

**ERNEST ORLANDO LAWRENCE  
BERKELEY NATIONAL LABORATORY****TOUGH90: A FORTRAN90  
Implementation of TOUGH2**

George J. Moridis

**Earth Sciences Division**

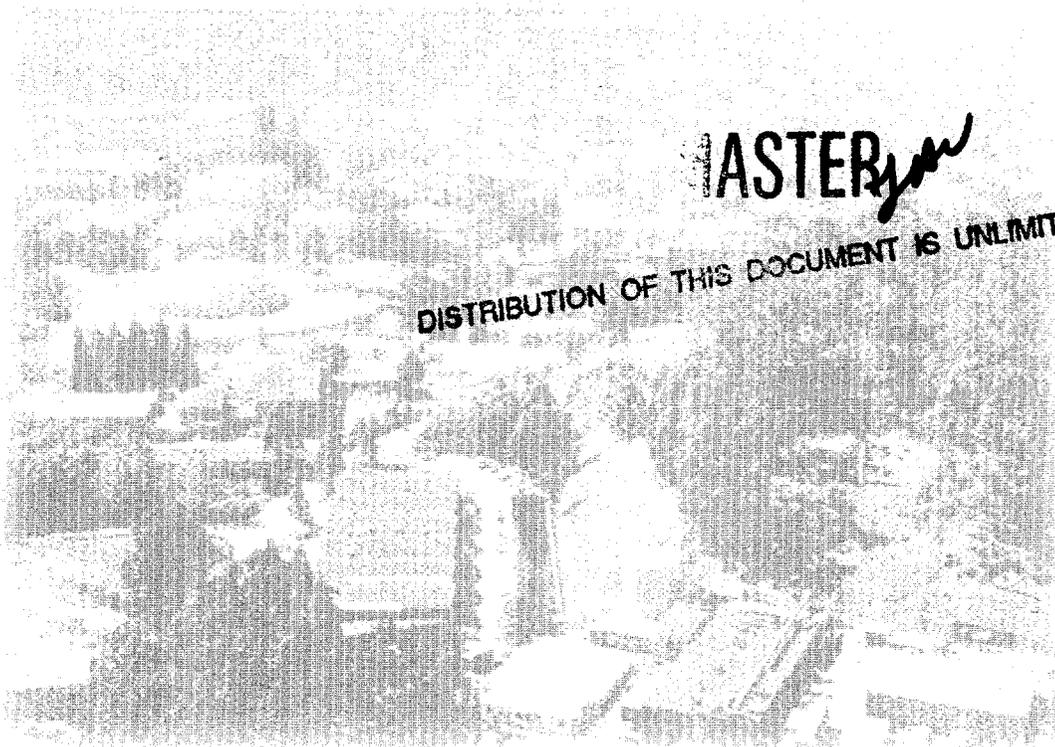
April 1998

Presented at the  
*TOUGH98 Workshop*,  
Berkeley, CA,  
May 4-6, 1998,  
and to be published in  
the Proceedings

**RECEIVED**

**JUL 23 1998**

**OSTI**



**MASTER** *you*

**DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED**

#### **DISCLAIMER**

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor The Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or The Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, or The Regents of the University of California.

Ernest Orlando Lawrence Berkeley National Laboratory  
is an equal opportunity employer.

## **DISCLAIMER**

**Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.**

**LBNL-41640**

# **TOUGH90: A FORTRAN90 IMPLEMENTATION OF TOUGH2**

*George J. Moridis*

**Earth Sciences Division  
Lawrence Berkeley National Laboratory  
University of California  
Berkeley, CA 94720**

**April 1998**

This work was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Geothermal Technologies, of the U.S. Department of Energy, under contract No. DE-AC03-76SF00098.

# TOUGH90: A FORTRAN90 Implementation of TOUGH2

*George J. Moridis*

Earth Sciences Division, Lawrence Berkeley National Laboratory  
University of California, Berkeley, CA 94720

## **Abstract**

TOUGH90 is a FORTRAN90 implementation of TOUGH2, and represents a major change in syntax and architecture over TOUGH2, while maintaining full backward compatibility with existing input data files. The main features of TOUGH90 include dynamic memory management, the use of modules, derived types, array operations, matrix manipulation, and new and very powerful intrinsic procedures. These result in a faster, more efficient and compact code, which is conceptually simpler, and significantly easier to modify and upgrade.

