Re-Engineering the Finite Element Library: The Transformation of a Legacy Fortran Library

Dr C Greenough

October 2003

Abstract

The Finite Element Library (`FELIB`) [1, 2, 3] was first designed and implemented in the early 1980s and since then there have been four releases of the software - Release 4 is the current release.

Many individuals and groups have made use of `FELIB` in developing finite element based applications and in the teaching of finite element techniques. Some 3000 known copies of the library are known to exist (through monitoring *httpd* and *ftp* accessing) and there are probably many other copies obtained through third parties.

This report details the re-design and re-engineering of the original Fortran 77 `FELIB` to make use of the new features of Fortran 90/95. This process provides a very useful way in assessing some of the software tools which can assist in this transformation and re-design. The report contains short summaries on tool such as `TOOLPACK` and `plusFORT` used in this work.

The basic design goals are discussed in light of Fortran 90/95 features and methods of implementation detailed. There is a short debate on whether to use `POINTER`s to arrays or `ALLOCATABLE` arrays and the overall `MODULE` structure of this implementation is described.

Full details of the Fortran 77 and Fortran 90/95 versions of `FELIB` are to found on the Mathematical Software Group Web site under `FELIB`.

Keywords: finite element, library design, Fortran 95, legacy software

---

Email: `c.greenough@rl.ac.uk`
Reports can be obtained from: `http://epubs.stfc.ac.uk`

Software Engineering Group
Scientific Computing Department
STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
Oxfordshire OX11 0QX

1 Introduction

The Finite Element Library (`FELIB`) [1, 2, 3] was first designed and implemented in the early 1980s and since then there have been four releases of the software - Release 4 is the current release.

Many individuals and groups have made use of `FELIB` in developing finite element based applications and in the teaching of finite element techniques. Some 3000 known copies of the library are known to exist (through monitoring *httpd* and *ftp* accessing) and there are probably many other copies obtained through third parties.

The original library, based on an existing prototype library of Prof IM Smith of Manchester University [5], was implemented in Fortran 66, with subsequent versions making use of Fortran 77, as part of one of the Rutherford Appleton Laboratory's Engineering Support programs funded by the then SERC (now EPSRC). The Numerical Algorithms Group Ltd (Oxford) provided the primary outlet to the scientific community. The prototype library was highly re-engineered and its two-level structure fully documented.

Although other language versions of the base Fortran 66 library were partially implemented - C, DAP-Fortran and Ada - it is not until recently that is was thought useful to consider developing a Fortran 90/95 version. In 1998 Prof Smith and Dr Griffiths [6] published a Fortran 90/95 version their library which has grown significantly through the years and there is also a version for SMPD machines [7].

Within the CLRC programme FELIB has developed and PVM/MPI versions have been developed [4] as too have a family of pre- and post processing routines tailored to FELIB. Even though originally designed in the 80s `FELIB` is still well used judged by the level of Email inquiries and comments.

2 Design goals - old and new

The primary goal of `FELIB` was to provide a tool box of routines components providing the majority of the steps required in a finite element base analysis together with a selection of *example* programs to illustrate there use. The two-level structure of the library satisfied the requirement.

The first Finite Element Library was targeted at Fortran 66. Even though Fortran 77 had been defined there were insufficient compilers to make the language attractive to developers. In 1985 it was thought that *Fortran 77 was not yet universal* [3]. Since then `FELIB` has taken on Fortran 77 and as mentioned above implementations have been started in a number of other languages. The same could be said of Fortran 90/95 now - has it become *universal*? Again, *no* is probably the answer! However with Fortran 2000 nearing the completion of its definition and with the availability of many more Fortran 90/95 compilers (`g95` will may well appear soon and the Intel compiler is available on Linux systems) it would appear that it has nearly achieved *universal* status.

Hence a Fortran 90/95 version of `FELIB` would not appear ill-timed. What should we be looking for in this new version? Clearly an exploitation of the new features of Fortran 90/95: arrays, modules, dynamic memory allocation etc. The goals of the first library have not changed in providing a prototyping tool. However, as Fortran 90/95 has Fortran 77 as a complete subset (apart from a list of deprecated features) it would seem reasonable that there should be an easy migration path from the old form of the library to the new.

To this end `FELIB90` contains a variety of routines that are not strictly necessary in Fortran 90/95. For example, the routine for initialising an array to zero, `MATNUL` and the matrix multiplication routine `MATMUL`. As explained below these can be replaced by the use of some of the basic features of Fortran 90/95 or by the use of standard intrinsic functions.

Fortran 90/95 provides mechanisms for defining application specific data types and structures. The simple array definitions of arrays such as the mesh geometry, `COORD`, could be replace by more object oriented structures. However it was decide to retain the simple array structures for the main data elements of the library.

Also FELIB used a large number of intermediate arrays such DTPD and ELK. These aided the readability of the FELIB programs and allowed them to mirror the mathematical analysis. Fortran 90/95 provides a number of mechanisms for allocating array space dynamically. In this re-design we will use these features to reduce the number of array definitions, space allocations and initialisations the programmer is require to perform.

For example the routine QQUA4 returns the abscissas and weights of a four-point quadrature rule. Its FELIB call is

```
CALL QQUA4(WGHT,IWGHT,ABSS,IABSS,JABSS,NQP,ITEST)
```

The arrays WGHT and ABSS are intermediate arrays whose sizes are determined by the content of the subroutine. They are only required for the assembly phase and the memory they occupy could be released once this phase is completed. By making these arrays dynamic and allowing the routine to allocate appropriate space the programmer task can be simplified. The call to the FELIB90 version of the routine could be reduced to

```
CALL QQUA4(wght,abss,nqp)
```

where the arrays wght and abss would be returned with the correct size, populated with data together with nqp. The user would still be required to provide a declaration of the arguments types but not necessarily their sizes. The routine would manage the allocation of memory space and initialisation.

As with many programs the initial lines of code are given over to declarations of variables and initialisations. In FELIB these could take up 10 to 30 lines. To aid this declaration process each FELIB90 example program will be provided with a definitions file which will specify the types of the most common variable and intermediate arrays. An example of one of these is given in Section 6.

3 Use of Fortran 90/95 features

Fortran 90/95 has made many additions to improve the Fortran language and some that open up new approaches to library design and implementation. In this section we will highlight some of these changes and consider how they might improve the design of a Fortran 90/95 version of the Finite Element Library.

3.1 Array features

The array features are one clear example of an important Fortran 90/95 feature. It is now possible to initialise and perform calculations on complete arrays whereas before, in Fortran 77, this required multiple nested loop structures.

To initialise the array SYSK to zero in Fortran 77 required:

```
   INTEGER ISYSK, JSYSK
   PARAMETER (ISYSK=200, JSYSK=20)
   REAL SYSK(ISYSK,JSYSK)
......

   DO 10 I=1,TOTDOF
      DO 20 J=1,HBAND
         SYSK(I,J) = 0.0D0
20    CONTINUE
10 CONTINUE
```

or

```
   CALL MATNUL(SYSK,ISYSK,JSYSK,TOTDOF,HBAND,ITEST)
```

using the FELIB routine. Other often used routines during the element matrix construction and assembly are MATADD, MATRAN and MATMUL. The array features of Fortran 90/95 provide a more compact and natural way of performing these actions. For example:

```
SYSK=0
```

can replace a call to MATNUL.

```
ELK = ELK + DTBD
```

would replace a call to MATADD.

