Wesley, Wokingham, England, second edition, 1990.

[5] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Numerical Linear Algebra and Optimization*, volume 1. Addison-Wesley, Redwood City, California, 1991.

[6] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, Massachusetts, 1989.

[7] David Kincaid and Ward Cheney. *Numerical Analysis.* Brooks/Cole, Belmont, California, 1991.

[8] J. Stoer and Bulirsch. *Introduction to Numerical Analysis.* Springer-Verlag, New York, 1980.