## **1. Introduction**

The TOUGH2 [Pruess, 1991] family of codes is a descendant of the earlier MULKOM code [Pruess, 1983], and provides multi-dimensional numerical models for simulating the coupled transport of water, vapor, non-condensable gas and heat in porous and fractured subsurface media. These models describe the processes and interactions involved in the flow of fluids in the subsurface, including the appearance and disappearance of liquid and vapor phases, boiling and condensation, multiphase flow due to pressure, gravity, and capillary forces, vapor adsorption with vapor pressure lowering, heat conduction, and heat exchange between rocks and fluids. TOUGH2 offers the flexibility of handling different fluid mixtures, e.g., water, water with tracer; water and CO<sub>2</sub>; water and air; water and air with vapor pressure lowering, water and hydrogen; water, gas and an organic liquid phase. Additional information is available in a number of reports [Pruess, 1991, 1995; Pruess *et al.*, 1996; Wu *et al.*, 1996], and on the web at URL <http://ccs.lbl.gov/TOUGH/>.

The code in all the members of the TOUGH2 family is written in FORTRAN, the most widely used scientific programming language. Each FORTRAN version is a superset of all previous versions, which therefore allows the seamless integration of code segments developed at different times since the first release of the MULKOM [Pruess, 1983] parent to TOUGH2. In that respect, the current version of TOUGH2 can be compiled and run without any problem using a FORTRAN90 compiler.

FORTRAN90 is the latest version of FORTRAN, and was released as an international standard language in 1991 [Ellis *et al.*, 1994]. It has many new features and capabilities (based on experience gained with similar concepts in other languages) which extend the functionality of the language and broaden its applicability, in addition to providing its own contributions to the development of new programming concepts. TOUGH90 is a FORTRAN90 implementation of TOUGH2, written to take advantage of the unique capabilities of the language. As such, it does not represent an incremental change over TOUGH2 but is rather a completely rewritten code with different syntax and architecture, although full backward compatibility with existing input data files is maintained.

In this paper, the most important features of TOUGH90 are discussed, i.e., the use of modules, dynamic memory management, derived types, array operations, matrix manipulation, and new intrinsic procedures. For demonstration purpose, code segments of TOUGH90 are compared to equivalent TOUGH2 code.

## **2. The Use of Modules**

Modules are new program units of FORTRAN 90, which provide a simple but highly adaptable method to compartmentalize code. They are defined as a collection of declarations and subprograms, and they make some or all of the entities declared within them accessible to the program units that invoke them.

Modules in TOUGH90 replace all uses of INCLUDE statements, COMMON blocks, and statement functions. The power of modules is in the collection of these basic concepts, i.e., shared declarations, globally accessible data, inline code expansion, etc., and generalization in the more flexible and powerful object of a module [Kerrigan, 1993]. This enables highly adaptable and easy to compartmentalize code, provides protection of data and of important source code of major data specifications or subprograms, and makes code upgrading an extremely easy task. In TOUGH90, all subprograms related to fluid properties are included in appropriate modules.

A simple TOUGH90 module is:

```
MODULE T90_Arithmetic
  IMPLICIT NONE
  SAVE
  !Integer Parameters
  INTEGER, PARAMETER :: kind_n = SELECTED_REAL_KIND(P=14) !Set the required accuracy
END MODULE T90_Arithmetic
```

The T90\_Arithmetic module demonstrates both the concept of the module and the power of the new intrinsic functions of FORTRAN90. In addition to REAL and REAL\*8 variables, FORTRAN90 introduces the REAL(KIND=n) declaration. Setting n=4, 8, 16 yields single-, double- and quad-precision arithmetic. The intrinsic function SELECTED\_REAL\_KIND(P=14) returns the KIND parameter of a real data type with decimal precision of 14 digits. Thus, inclusion of the module in the various TOUGH90 program units allows the automatic setting of the arithmetic of real variables regardless of machine and compiler specificities. In the module

```
MODULE T90_Constants
  USE T90_Arithmetic
  IMPLICIT NONE
  SAVE
  !Real Parameters
  REAL(KIND = kind_n), PARAMETER :: pi = 3.1415926536e0
  REAL(KIND = kind_n), PARAMETER :: t_0 = 2.7316e2
  REAL(KIND = kind_n), PARAMETER :: zero = 0.0
  REAL(KIND = kind_n), PARAMETER :: one = 1.0
  REAL(KIND = kind_n), PARAMETER :: large = 1.0e50
END MODULE T90_Constants
```