Fortran 90/95 provides a number of basic array/matrix intrinsic functions. In the context of FELIB three of the more important are MATMUL, TRANSPOSE and DOT_PRODUCT. So

```
 CALL MATMUL(LDER,ILDER,JLDER,GEOM,IGEOM,JGEOM,
*            JAC,IJAC,JJAC,DIMEN,NODEL,DIMEN,ITEST)
```

can be replaced by the considerable simpler

```
JAC = MATMUL(LDER,GEOM)
```

and

```
CALL SCAPRD(GEOM(1,1),IGEOM,FUN,IFUN,NODE,X,ITEST)
```

replace by

```
X = DOT_PRODUCT(GEOM(1:NODEL,1),FUN)
```

Immediately one notices the reduction in actual arguments in the routines calls. Fortran 90/95 carries much more information about arrays than Fortran 77 did. Properties as an array's *rank* or an array's *shape* are readily available through a range of INTRINSIC functions. For example RANK, SHAPE, SIZE, LBOUND and UBOUND. These, together with the stricter conditions on the property matching between actual and dummy arguments in procedure calls, enable a considerable amount of information on arrays to be passed implicitly into procedures. Although these new features can simplify or modify many of the steps involved in an FELIB program there is a possible danger in compactness making the software more opaque. For example the construction of one quadrature point contribution to an element stiffness matrix could be written as:

```
ELK = ELK + MATMUL(TRANSPOSE(MATMUL(jacin,lder)), &
                   MATMUL(p,MATMUL(jacin,lder)))
```

Clearly this is more compact but it is hardly transparent. A balance will be required between succinctness and clarity particularly in software that is to be used as a teaching aid.

3.2 Dynamic storage allocation

An important feature of Fortran 90/59 is the ability to dynamically allocate storage to arrays at execution time. Fortran 90/95 provides two mechanisms to make this possible: the ALLOCATABLE array and the POINTER array. Both these types can be specified without size information:

```
REAL (wp), ALLOCATABLE ::  sysk(:,:)
REAL (wp), POINTER ::  sysm(:,:)
```

These statements define the two arrays sysk and sysm whose memory allocation can be specified thus:

```
ALLOCATE(sysk(100,10))
ALLOCATE(sysm(100,10))
```

In the first of these `sysk` has been defined as an `ALLOCATABLE` array and once memory is allocate `sysk` is a unique reference to this memory. The second allocate creates a similar reference to memory which can be used in the same way. However this may not be a unique reference. The nature of `POINTER`s allow multiple references to the same locations in memory. Although this would not be intended in the context of `FELIB` it is a possibility.

This is one of the features of `POINTER` arrays that the programmer must be aware. This non-uniqueness inherent in `POINTER`s can lead to unfortunate side effects.

Probably the most important of these is the possibility of *memory leaks* if the arrays are not allocated and deallocated assiduously. It is very easy to allocate space to a `POINTER` and then to later re-allocate a different section of memory to the same `POINTER` without de-allocating the former. In general compilers and run-time system will not flag this as an error. However the result is that there will be sections of memory reserved with no way of referencing it. This would have the potential of *eating* away the memory available to the application if the `ALLOCATE` is contained in some form of loop.

A second characteristic of `POINTER`s that can be a nuisance is that of initialisation. The `POINTER` declaration above defines a `POINTER` to an array but does not give an initial value to the `POINTER`. As a consequence the result from the intrinsic function `ASSOCIATED` is really undefined although, as is often the case, many compilers set these variables automatically to `NULL`. However, as this is the only mechanism through which it is possible to determine whether a `POINTER` has been associated with a target, it is essential that `POINTER` variables be explicitly initialised. This can be readily achieved using either the `NULL()` intrinsic or the `NULLIFY` statement. For example:

```
REAL (wp), POINTER ::  sysm(:,:)  => NULL()
```

will define and initialised to `NULL` the `POINTER sysm` whereas

```
NULLIFY(sysm)
```

performs the same task as an executable statement.

An important point to consider in the library context is how these array types can be passed into subroutines and functions and how their dynamic properties can be exploited. In the Fortran 90/95 standard it was not possible to pass `ALLOCATABLE` arrays as dummy arguments. Passing `POINTER` arrays was allowed. However, during 2001 an extension to allow the passing of `ALLOCATABLE` arrays into procedures was proposed [8]. This was excepted by the Fortran Standards body but as of yet only a few Fortran 90/95 compilers support for this feature. In a series of simple checks on the passing of `ALLOCATABLE` arrays into procedures using a number of compilers (Intel `ifc`, Lahey `lf95`, `NAG f95`, `DEC f95`): one gave an error message (ifc), one gave a warning and continued compilation (`NAG f95`), two compiled without warnings (`lf95`, `DEC f95`). Only two the those that compiled and linked successfully executed correctly (Lehay `lf95`, `NAG F95`).

Not supporting this feature stops the programmer passing an unallocated `ALLOCATABLE` array into procedure and making the allocation within the procedure. Defining arrays as `ALLOCATABLE` is one method of providing dynamic arrays within Fortran 90/95. However, if the arrays within `FELIB90` are defined as `ALLOCATABLE` to provide this functionality it will inhibit dynamically allocated intermediate arrays as described above in Section 2 as desirable. To overcome this problem all the arrays within `FELIB90` will be of a `POINTER` type. Although this may lead to problems of memory leaks, by the use of a stack of dynamic arrays and a type of semi-automatic garbage collection, these potential problems can be minimised from the library's point of view. However it will still be possible for the user to fall foul of these difficulties.

3.3 Optional arguments

Another important new feature of Fortran 90/95 is the provision of optional dummy arguments in procedure calls. These can shorten the argument lists of library routines significantly. A simple example of the use of optional arguments is the `PRTGEO` routine. This routine outputs the coordinate array `COORD`. The full argument list would be:

```
CALL PRTGEO(coord,totnod,dimen,nout,itest)
```

This can be reduce to

```
CALL PRTGEO(coord)
```

if the size and shape of `coord` matches exactly the element coordinate data and there is a default output channel `nout`. `totnod` and `dimen` can be determined using the intrinsic function `SIZE`.

A design goal of `FELIB90` has been to reduce required arguments lists to a minimum whilst maintaining flexibility and control through optional arguments.

Because of the nature and implementation of optional arguments in Fortran 90/95 the head of most routines has a group of code dealing with optional arguments. For an optional argument `arg1` it takes the form

```
! Dummy argument
INTEGER, OPTIONAL :: arg1
.......
! Local variables
INTEGER :: larg1
.......
IF (PRESENT(arg1))THEN
larg1 = arg1
ELSE
larg1 = default_arg1
ENDIF
```

where `larg1` is the local variable associated with `arg1`. It should be note that `arg1` cannot be referenced if it is not present. It may only be reference through the `PRESENT` intrinsic.

In general `FELIB90` only has scalars as optional arguments. To aid clarity `FELIB90` uses a simple set generic routines to assign default values to optional arguments. These are provided through the module `mod_setopt`. The part of the code is shown below.

```
MODULE mod_setopt

USE FELIB_GLOBALS, only : wp

INTERFACE setopt
   MODULE PROCEDURE complex_setopt
   MODULE PROCEDURE real_setopt
   MODULE PROCEDURE integer_setopt
   MODULE PROCEDURE character_setopt
END INTERFACE

CONTAINS

..........................

SUBROUTINE real_setopt(dummy,value,actual)
IMPLICIT NONE
REAL (wp) :: dummy, value
REAL (wp) , OPTIONAL :: actual
```

```
IF( PRESENT(actual)) THEN
  dummy=actual
ELSE
  dummy=value
ENDIF

END SUBROUTINE real_setopt

SUBROUTINE integer_setopt(dummy,value,actual)
IMPLICIT NONE
INTEGER :: dummy, value
INTEGER , OPTIONAL :: actual


IF( PRESENT(actual)) THEN
  dummy=actual
ELSE
  dummy=value
ENDIF

................................

END MODULE mod_setopt
```

Now the setting of optional arguments is reduced to:

```
call setop(larg1,default_arg1,arg1)
```

Use of this routine clears up the opening statements of the library routines.

3.4 Generic routines

As can be seen from the above another very useful feature of Fortran 90/95 is the definition of *generic* interfaces: functions with differing argument types being called by the same *generic* name. This an example of the idea of *overloading* in Fortran 90/95. `FELIB90` uses generic routines where ever thought useful. Routines such as `MATNUL` can be made generic and thus capable of operating on `INTEGER`, `REAL` and `COMPLEX` arrays. As seen above `setopt` is define as a generic routine capable of operating on `INTEGER`, `REAL`, `COMPLEX` and `CHARACTER` variables.

In the Fortran 66 and Fortran 77 versions of the library `COMPLEX` variables were treated as order pairs of `REAL` values and required their own set of manipulation routines such as `CMTNUL` and `CSYSOL`. Fortran 90/95 defines a `COMPLEX` data type which much simplifies some of the routines through the use of *generic* interfaces. As a consequence this does require that many routines are defined as generic routines.

3.5 Intrinsic functions

Where ever possible the standard Fortran 90/95 `INTRINSIC` functions have been used in the Level 0 Library routines. For example `HUGE` is used to obtain the largest `REAL` and largest `INTEGER` available to the library. `EPSILON` is use to determine the smallest value for which $1 + \epsilon > 1$. There are one or two problems with using the standard Fortran 90/95 intrinsics. One of the design goal of `FELIB90` to provide an easy migration path requires that potentially obsolete routines be provide. `MATMUL` is probably the most important of these. The name `MATMUL` conflicts with the Fortran 90/95 intrinsic of the same name and will cause compiler warnings or errors if `MATMUL` is either declared or used in this way.

To circumnavigate this problem a new interface to the Fortran 90/95 intrinsic has been defined. This is another feature of Fortran 90/95: procedures defined in modules can have their names aliased. FELIB90 defines an alias to the intrinsic MATNUL as MAXTRIX_MULTIPLY: (similar to DOT_PRODUCT and TRANSPOSE in style). This is defined in the simple module

```
MODULE mod_matmul_intrinsic

  INTRINSIC MATMUL

END MODULE mod_matmul_intrinsic
```

and a USE statement at the head of the FELIB90 module.

```
USE mod_matmul_intrinsic, ONLY : matrix_multiply => matmul
```

These statements essentially map the name matrix_multiply onto the intrinsic function MATMUL. Through this mechanism the intrinsic MATMUL is made available to FELIB90 programs. Thus

```
CALL matmul(lder,geom,jac)
```

and

```
jac=MATRIX_MULTIPLY(lder,geom)
```

are equivalent.

4 Management of dynamic arrays

As mentioned above it was decided to implement the library using POINTER arrays to provide the maximum flexibility in managing intermediate and temporary arrays even though this does have some potential pit falls. However these can be minimised by providing the user with a collection of array management routines although it is not really expected that the user would make use of these routines. It is thought that as more compilers adopt the recommendations of the ISO TR 15581 this approach can be easily modified to make use of ALLOCATABLE arrays.

In this version of FELIB90 a module (mod_space) of memory management routines based on POINTER arrays has been implemented. The module contains three types of routine: one set the create vectors and arrays, a second set to destroy and another the check the memory allocations.

The module mod_space maintains and manages a set of arrays which point to specific arrays and vectors. The basic allocation process starts with a call to create within a subroutine.

```
CALL create(elk,dofel,dofel)
```

The create routine performs the following steps:

loop *list of current allocations for an association*
  to elk *using the* associated *intrinsic.*
  if *a current association exists* **then**
    if *size and shape of allocated space is ok* **then**
      return to calling routine
    else
      deallocate using destroy
      create new space using create *(recursively)*
    endif
  else
    find next free pointer in list
    allocate new space
    associate with target name
    mark pointer as in use
  endif
end loop

`create` is a generic routine and provides for vectors and two-dimensional arrays of the basic types required by `FELIB90`: `real, integer` and `complex`. The module `mod_space` also provides: `destroy`, a generic routine for deallocating memory space and disassociating pointers and targets and `check`, a routine to check on the status of a vector or array.

5 Library construction

`FELIB90` has taken on a modular approach to its design - the library is a `MODULE` to be `USE`d by the user program. All the `FELIB90` routines are contained in their own modules and these are `USE`d by the `FELIB90` module to build the complete library. A consequence of this is that all the interfaces of the library routines are explicit and can be used by the compiler at compile and run time to provide diagnostics.

This approach has some impact on developing the library in terms of compilation but these are minimal as `FELIB90` is a *small* library. There are benefits in as much that the developer is not required to generate interface blocks for the user to reference.

As mentioned above each routine of `FELIB90` is a module in its own right. This makes providing generic interfaces straightforward and ensures explicit interfaces. So for example the routine `MATNUL` has the following code

```
     MODULE mod_matnul

   !----------------------------------------------------------------------
   ! PURPOSE
   !      MATNUL creates and sets matrix A to the null matrix

   ! HISTORY

   !      Copyright (C) 2000 : CCLRC, Rutherford Appleton Laboratory
   !                           Chilton, Didcot, Oxfordshire OX11 0QX

   !      Release 1.0   2 Jul 2000 (CG)

   ! ARGUMENTS in
   !      M       number of rows of A set to zero (OPTIONAL)
   !      N       number of columns of A set to zero (OPTIONAL)
   !      ITEST   error checking option (OPTIONAL)

   ! ARGUMENTS out
   !      A       array set to zeros

   ! ROUTINES called

   !**********************************************************************

         USE felib_globals,only : wp
         USE mod_space,only : create
         USE mod_setopt,only : setopt

         PRIVATE
         PUBLIC matnul

         INTERFACE matnul
           MODULE PROCEDURE complex_matnul
           MODULE PROCEDURE real_matnul
           MODULE PROCEDURE integer_matnul
         END INTERFACE
```

```
      CHARACTER (5) :: srname = 'MATNUL'

    CONTAINS

.............................

      SUBROUTINE real_matnul(a,m,n,itest)
        IMPLICIT NONE

! Dummy arguments

        INTEGER, OPTIONAL :: m, n, itest
        REAL (wp), POINTER :: a(:,:)
        INTENT (IN) :: m,n
        INTENT (INOUT) :: itest

! Local variables

        INTEGER :: i, ierror, j, mm, nn
        CHARACTER (5) :: srname = 'MATNUL'

! Check optional arguments and association of A

        CALL setopt(mm,size(a,1),m)
        CALL setopt(nn,size(a,2),n)

! Create A if necessary then initialise

        CALL create(a,mm,nn)

        a(1:mm,1:nn)=0

      END SUBROUTINE real_matnul

.............................

    END MODULE mod_matnul
```

As can been seen from the source MATNUL is a generic routine that will if necessary create the storage to be associated with a variable using the routine CREATE and manages optional arguments using the SETOPT routine. This is type of FELIB90 routines.