which defines some basic parameters used in TOUGH90, the T90\_Arithmetic module is invoked (thus making its contents accessible) by using the command USE T90\_Arithmetic. Compared to a module, the concepts of COMMON and INCLUDE are limited.

### 3. Dynamic Memory Allocation

Memory allocation in TOUGH2 involves fixed-size arrays, and recompilation is required if the array size is increased. The need to recompile is eliminated in TOUGH90 by exploiting the dynamic memory management capabilities of FORTRAN90 through the use of allocatable arrays. These afford complete control over the array size, which is specified during the program execution. Moreover, memory occupied by arrays no longer needed is released, and made available to other arrays, thus increasing the size of tractable problems. This represents an important capability for creation and handling of internal work arrays in subprograms. An example of dynamic memory allocation is demonstrated in the Water\_properties module below:

```
MODULE Water_properties
  USE T90_Arithmetic
  IMPLICIT NONE
  !
  PRIVATE
  PUBLIC :: SAT_PRESSURE
  !
  CONTAINS
  SUBROUTINE SAT_PRESSURE(temp_r,sat_p,unit_n)
    IMPLICIT NONE
    !
    INTEGER :: unit_n      ! Number of printout unit
    INTEGER :: nnn        ! size of "temp_r"
    INTEGER :: n_out      ! # of cells where "temp_r" is outside the range
    INTEGER :: i          ! counter
    INTEGER :: alloc_er1,alloc_er2,alloc_er3,dealloc_er1,dealloc_er2,dealloc_er3
  ! Integer arrays
  INTEGER, DIMENSION(:), ALLOCATABLE :: igood,sat_p ! flag for saturation pressure
  ! Real arrays
  REAL(KIND = kind_n), DIMENSION(:), INTENT(IN) :: temp_r ! temperature vector
  REAL(KIND = kind_n), DIMENSION(:) :: sat_p ! saturation pressure vector
  REAL(KIND = kind_n), DIMENSION(:), ALLOCATABLE :: tc,sc ! intermediate arrays
  ! Constant coefficients for Psat = Psat(temp_r)
```

```

REAL(KIND = kind_n), DIMENSION(5) :: a_sat = (/ -7.691234564e0, -2.608023696e1, -1.681706546e2, &
& 6.423285504e1, -1.189646225e2 /)
REAL(KIND = kind_n), DIMENSION(4) :: b_sat = (/ 4.167117320e0, 2.097506760e1, 1.0e9, 6.0e0 /)
!
nnn = SIZE(temp_r)
ALLOCATE (tc(nnn), STAT = alloc_er1) ! Allocate space for the work arrays
ALLOCATE (sc(nnn), STAT = alloc_er2)
ALLOCATE (igood_sat_p(nnn), STAT = alloc_er3)

IF(alloc_er1 /= 0 .OR. alloc_er2 /= 0 .OR. alloc_er3 /= 0) THEN
  WRITE(UNIT = unit_n, FMT = 6001)
  STOP
END IF

!
sc = 0
igood_sat_p = 0

!
WHERE(temp_r >= 1.0e0 .AND. temp_r <= 5.0e2) ! If the temperature is within range, ...
  tc = (temp_r + 2.7315e2) / 6.473e2 ! ... calculate sat_p
  sc = a_sat(1) * (1.0e0 - tc) + a_sat(2) * (1.0e0 - tc)**2 + a_sat(3) * (1.0e0 - tc)**3 &
    & a_sat(4) * (1.0e0 - tc)**4 + a_sat(5) * (1.0e0 - tc)**5
  sat_p = 2.212e7 * exp(sc / (tc * (1.0 + b_sat(1) * (1.0 - tc) + b_sat(2) * (1.0 - tc) * (1.0 - tc))) &
    & - (1.0 - tc) / (b_sat(3) * (1.0 - tc) * (1.0 - tc) + b_sat(4)))
ELSEWHERE ! Otherwise, ...
  igood_sat_p = 2 ! ... set the flag and
END WHERE

!
n_out = COUNT(igood_sat_p .EQ. 2)
IF(n_out > 0) THEN
  WRITE(UNIT = unit_n, FMT = 6002) n_out ! ... write the info out ...
  STOP
END IF

!
DEALLOCATE (tc, STAT = dealloc_er1)
DEALLOCATE (sc, STAT = dealloc_er2)
DEALLOCATE (igood_sat_p, STAT = dealloc_er3)
IF(dealloc_er1 /= 0 .OR. dealloc_er2 /= 0 .OR. dealloc_er3 /= 0) THEN
  WRITE(UNIT = unit_n, FMT = 6003)
  STOP
END IF

...
END SUBROUTINE SAT_PRESSURE
!
END MODULE Water_properties