The final step is to build the full library, FELIB90. Again FELIB90 is a module in its own right and contains a sequence of USE statements to include all the library routines - one per routine. Below is show the FELIB90 code.

```
    MODULE felib90

! MODULE felib90 is the main defining modules of FELIB90. All
! user callable routines are included here.

! System

      USE felib_globals, ONLY : wp
      USE mod_space, ONLY : create, destory, check

! Routines

      USE mod_bndwth, ONLY : bndwth
      USE mod_qqua4, ONLY : qqua4
```

```
      USE mod_elgeom, ONLY : elgeom
      USE mod_quam4, ONLY : quam4
      USE mod_quam8, ONLY : quam8
      USE mod_scaprd, ONLY : scaprd
      USE mod_matmul, ONLY : matmul
      USE mod_matnul, ONLY : matnul
      USE mod_matran, ONLY : matran
      USE mod_matvec, ONLY : matvec
      USE mod_prtval, ONLY : prtval
      USE mod_asrhs, ONLY : asrhs
      USE mod_assym, ONLY : assym

      .............................

      USE mod_chosol, ONLY : chosol
      USE mod_direct, ONLY : direct
      USE mod_matinv, ONLY : matinv
      USE mod_getgeo, ONLY : getgeo
      USE mod_gettop, ONLY : gettop
      USE mod_prtgeo, ONLY : prtgeo
      USE mod_prttop, ONLY : prttop
      USE mod_errmes, ONLY : errmes
      USE mod_asful, ONLY : asful
      USE mod_vecnul, ONLY : vecnul
      USE mod_vecadd, ONLY : vecadd
      USE mod_matadd, ONLY : matadd
      USE mod_setopt, ONLY : setopt

! Redefinition of intrinsic MATMUL

      USE mod_matmul_intrinsic, ONLY : matrix_multiply => matmul

    END MODULE felib90
```

6 Program definitions

To help make the programs more readable the definitions of many of the standard intermediate arrays such as `ELK` and `GEOM` have been collected together into a definitions module - for example `def3p1`. These are provided with the program file and are to be compiled with the user program. Clearly the user can add to these definitions if thought useful or replace them with their own specific definitions. An example of this type of module is given below.

```
    MODULE def3p1

! All arrays within FELIB programs are defined as POINTERs. This
! enables dynamic allocation within FUNCTIONS and SUBROUTINES. Many
! routines automatically ALLOCATE space for a current set of intermediate
! variables use in the solution process.

! These variables are created through a library of space management routines
! which includes garbage collection. These routines do not inhibit the user
! defining his own arrays locally or by using the space creataion routines.

! This modules contains a standard set of variable definitions
! often found in basic FELIB programs.

      USE felib_globals, ONLY : wp
```

```
! Allocatable arrays - dependent on element types and problem
!                       dimensionality

      INTEGER, POINTER ::     &
        steer(:)   => null()    ! Element steering vector
      REAL (wp), POINTER ::   &
        elq(:)     => null(), & ! Element vector
        fun(:)     => null(), & ! Shape function vector
        xy(:)      => null(), & ! Global coordinate vector
        geom(:,:)  => null(), & ! Local geometry array
        elk(:,:)   => null(), & ! Element stiffness arrat
        lder(:,:)  => null(), & ! Local derivatives of shape funtions
        jac(:,:)   => null(), & ! Transformation jacobian
        jacin(:,:) => null(), & ! Inverse of JAC
        geomt(:,:) => null(), & ! Element geometry transposed
        wght(:)    => null(), & ! Quadrature weights
        abss(:,:)  => null(), & ! Quadrature abssise
        p(:,:)     => null(), & ! Permeitvity array P
        pd(:,:)    => null(), & ! P transposed
        scvec(:)   => null(), & ! Element source vector
        gder(:,:)  => null(), & ! Global derivatives of shape functions
        dtpd(:,:)  => null(), & ! Element matrix
        gdert(:,:) => null()    ! Tranpose of GDERT


    END MODULE def3p1
```

7 An example program

In this section a complete FELIB90 Level 1 program is shown. The structure is very much like that of the Fortran 77 programs. When compared with the Fortran 77 version it can be seen the actual argument lists of the routines are much shorter. A comparison with the Fortran 77 program SEG3P1 [1] will also show the similarity of structure and therefore this program should be recognisable to existing users of FELIB.

Throughout this program the shortest possible argument list have been used and use has been made of all the system defaults provided by the library. In particular input/output channel numbers.

As mention above FELIB90 contains some redundant routines and code to aid the transition from FELIB to FELIB90 and also to aid clarity in a teaching context. For the programmer who wishes to move to a full Fortran 90/95 implementation a number of *Notes* have been added to each Level 1 program. The next section, Section 8, give the notes on this program.

```
      !*****************************************************************

1     PROGRAM seg3p1

      !*****************************************************************

      !     Copyright (C) 2003 : CLRC, Rutherford Appleton Laboratory
      !     Chilton, Didcot, Oxfordshire OX11 0QX

      ! N.B. The working precision of the current library is held
      !      in the variable WP. This must be used in all REAL
      !      declarations of variables used by FELIB90.

      !      The program also uses the standard FELIB90 values for
```

```
!      nin and nout.

!*****************************************************************

   USE felib90 ! Use FELIB90 all routines
   USE def3p1 ! Use standard SEG3P1 definitions
   IMPLICIT NONE

! Parameters

   REAL (WP), PARAMETER :: scale = 1.0E+10

! Local variables

   INTEGER :: bndnod, dimen, dofel, dofnod, hband, i, iquad, itest, j, &
     nele, nodel, nqp, totdof, totels, totnod
   REAL (WP) :: det, eta, quot, strgth, x, xi, y

! Allocatable arrays - mesh size dependent - user defined in the data

   INTEGER, POINTER :: bnode(:), nf(:,:), eltop(:,:)
   REAL (WP), POINTER :: bval(:), rhs(:), coord(:,:), sysk(:,:)

! Intrinsic functions

   INTRINSIC abs

!     Initialisation of POINTERS to main arrays

   NULLIFY (bnode,nf,eltop)
   NULLIFY (bval,rhs,coord,sysk)

!     Set error checking flag

   itest = 0

!     **********************
!     *                    *
!     * Input Data Section *
!     *                    *
!     **********************

!     Input of nodal geometry

   CALL getgeo(coord,totnod,dimen)
   CALL prtgeo(coord)

!     Input of element topology

   CALL gettop(eltop,totels)
   CALL prttop(eltop)

!     Input of permeabilities, construction of permeability matrix P
!     and source strength

   CALL matnul(p,dimen,dimen)
   WRITE (nout,'(/A)') 'Permeabilities'
   READ (nin,'(2F10.0)') (p(i,i),i=1,dimen)
   WRITE (nout,'(2F10.5)') (p(i,i),i=1,dimen)
```

```
      WRITE (nout,'(/A)') 'Source Strength'
      READ (nin,'(F10.0)') strgth
      WRITE (nout,'(F10.5)') strgth

!     Input of number of degrees of freedom per node, input of
!     boundary conditions and construction of nodal freedom array NF

      WRITE (nout,'(/A)') 'Degrees of freedom per node (DOFNOD)'
      READ (nin,'(I5)') dofnod
      WRITE (nout,'(I5)') dofnod

!    Input boundary conidtions

      WRITE (nout,'(/A)') 'Boundary Conditions'
      READ (nin,'(I5)') bndnod
      WRITE (nout,'(I5)') bndnod

      CALL vecnul(bnode,bndnod)
      CALL vecnul(bval,bndnod)
      DO i = 1, bndnod
        READ (nin,'(I5,F10.0)') bnode(i), bval(i)
        WRITE (nout,'(I5,F10.5)') bnode(i), bval(i)
      END DO

! Setup nodel freedom array

      CALL matnul(nf,totnod,dofnod)
      totdof = 0
      DO i = 1, totnod
        DO j = 1, dofnod
          totdof = totdof + 1
          nf(i,j) = totdof
        END DO
      END DO

!     Calculation of semi-bandwidth

      CALL bndwth(eltop,nf,hband)

!  ************************************
!  *                                  *
!  * System Stiffness Matrix Assembly *
!  *                                  *
!  ************************************


! System matrices setup and initalise : rhs, sysk

      CALL matnul(sysk,totdof,hband)
      CALL vecnul(rhs,totdof)

! Setup quadrature

      CALL qqua4(wght,abss,nqp)

! Begin main element loop

      DO nele = 1, totels !Loop over all elements
```

```
    nodel = eltop(nele,2)
    dofel = dofnod*nodel
    CALL elgeom(nele,eltop,coord,geom)

!     Integration loop for element stiffness using NQP quadrature
!     points

    CALL matnul(elk,dofel,dofel)
    CALL vecnul(elq,dofel)
    CALL vecnul(scvec,dofel)

    DO iquad = 1, nqp !Numerical integration

!     Form linear shape function and space derivatives in the local
!     corrdinates. Transform local derivatives to global coordinate
!     system

      xi = abss(1,iquad)
      eta = abss(2,iquad)
      CALL quam4(fun,lder,xi,eta)

      CALL matran(geom,geomt)
      CALL matvec(geomt,fun,xy)
      x = xy(1)
      y = xy(2)

      CALL matmul(lder,geom,jac)
      CALL matinv(jac,jacin,det)
      CALL matmul(jacin,lder,gder)

!     Formation of element stiffness ELK

      CALL matmul(p,gder,pd)
      CALL matran(gder,gdert)
      CALL matmul(gdert,pd,dtpd)

      quot = abs(det)*wght(iquad)
      dtpd = dtpd*quot
      scvec = fun*src(x,y,strgth)*quot

      CALL matadd(elk,dtpd)
      CALL vecadd(elq,scvec)

    END DO !Loop over quadrature points - iquad

!     Assembly of system stiffness matrix

    CALL direct(nele,eltop,nf,steer)
    CALL assym(sysk,elk,steer)
    CALL asrhs(rhs,elq,steer)

  END DO !Loop over elements - nele

!     ********************
!     *                  *
!     * Equation Solution *
!     *                  *
!     ********************
```

```
     !     Modification of stiffness matrix and right-hand side to
     !     implement boundary conditions

81        DO i = 1, bndnod
82          j = bnode(i)
83          sysk(j,hband) = sysk(j,hband)*scale
84          rhs(j) = sysk(j,hband)*bval(i)
85        END DO

     !     Solution of system matrix for the nodal values of the
     !     potential

86        CALL chosol(sysk,rhs)

87        WRITE (nout,'(/A)') 'Nodal Potentials'
88        CALL prtval(rhs,nf)

89        STOP

90      CONTAINS

     !*****************************************************************
     ! Source function

91        FUNCTION src(x,y,strgth)

92          USE felib90

93          IMPLICIT NONE

     ! Dummy arguments

94          REAL (wp) :: src
95          REAL (wp) :: strgth, x, y
96          INTENT (IN) strgth, x, y

97          src = 0.0D0
98          IF ((x>1.0D0) .AND. (x<2.0D0) .AND. (y>1.0D0) .AND. (y<2.0D0)) &
99            src = strgth

100        END FUNCTION src

101      END PROGRAM seg3p1
```