```

In its entirety, the `Water_properties` module includes all the variables and subprograms which compute the properties of the water substance in the liquid and vapor state. The portion shown above includes only the subprogram which calculates the saturation pressure of water as a function of temperature. Regarding dynamic memory allocation, the three temporary arrays `tc`, `sc` and `igood_sat_p` are first declared as allocatable arrays (1st and 2nd underlined statements). The size of the assumed-shape array `temp_r` (containing the input temperature vector) is determined using the `SIZE` intrinsic function (3rd underlined statement), and then memory for the `tc`, `sc` and `igood_sat_p` arrays is allocated (4th through 6th underlined statements). After the computation of the saturation pressures (assumed-shape array `sat_p`), memory no longer needed is deallocated (last three underlined statements).

This module provides an opportunity to discuss some additional features of `TOUGH90`. `Water_properties` as shown above includes only a subprogram, but no variable declarations. It controls data access and protection through the use of the `PRIVATE` and `PUBLIC` statements. If these are missing, all contents in a module are public (i.e., accessible) to the program unit that invokes it. In the case of `Water_properties`, the contents of the module are all private (i.e., protected and inaccessible), with the exception of the subroutine declaration which is explicitly declared as public. Therefore, all the data in `Water_properties` (e.g., the values of the parameter arrays `a_sat` and `b_sat`) are protected and cannot be accessed and/or altered during execution.

The other very important feature of `FORTTRAN90` in the `Water_properties` module is the use of whole array operations. The statement `sc=0` involves such an operation by setting the whole `sc` array equal to 0. This is entirely equivalent to using a `DO` loop, but is simpler, less error-prone, and usually faster. The masked array assignment of the `WHERE` construct in the module is directly related to array processing. The assignment statements following it are executed for only those array elements for which the mask expression (i.e., `temp_r >= 1.0e0 .AND. temp_r <= 5.0e2`, the range of acceptable temperatures) is true. Conversely, the statements following the `ELSEWHERE` statement are executed for those elements for which the mask is false. It must be clearly pointed out that although the `WHERE` construct has a certain syntactic similarity to the block `IF` construct, the former

does not involve sequential operations. Its effect is the simultaneous assignment of all the array elements, with the mask either preventing some of the assignment taking place, or causing different ones to take place.

#### 4. Derived Types

FORTRAN77 in TOUGH2 requires that an array contain information of a single data type, leading to arrays which hold either numbers or characters, but not both. FORTRAN90 in TOUGH90 allows the creation of new data types to supplement the intrinsic types provided by the language. These derived types are powerful tools for the creation of data structures which contain elements of any data type mixed freely in any proportion. Gridblock names, connections and properties can thus be grouped in derived-type arrays, allowing easier handling as well as programming. The use of derived types is illustrated in the module T90\_grid.

```

MODULE T90_Grid
  USE T90_Arithmetic
  !
  IMPLICIT NONE
  SAVE
  !Type Declaration
  TYPE element_attributes
    CHARACTER (LEN = 5) :: name      ! element name
    INTEGER              :: mat_n    ! element rock number
    REAL(KIND = kind_n) :: vol      ! element volume
    REAL(KIND = kind_n) :: phi      ! element porosity
    REAL(KIND = kind_n) :: p        ! element pressure
    REAL(KIND = kind_n) :: tempr    ! element temperature
  END TYPE element_attributes
  !
  TYPE connection_attributes
    CHARACTER (LEN = 5) :: name_1,name_2 ! element names in a connection
    INTEGER              :: nex_1,nex_2   ! element numbers in a connection
    INTEGER              :: isox         ! specifies permeability as k = k(isox); isox=1,2,3 for x,y,z
    REAL(KIND = kind_n) :: del_1,del_2   ! element distances from interface
    REAL(KIND = kind_n) :: area         ! connection area
    REAL(KIND = kind_n) :: beta        ! angle between the g vector and the element line
  END TYPE connection_attributes
  !Dimensioning
  TYPE (element_attributes), DIMENSION(:), ALLOCATABLE :: element
  TYPE (connection_attributes), DIMENSION(:), ALLOCATABLE :: connection
  !
END MODULE T90_Grid

```

The derived types `element_attributes` and `connection_attributes` which are defined in the module include character, integer and real data types. Note that two allocatable arrays are defined: `element` of type `element_attributes` and `connection` of type `connection_attributes`, respectively. Memory for these arrays is allocated dynamically immediately after determining the number of elements (`nel`) and connections (`ncon`), i.e.,

```

ALLOCATE (element(nel), STAT = alloc_er1)
ALLOCATE (connection(ncon), STAT = alloc_er2)

```

Array operations can be used to assign values to the various components of the derived types. For example, an initial temperature distribution of 20 °C is assigned by the statement

```

element%tempr = 20.

```

Thus the set of TOUGH2 statements

```

PARAMETER (MNEL=800, MNCON=2400, MNEQ=3, MNK=2, MNPH=2, MNB=6)
PARAMETER (MNOGN=50, MGTAB=2000)
...
COMMON /MN/ NEL, NCON, NOGN, NK, NEQ, NPH, NB, NK1, NEQ1, NBK, NSEC, NFLUX
...
COMMON /E1/ ELEM (MNEL)
COMMON /E2/ MATX (MNEL)
COMMON /E3/ EVOL (MNEL)
COMMON /E4/ PHI (MNEL)
COMMON /E5/ P (MNEL)
COMMON /E6/ T (MNEL)
...
COMMON /C1/ NEX1 (MNCON)
COMMON /C2/ NEX2 (MNCON)
COMMON /C3/ DEL1 (MNCON)
COMMON /C4/ DEL2 (MNCON)
COMMON /C5/ AREA (MNCON)
COMMON /C6/ BETA (MNCON)
COMMON /C7/ ISOX (MNCON)
COMMON /C9/ ELEM1 (MNCON)
COMMON /C10/ ELEM2 (MNCON)

```

is replaced by the T90\_grid module and the following T90\_Dimensions module

```

MODULE T90_Dimensions
  IMPLICIT NONE
  SAVE
  !Integer Parameters
  INTEGER :: nel      ! # of elements/gridblocks
  INTEGER :: nela     ! # of active elements
  INTEGER :: ncon     ! # of connections
  INTEGER :: neq      ! # of equations per element
  INTEGER :: nph      ! # of phases
  INTEGER :: nk       ! # of components
  INTEGER :: nogrn    ! # of sources and/or sinks
  INTEGER :: neq_tot  ! # of total equations ( = order of the Jacobian)
  INTEGER :: n_zero   ! # of non-zero elements of the Jacobian
END MODULE T90_Dimensions

```

The information contained therein is accessible to all the program that invoke the modules. Note that there is no explicit declaration of the values of the variables in T90\_Dimensions. This is because the array sizes in TOUGH90 are allocated dynamically. The use of the derived types allows a simpler and more intuitive handling of the grid-related properties.