8 An example of program *notes*

For each Level 1 Programs a set of *Notes* has been developed. These indicate how a programmer could modify the Level 1 Programs to use more fully the features of Fortran 90/95. These include the use of explicit `ALLOCATE` statements for memory allocations and the use of other Fortran 90/95 intrinsics.

Statement 15: The routine `getgeo` allocates memory for `coord` depending on the data. The total number of nodes, `totnod` and the dimensionality of the problem, `dimen` are returned. The array `coord` can be defined and initailised using the statements

```
ALLOCATE(coord(totnod,dimen))
coord=0.0
```

provided `totnod` and `dimen` are known.

Statement 17: The routine `gettop` allocates memory for `eltop` depending on the data. The total number of elements, `totels` is returned. The array `eltop` can be defined and initialised using the statements

```
ALLOCATE(eltop(totels,max_nodel+2))
eltop=0
```

provided `totels` and `max_nodel` are known. `max_nodel` is set to the largest number of nodes in an element (`nodel`) for the given mesh. For simple meshes this will be equal to `nodel`.

Statement 19: The routine `matnul` allocates memory and initialises the array `p`. This can be performed using the statements

```
ALLOCATE(p(dimen,dimen))
p=0.0
```

Statement 32: The routine `vecnul` allocates memory and initialises the array `bnode`. This can be performed using the statements

```
ALLOCATE(bnode(bndnod))
bonde=0
```

Statement 33: The routine `vecnul` allocates memory and initialises the array `bval`. This can be performed using the statements

```
ALLOCATE(bval(bnnod))
bval=0.0
```

Statement 38: The routine `matnul` allocates memory and initialises the array `nf`. This can be performed using the statements

```
ALLOCATE(nf(totnod,dofnod))
nf=0
```

Statement 47: The routine `matnul` allocates memory and initialises the array `sysk`. This can be performed using the statements

```
ALLOCATE(sysk(totdof,hband))
sysk=0.0
```

Statement 48: The routine `vecnul` allocates memory and initialises the array `rhs`. This can be performed using the statements

```
ALLOCATE(rhs(totdof)
rhs=0.0
```

Statement 54: The routine `matnul` allocates memory and initialises the array `elk`. This can be performed using the statements

```
ALLOCATE(elk(dofel,dofel))
elk=0.0
```

Statement 55: The routine `vecnul` allocates memory and initialises the array `elq`. This can be performed using the statements

```
ALLOCATE(elq(dofel))
elq = 0.0
```

Statement 56: The routine vecnul allocates memory and initialises the array scvec. This can be performed using the statements

```
ALLOCATE(scvec(dofel))
scvec = 0.0
```

Statement 61 to 62 : The routine matran transposes the array geom. It creates and initialises memory for the intermediate array geomt. This can be performed using the intrinsic TRANSPOSE:

```
geomt=TRANSPOSE(geom)
```

provided geomt has been created and is of a suitable size and shape.

The routine matvec post multiplies the matrix geomt by the vector fun. This can be performed using the intrinsic MATMUL which been mapped onto the function MATRIX_MULTIPLY in FELIB90. matvec will create and initial memory for the intermediate array xy so xy will need to be created.

```
xy=MATRIX_MULTIPLY(geomt,fun)
```

Statements 61 to 64: An alternative to these statements is to calculate x and y directly through an array section using either the FELIB90 routine scaprd

```
CALL scaprd(geom(1:nodel,1),fun,x)
CALL scaprd(geom(1:nodel,2),fun,y)
```

or the intrinsic DOT_PRODUCT:

```
x=DOT_PRODUCT(geom(1:nodel,1),fun)
y=DOT_PRODUCT(geom(1:nodel,2),fun)
```

This is possible because of the way in which Fortran stores its arrays in memory.

Statement 65 to 67: The routine matmul multiplies the arrays lder and geom together. This can be performed using the intrinsic MATMUL. This has been mapped onto the function MATRIX_MULTIPLY in FELIB90.

```
jac=MATRIX_MULTIPLY(lder,geom)
```

jac must be allocated with suitable size and shape.

The routine matmul multiplies the arrays jacin and lder together. This can be performed using the intrinsic MATMUL which been mapped onto the function MATRIX_MULTIPLY in FELIB90.

```
gder=MATRIX_MULTIPLY(jacin,lder)
```

gder must be allocated with suitable size and shape.