### 5. Array Operations, Matrix Manipulation and Expanded Set of Intrinsic Functions

An indication of the power and convenience of whole array operations has already been shown in Section 3. In this section we discuss some additional features of array operations and related intrinsic procedures. This is illustrated by an example involving the following code segment from the DBCG subroutine of the T2CG1 module [Moridis and Pruess, 1995]:

```

      IF(ITER .EQ. 1) THEN
        DO 18 I = 1,N
          P(I) = Z(I)
          PP(I) = ZZ(I)
18      CONTINUE
        ELSE
          BK = BKNUM/BKDEN
          DO 20 I = 1, N
            P(I) = Z(I) + BK*P(I)
            PP(I) = ZZ(I) + BK*PP(I)
20      CONTINUE
        ENDIF
        BKDEN = BKNUM
C
        CALL MATVEC(N, P, Z, NELT, IA, JA, A, ISYM)

        DDOT = 0.D0
        DO 25 I = 1,N
          DDOT = DDOT + PP(I)*Z(I)
25      CONTINUE
        AKDEN = DDOT

```

In the same routine of TOUGH90 this segment is replaced by the much easier and more compact

```

IF(ITER .EQ. 1) THEN
  P = Z
  PP = ZZ
ELSE
  BK = BKNUM/BKDEN
  P = Z + bk*p
  PP = ZZ + bk*pp
ENDIF
BKDEN = BKNUM
!
CALL MATVEC(N, P, Z, NELT, IA, JA, A, ISYM)
!
AKDEN = DOT_PRODUCT(pp,z)

```

It is evident that the code in TOUGH90 is more intuitive and much easier to develop and follow, as the code is quite similar to the descriptive pseudocode. The above example also illustrates the use of the DOT\_PRODUCT array intrinsic function, which returns the dot product of the two vectors in its argument list and can be significantly faster than the equivalent DO loop, in addition to being simpler and less error-prone. Similarly, the determination of the maximum residual and its location in the MULTI subroutine changes from

```

C
C
C-----TEST FOR CONVERGENCE-----
C
      RERM=0.D0
      DO10 N=1,NELA
        NLOC=(N-1)*NEQ

```

```

DO10 K=1,NEQ
  NLM=NLOC+K
  DOA=ABS(DOLD(NLM))
  IF(DOA.LT.RE2) RER=R(NLM)/RE2
  IF(DOA.GE.RE2) RER=R(NLM)/DOLD(NLM)
  IF(ABS(RER).LE.RERM) GOTO 10
  RERM=ABS(RER)
  NER=N
  KER=K
10 CONTINUE

```

to the following array-based syntax in TOUGH90.

```

WHERE(abs(dold).LT.RE2)
  rerm = MAXVAL(abs(r/RE2))
  merm = MAXLOC((abs(r/RE2)))
ELSEWHERE
  rerm = MAXVAL(abs(r/dold))
  merm = MAXLOC((abs(r/dold)))
END WHERE
nnn = SIZE(r,DIM=2)
ner = merm(1)
ker = merm(nnn)

```

The above segment uses the WHERE construct, and obtains the maximum values and their locations using the MAXVAL and MAXLOC intrinsic procedures. The array merm contains the indices of the location of the maximum residual, from which the corresponding element and equation numbers are determined.

The matrix manipulation capabilities of FORTRAN90 greatly reduce the coding complexity in the handling of linear algebra. All the matrix-vector multiplication routines (e.g., DSMV, DSMTV, DSLUI, DSLUTI) in the T2CG1 package [Moridis and Pruess, 1995] have been replaced by the very efficient (in terms of speed and storage) procedure

```
MATMUL(matrix_A,matrix_B)
```

where matrix\_A and matrix\_B are arrays of rank 1 or 2. The replacement of whole TOUGH2 subroutines by the simple and powerful new intrinsic functions available to the FORTRAN90 is quite common in TOUGH90. For example, the FLOP subroutine in TOUGH2 determines the number of significant decimal digits and the default increment for the calculation of derivatives based on machine-specific values.

```

SUBROUTINE FLOP
...
  A = SQRT(.99D0)
  B = A
  DO 1 N=1,260
    B = B/2.D0
    C = A+B
    D = C-A
    IF(D.EQ.0.D0) GO TO 2
  1 CONTINUE
C
  2 B2=B*2.D0
  N10=-INT(LOG10(B2))
  DF=SQRT(B2)
...
END

```

FLOP is replaced in TOUGH90 by the statements

```

n10 = PRECISION(one)
df = SQRT(EPSILON(one))

```

which use the intrinsic FORTRAN90 functions PRECISION (determining the decimal precision of a variable of the same type as one) and EPSILON (returning the smallest possible number  $\epsilon$  such that  $1.0+\epsilon$  is numerically different than 1). Similarly, the subroutine SECONDS (which provides timing information in TOUGH2 by using system- and machine-specific subroutines) is replaced in TOUGH90 by::

```
CALL SYSTEM_CLOCK(COUNT=itime, COUNT_RATE=irate)
```

This is a FORTRAN90 generic statement, and is compiler- and machine-independent, thus eliminating the need for adjustments which TOUGH2 requires when moving between computing platforms and compilers.