Statements 68 to 75: The section of code deals with the construction and assembly the element stiffness matrix dtpd. There two approaches to replacing these statements: firstly by mirroring the FELIB90 routines using intrinsics

```
pd=MATRIX_MULTIPLY(p,gder)
gdert=TRANSPOSE(gder)
dtpd=MATRIX_MULTIPLY(gdert,pd)
scvec=fun*src(x,y,strgth)*quot
```

remembering that the intrinsic matnul is mapped to the FELIB90 routine MATRIX_MULTIPLY or by combining these three steps into a single compound statement

```
dtpd=MATRIX_MULTIPLY(TRANSPOSE(gder),MATRIX_MULTIPLY(p,gder))
```

dtpd must be allocated with a suitable size and shape. The final collections can be performed using

```
elk = elk + dtpd
elq = elq + scvec
```

instead of using the FELIB90 matadd and vecadd routines.

9 The transformation process

FELIB had been developed originally in Fortran 66 - a fixed source form with all upper case characters - no IF - THEN - ELSE constructs and the use of GOTO statements. Moreover FELIB was in strict Fortran 66 having been verified by QA tools such PFORT [13].

The basic transformation process had the following steps:

- Compile and run on test data. Save results from tests.
- Verify Fortran 66 code against standard (using PFORT or the TOOLPACK tool istpf)
- The conforming Fortran 66, which includes many GOTO blocks was restructured using the TOOLPACK tool istst and plusFORT SPAG program. With the correct options SPAG could transform directly to Fortran 90/95 but it was thought that staging through restructured Fortran 77 would allow some result testing.
- Re-compilation and testing of new Fortran 77 using test data.
- Transformation of comments etc with istuc.
- Re-compilation and testing.
- Transformation into free format Fortran 90/95 using SPAG. Others tools in the NAGWare suite could have been used.
- Compilation and testing of new source code using test data.

At this point we have transform the Fortran 66 code into free format Fortran 90/95. Along the route each tool will have detected some problems with the code that required corrections. In general the corrections were made to the original Fortran 66 and the process repeated.

At this point the structural elements of the re-design were implemented. As the overall structure and functionality of the example programs and library were not going to changed the modification at this point were made on a routine by routine basis using a set of simple edit scripts. For examples: changing the type of variable SRNAME, the routine name, from DOUBLE PRECISION to CHARACTER*6 were simple awk scripts.

```
DOUBLE PRECISION SRNAME
DATA SRNAME /' SRNAME '/
```

became

```
CHARACTER*6 srname = "SRNAME"
```

There were many other similar examples.

As with many of the other changes these had to be made on a routine by routine basis following some basic re-design rules.

10 Some notes on development tools

In this section we review some of the software tools used in the re-engineering of `FELIB`. These tools were used to process the source code and check the executables. Regular use of tools such as these will speed development by helping to prevent or to find errors in user programs and in making source code easier to read and understand. Real benefits can be gained from the use of the tools during maintenance of existing software as the checks help to ensure that modifications are properly applied and that the style of the code remains consistent.

10.1 Source code transformers

One of the major stumbling blocks in any re-design or re-engineering process is the thought that you have thousands of line of code to change. This has been well recognised in the Fortran community as the use of Fortran 90/95 has developed. For Fortran 77 many source code analysis and restructuring tools had been developed: notably through the `TOOLPACK` Project. As Fortran 77 was a fully compliant subset of Fortran 90/95 it was generally straightforward to develop source code transformers to transform the fixed format Fortran 77 programs to either fixed or free format Fortran 90/95.

Three transformers have been used in this project: the `TOOLPACK` suite, the `spag` program from the `plusFORT` suite of Polyhedron Software Ltd and the `f95` Declaration Standardiser of the `NAGware` Tools from the Numerical Algorithms Group Ltd. All these tools will take a fixed format Fortran 77 program and transform the source. `TOOLPACK` will generate well structured and formated Fortran 77 from *old* Fortran 66 and the other two will transform conforming Fortran 77 into reasonable Fortran 90/95 in either a fixed of free format.

However problems do arise if the source Fortran is not conforming. So often elements of the Unix C pre-processor `cpp` are used as version control constructs in Fortran programs. `spag` from `plusFORT` was more tolerant of language dialects than `decs95`. However both tools were very useful in producing free format Fortran 90/95 code from the original Fortran 77.

10.2 The `TOOLPACK` Suite

For some programs Fortran 66 is the implementation language: the use of `GOTO`s, arithmetic `IF`s and computed `GOTO` statements.

`TOOLPACK` is a suite of software tools designed in the 1980s to support the Fortran programmer. In this context, a 'software tool' is a utility program to assist in the various phases of constructing, analysing, testing, adapting, or maintaining a body of Fortran software. Typically, the input to such a tool is your Fortran source code. The tool processes this and produces output that may have one or both of the following forms:

- A report that gives an analysis of the input program, e.g. a summary of the types of statements used; this type of tool is called a static analyser.
- A modified version of the input program; in this case, the tool is called a transformer. An example is a formatter which improves the appearance of the code.

In some cases the input may be test data, documentation, or a report generated by a previously applied tool. Tools that assist directly in preparing documents are usually called documentation generation aids. These and other tools serving utility functions all have an important role to play and so, even if they do not process a program directly, they are still regarded as programming aids.

Further examples of the software tools provided include:

- A text editor with Fortran 77 oriented features.
- A transforming tool that standardises the declarative part of a Fortran program.

- An instrumenter that modifies the program by inserting monitoring and other control statements. The instrumented program is then compiled and executed, and data is gathered that is used to generate reports. Execution of an instrumented program is an example of dynamic analysis.

The `TOOLPACK` Suite is public domain and is easily obtained although they are now of limited use as the community migrates the Fortran 90/95. Some of the tools contained in `TOOLPACK` have been packaged into the `NAGWare` Fortran 77 Tools. See

    http://www.nag.co.uk/public/tpack.asp

for details.

10.3 `plusFORT`

`plusFORT` is a suite of tools for Fortran programmers. The main components are summarised below.

- `SPAG`: The primary analysis and restructuring tool.
- `GXCHK`:A global static analysis tool.
- `CVRANAL`:A coverage analysis reporting tool.
- `QMERGE`:A version selection tool.
- `QSPLIT`:A small file-splitting utility.
- `AUTOMAKE`:A tool for minimal recompilation.

`SPAG`, the `plusFORT` restructuring tool, was the one tool that was extensively used. It can unscramble spaghetti Fortran 66 code, and convert it to structured Fortran 77. It also converts back and forth between standard Fortran 77, and code with VAX and Fortran 90/95 extensions such as `DO WHILE`, `ENDDO`, `CYCLE`, `EXIT` and `SELECT CASE`.

`SPAG` does not change the meaning of a program, or even the order in which statements are executed; it does change the way the program logic is written down, making it much easier to understand and maintain. Blocks of code are reordered so that logically related sections are physically close, and jumps in control flow are minimised. `SPAG` may also replicate small code fragments where this improves the re-structured code. `SPAG` computes complexity metrics before and after restructuring. `SPAG` contains a powerful code beautifier, with dozens of options controlling spacing, case, labels, indentation, use of `CONTINUE` etc. You can use `SPAG` to switch back and forth between the Fortran 77 and Fortran 90/95 source forms.

There are over 100 configuration options which allow you to customise `SPAG` output to local conventions and requirements. See

    http://www.polyhedron.com

for details.

10.4 `NAGWare` Tools

The `NAGWare` Fortran Tools provide users with the ability to analyse and transform Fortran 77 and Fortran 90/95 codes. These tools can be used in a range of ways:

- Quality assurance standardisation - enforcing coding standards
- Porting to new platforms

- Converting from fixed format Fortran 77 to free format Fortran 95
- Normal day-to-day development

The `NAGWare` Fortran Tools suite consists of the following components:

- `NAGWare` *Fortran 95 Tools*: The `NAGWare` f95 Tools provide analysis and transformational tools that accept as input Fortran 77 and fixed or free format Fortran 95. Output from the transformational tools is always free format, so these tools are effectively fixed to free format translators. This set of tools provides analysis capabilities that include a call graph generator and transformational tools that include a configurable pretty printer, declaration standardiser and precision standardiser.
- `NAGWare` *Fortran 77 Tools*: The `NAGWare` f77 Tools are a collection of tools for processing, analysing and transforming Fortran 77 source code. The tools accept as input standard conforming Fortran 77 with some common extensions and output fixed format Fortran 77. So these tools are used where it is not desired to move forward to free format Fortran 95. The analysis capabilities which include a portability verifier (standard conformance checker) and call graph generator, can be useful as a first step in porting code from Fortran 77 to Fortran 95 or as an aid to further development work on the Fortran 77 code.

The transformational tools include a configurable pretty printer, declaration standardiser and precision transformer. See

    http://www.nag.co.uk

for details.

10.5 Memory checking

One of the major sources of difficulty in using dynamic arrays in Fortran 90/95 is memory leakage. Without a very careful count of `ALLOCATES` and `DEALLOCATES` it is very easy for leaks to arise. This is often true of library software but it is particularly true of `FELIB90` as it attempts to *hide* much of its dynamic memory management.

During this develop the `memprof` program has been used to help track memory leaks. The program is freely available over the Internet from

    http://www.gnome.org/projects/memprof/

and is easily installed and used. `memprof` is a tool for profiling memory usage and finding memory leaks. Its two major features are:

- It can generate a profile how much memory was allocated by each function in your program.
- It can scan memory and find blocks that you have allocated but are no longer referenced anywhere.

`memprof` works by pre-loading a library to override the C library's memory allocation functions and does not require you to recompile your program.

One advantage `memprof` has over some other similar tools that are available is that it has a nice GUI front-end and is relatively easy to use. It appears to work fine on `FELIB90` although its diagnostic output, instruction addresses, is not particularly useful. It does however give a useful way of indicating the presence of memory leaks.

10.6 Case transformer: `istuc`

As noted above `FELIB` was originally in a the fixed, upper case format of Fortran 66. As a results all the comments in the software were upper case. Given that Fortran 77 and Fortran 90/95 allowed mixed cases it was thought useful to transform the comments into mixed case. The comments also often referenced variable names. It was thought helpful if these could be left, together with a few other key words, in upper case.

To make this process as automatic as possible the utilities and libraries of the `TOOLPACK` suite were used the develop an addition tool to preform this task. `TOOLPACK` provided all that was need to parse the software, edit and reformed the comment lines and re-construct the programs source form. `istuc` was used within the `TOOLPACK` command environment `istce`. The following `TOOLPACK` script was used to process each file in `FELIB`

```
lx #&1.f,&1-lx.lst,&1-lx.tkn,&1-lx.cmt
uc &1-lx.tkn,&1-lx.cmt,&1-uc.tkn,&1-uc.cmt,u-words
pl &1-uc.tkn,&1-uc.cmt,#&1.pol,-
```

the script being called thus

```
ce:com/edit asful
```

for the routine `asful.f`. `lx` is the `TOOLPACK` lexical analyser which decomposes the source code and generates a listing stream (`.lst`), token stream (`.tkn`) and a comment stream (`.cmt`). `uc` processes the token and comment streams and passes then on to the `TOOLPACK` polish tool, `pl`, which reconstitutes the source code. `lx` and `pl` are standard `TOOLPACK` tools.

Although one might wish to preform similar processes in Fortran 90/95 no tool set like that of `TOOLPACK` is available. Such tasks would then need performing languages such as `perl` [14] or `python` [15].

11 Conclusions

In this report we have described the re-design of the Finite Element Library in Fortran 90/95 and explained the major design choices. The overall structure of the library has been discussed and the use of generic routines and dynamic memory allocation explored.

We believe that the resulting library will provide a useful addition to the vast body of Fortran 90/95 computational engineering software available to the community.

The Fortran 90/95 version of the Finite Element Library will be made available to the research community through the Group's Web site at:

```
http://www.mathsoft.cse.clrc.ac.uk/felib90
```

At present only a small subset of `FELIB` is available in Fortran 90/95 but this will grow in time. All additional `FELIB90` material such as software and documentation, will be made available at this address.

One final comment. Fortran 90/95 has the potential to design and implement programs in an object oriented approach. The work in this report is a stepping stone to an object oriented version of the Finite Element Library.

References

[1] C. Greenough, K. Robinson, *The Finite Element Library - Level 1 Documentation - Version 3*, Rutherford Appleton Laboratory, 1990

[2] C. Greenough, K. Robinson, *The Finite Element Library - Level 0 Documentation - Version 3*, Rutherford Appleton Laboratory, 1990

[3] C. Greenough, "The Finite Element Library - from design to realisation", Rutherford Appleton Laboratory, Technical Report, RAL-85-011, 1985

[4] C. Greenough, C.J. Hunt, PARFEL - An Extension of the NAG/SERC Finite Element Library for Multi-Processor Message Passing System, Rutherford Appleton Laboratory Report, RAL-90-070, 1990

[5] I.M. Smith, *Programming the Finite Element Method with Application in Geo-mechanic*, John Wiley, Chichester, 1982

[6] I.M. Smith, D.V. Griffiths, *Programming the Finite Element Method*, 3rd Ed, John Whiley, Chichester, 1998

[7] I.M Smith, A General Purpose System for Finite Element Analysis in Parallel, Engineering Computations, v17, No1, pp75 - 91, 2000

[8] M. Cohen (ed.), "Information technology - Programming languages - Fortran - Enhanced data type facilities", ISO/IEC TR 15581(E), ISO, Geneva

[9] *TOOLPACK* see `http://www.nag.co.uk/public/tpack.asp`

[10] *plusFORT* Reference Manual, Polyhedron Software Ltd
(see `http://www.polyhedron.com/pf/manual/index.html`)

[11] *NAGWare* see `http://www.nag.co.uk/nagware/NQ.asp`

[12] M. Metcalf, J. Reid, *Fortran 90 Explained*, Oxford University Press

[13] B.G. Ryder, The PFORT Verifier, Software, Practice and Experience, **4**, pp359-378, 1974.

[14] R.L. Schwartz, T. Christiansen, *Learning Perl*, O'Reilly & Associates Inc, 1997

[15] G. Van Rossum, F.L. Drake Jr (Editor), *An Introduction to Python*, Network Theory Ltd., April 2003

Appendix A - Reduced Fortran 90/95 version of Seg3p1

In this section we provide another version of Seg3p1 using more more of the features available in Fortran 90/95. We have made some of the substitutions using the Notes in Section 8 and placed multiple statements on source lines. We have introduced a number of additional *intrinsic* functions: `matrix_inverse` and `matrix_determinant`. This has made the assembly loop more compact.

It will be noticed that `ALLOCATE` and `DEALLOCATE` statements have been introduced to manage the dynamic memory arrays. In this program the `ALLOCATE` and `DEALLOCATE` statements have been placed near the array's point of use and not as a vast initialisation block. Also some have been placed within the element loops. This is not strictly necessary but gives an indication of where they might be needed in a more complex program using more than one element type.