## 6. Summary

TOUGH90 is a FORTRAN90 implementation of TOUGH2, and represents a major change in syntax and architecture over TOUGH2, while maintaining full backward compatibility with existing input data files. The main features of TOUGH90 include dynamic memory management, the use of modules, derived types, array operations, matrix manipulation, and new and very powerful intrinsic procedures. These result in a faster, more efficient and compact code, which is conceptually simpler, and significantly easier to modify and upgrade.

Modules in TOUGH90 replace all uses of INCLUDE statements, COMMON blocks, and statement functions. Additionally, all subprograms related to fluid properties are included in the appropriate modules. The power of modules is in the collection of the basic concepts represented in the standard FORTRAN features, i.e., shared declarations, globally accessible data, inline code expansion, etc., and generalization in the more flexible and powerful object of a module. This enables highly adaptable and easy to compartmentalize code, provides protection of data and of important source code of major data specifications or subprograms, and makes code upgrading an extremely easy task.

The need to recompile TOUGH90 with an increasing problem size is eliminated by exploiting the dynamic memory management capabilities of FORTRAN90. Allocatable arrays allow the total size of the arrays to be specified through the input files during the program execution. Moreover, memory occupied by arrays no longer needed is released, and made available to other arrays. This represents an important capability for creation and handling of internal work arrays in subprograms, and increases the size of tractable problems.

Derived types are powerful tools for the creation of data structures which contain elements of any data type mixed freely in any proportion. Gridblock names, connections and properties can thus be grouped in derived-type arrays, allowing easier handling as well as programming. TOUGH90 uses the array operations of FORTRAN90 for a code that is more transparent, faster and easier to program. Extensive use of array and matrix manipulation operations is made in the linear algebra of the solvers in TOUGH90. This leads to a sizable reduction in the memory requirements and improvements in the execution speed.

## 7. Acknowledgments

This work was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Geothermal Technologies, of the U.S. Department of Energy, under contract No. DE-AC03-76SF00098. Drs. J. Apps and C. Oldenburg are thanked for their helpful review comments.

## 8. References

- Ellis, T.M.R., I.R. Philips and T.M. Lahey, *FORTRAN90 Programming*, Addison-Wesley, New York, NY, 1994.
- Kerrigan, J.F., *Migrating to FORTRAN90*, O'Reilly and Associates, Sebastopol, CA, 1993.
- Moridis, G.J. and K. Pruess, Flow and transport simulations using T2CG1, a package of preconditioned conjugate gradient solvers for the TOUGH2 family of codes, *Rep. LBL-36235*, Lawrence Berkeley National Laboratory, Berkeley, CA, 1995.
- Pruess, K., Development of the general purpose simulator MULKOM, *Rep. LBL-15500*, Lawrence Berkeley Laboratory, Berkeley, CA, 1983.
- Pruess, K., TOUGH2 - A general-purpose numerical simulator for multiphase fluid and heat flow, *Rep. LBL-29400*, Lawrence Berkeley Laboratory, Berkeley, CA, 1991.
- Pruess, K (ed), Proceedings of the TOUGH2 Workshop '95, *Rep. LBL-37200*, Lawrence Berkeley Laboratory, Berkeley, CA, 1995.
- Pruess, K., A. Simmons, Y.S. Wu and G. Moridis, TOUGH2 software qualification, *Rep. LBL-38383*, Lawrence Berkeley National Laboratory, Berkeley, CA, 1996.
- Wu, Y.S., C.F. Ahlers, P. Fraser, A. Simmons and K. Pruess, Software qualification of selected TOUGH2 modules, *Rep. LBNL-39490*, Lawrence Berkeley Laboratory, Berkeley, CA, 1996.