There are a few other problems given the current operation of some of the routines. For example `qqua4`: to define storage for `wght` and `abss` the number of quadrature points, `nqp`, must be assumed. This makes returning the value redundant. Although in this program `qqua4` is placed outside the element loops in a multi-element type application this would be moved inside the element loop.

```
PROGRAM seg3p1

!*****************************************************************

!      Copyright (C) 2003 : CLRC, Rutherford Appleton Laboratory
!      Chilton, Didcot, Oxfordshire OX11 0QX

! N.B. The working precision of the current library is held
!       in the variable wp. This must be used in all REAL
!       declarations of variables used by FELIB90.

!       The program also uses the standard FELIB90 values for
!       nin and nout.

!*****************************************************************

  USE felib90 ! Use FELIB90 all routines

  USE def3p1 ! Use standard SEG3P1 definitions

  IMPLICIT NONE

! Parameters

  REAL (wp), PARAMETER :: scale = 1.0E+10

! Local variables

  INTEGER :: bndnod, dif, dimen, dofel, dofnod, elnum, eltyp, hband, i, &
    iquad, itest, j, nele, node, nodel, nodnum, nqp, totdof, totels, &
    totnod
  REAL (wp) :: det, eta, quot, strgth, x, xi, y

! Allocatable arrays - mesh size dependent

   INTEGER, POINTER :: bnode(:), nf(:,:), eltop(:,:)
   REAL (wp), POINTER :: bval(:), rhs(:), coord(:,:), sysk(:,:)

! Intrinsic functions
```

```
      INTRINSIC abs

!     Initialisation of POINTERS to main arrays

!     NULLIFY(bnode, nf, eltop, bval, rhs, coord, sysk)

      itest = 0

!     **********************
!     *                    *
!     * Input Data Section *
!     *                    *
!     **********************

!     Input of nodal geometry - memory for coord automatic

      READ (nin,'(2I5)') totnod, dimen
      ALLOCATE (coord(totnod,dimen))
      DO i = 1, totnod
        READ (nin,'(I5,2F10.0)') node, (coord(node,j),j=1,dimen)
      END DO
      CALL prtgeo(coord)

!     Input of element topology - memory for totels automatic

      READ (nin,'(3I5)') eltyp, totels, nodel
      ALLOCATE (eltop(totels,nodel+2))
      DO i = 1, totels
        READ (nin,'(10I5)') elnum, (eltop(elnum,j+2),j=1,nodel)
        eltop(elnum,1) = eltyp
        eltop(elnum,2) = nodel
      END DO
      CALL prttop(eltop)

!     Input of permeabilities, construction of permeability matrix P
!     and source strength

      ALLOCATE (p(dimen,dimen))
      p = 0.0
      WRITE (nout,'(/A)') 'Permeabilities'
      READ (nin,'(2F10.0)') (p(i,i),i=1,dimen)
      WRITE (nout,'(2F10.5)') (p(i,i),i=1,dimen)

      WRITE (nout,'(/A)') 'Source Strength'
      READ (nin,'(F10.0)') strgth
      WRITE (nout,'(F10.5)') strgth

!     Input of number of degrees of freedom per node, input of
!     boundary conditions and construction of nodal freedom array NF

      WRITE (nout,'(/A)') 'Degrees of freedom per node (DOFNOD)'
      READ (nin,'(I5)') dofnod
      WRITE (nout,'(I5)') dofnod

!    Input boundary conidtions

      WRITE (nout,'(/A)') 'Boundary Conditions'
      READ (nin,'(I5)') bndnod
```

```
   WRITE (nout,'(I5)') bndnod

   ALLOCATE (bnode(bndnod),bval(bndnod))
   bnode = 0.0
   bval = 0.0
   DO i = 1, bndnod
     READ (nin,'(I5,F10.0)') bnode(i), bval(i)
     WRITE (nout,'(I5,F10.5)') bnode(i), bval(i)
   END DO

! Setup nodel freedom array


   ALLOCATE (nf(totnod,dofnod))
   nf = 0
   totdof = 0
   DO i = 1, totnod
     DO j = 1, dofnod
       totdof = totdof + 1
       nf(i,j) = totdof
     END DO
   END DO

!     Calculation of semi-bandwidth

   CALL bndwth(eltop,nf,hband)

!  ************************************
!  *                                  *
!  * System Stiffness Matrix Assembly *
!  *                                  *
!  ************************************


! System matrices setup and initalise : rhs, sysk

   ALLOCATE (sysk(totdof,hband),rhs(totdof))
   sysk = 0.0
   rhs = 0.0

! Setup quadrature

   nqp = 4
   ALLOCATE (wght(nqp),abss(dimen,nqp))
   CALL qqua4(wght,abss,nqp)

! Begin main element loop

   DO nele = 1, totels !Loop over all elements

     nodel = eltop(nele,2)
     dofel = dofnod*nodel

! Initial memory space for working arrays

     ALLOCATE (jac(dimen,dimen))
     ALLOCATE (gder(dimen,nodel))
     ALLOCATE (dtpd(nodel*dofnod,nodel*dofnod))
```

```
! Element matrices setup and initalise: elk, elq, scvec

     ALLOCATE (elk(dofel,dofel),elq(dofel),scvec(dofel))
     elk = 0.0
     elq = 0.0
     scvec = 0.0

     ALLOCATE (geom(dofel,dimen))
     CALL elgeom(nele,eltop,coord,geom)

!     Integration loop for element stiffness using NQP quadrature
!     points

     DO iquad = 1, nqp ! Numerical integration

!     Form linear shape function and space derivatives in the local
!     corrdinates. Transform local derivatives to global coordinate
!     system

       xi = abss(1,iquad)
       eta = abss(2,iquad)

       ALLOCATE (fun(nodel),lder(dimen,nodel))
       CALL quam4(fun,lder,xi,eta)

       x = dot_product(geom(1:nodel,1),fun)
       y = dot_product(geom(1:nodel,2),fun)

       jac = matrix_multiply(lder,geom)
       gder = matrix_multiply(matrix_inverse(jac),lder)
       dtpd = matrix_multiply(transpose(gder),matrix_multiply(p,gder))

       quot = abs(matrix_determinant(jac))*wght(iquad)
       elk = elk + dtpd*quot
       elq = elq + fun*src(x,y,strgth)*quot

       DEALLOCATE (fun,lder)

     END DO !Loop over quadrature points - iquad

!     Assembly of system stiffness matrix

     CALL direct(nele,eltop,nf,steer) ! Memory for steer automatic
     CALL assym(sysk,elk,steer)
     CALL asrhs(rhs,elq,steer)

     DEALLOCATE (elk,elq,scvec) ! Deallocate element vector & arrays
     DEALLOCATE (jac,gder,dtpd)
     DEALLOCATE (geom,geomt)

   END DO !Loop over elements - nele

!     *********************
!     *                   *
!     * Equation Solution *
!     *                   *
!     *********************

!     Modification of stiffness matrix and right-hand side to
```

```
!     implement boundary conditions

    DO i = 1, bndnod
       j = bnode(i)
       sysk(j,hband) = sysk(j,hband)*scale
       rhs(j) = sysk(j,hband)*bval(i)
     END DO

!     Solution of system matrix for the nodal values of the
!     potential

     CALL chosol(sysk,rhs) ! rhs=chosol(sysk,rhs)

     WRITE (nout,'(/A)') 'Nodal Potentials'
     CALL prtval(rhs,nf)

     STOP

   CONTAINS


!*****************************************************************
! Source function

     FUNCTION src(x,y,strgth)

       USE felib90

       IMPLICIT NONE

! Dummy arguments

       REAL (wp) :: src
       REAL (wp) :: strgth, x, y

       INTENT (IN) strgth, x, y

       src = 0.0D0
       IF ((x>1.0D0) .AND. (x<2.0D0) .AND. (y>1.0D0) .AND. (y<2.0D0)) &
         src = strgth

     END FUNCTION src

   END PROGRAM seg3p1
```