使ってみませんか？ー便利なソフト利用法ー

Fortran コンパイラ「g95/gfortran」

核融合科学研究所
稲 垣　　滋

1．Fortran は実験データ解析に使えるか？

Fortran は定型のバッチ処理向き，というイメージが強く，ダイナミックな要素の多い実験データ解析は不可能と思われがちです．確かに FORTRAN77 では不可能かもしれません．例えばファイルを開くまでデータの大きさがわからない，しかもその大きさはファイルごとに変わり，上限も未定であるような場合はどうやってコーディングすればよいでしょうか？しかし Fortran は FORTRAN77 から Fortran90/95（最近 Fortran2003 の規格が発表されました）へと大幅な進化をとげ，最も現代的な言語へと成長しました．ここでは Fortran95の新機能を紹介して，実験データ解析にも十分使えることを示したいと思います．

2．Fortran95 での新機能

私がよく使う Fortran95 の新機能を以下に示します．

1．　ダイナミックアロケーション
2．　配列演算
3．　文字列操作関数
4．　モジュール，内部関数
5．　構造体，ポインタ

このなかで，実験データ解析に欠かせないのが 1 のダイナミックアロケーションです．FORTRAN77 では，配列のサイズは作業用の配列を含めてすべてコンパイル時に決まっていなければなりませんが，Fortran90 からは配列の大きさを実行時に変えることができるようになりました．この機能を用いることで，配列の大きさをファイルやキーボード入力などで指定することができます．Fortran90 の動的配列は，C の malloc 系関数とは異なり多次元配列が確保でき，確保後は FORTRAN77 の静的配列と全く同様に扱えます（その他の Fortran と C/C ++ との違いは Appendix 1 参照）．この動的配列の処理速度は，多くのコンパイラで静的配列と同様です．実際，この動的配列を用いるだけで今までの FORTRAN77 での煩わしさがかなり解消され，コードの可読性が良くなります（配列のサイズを変えるためだけに再コンパイルする必要がない．下位のルーチンの作業配列を上位のルーチンから渡す必要がないなど）．プログラム例 1 は非常に単純な例です．自作と明記されてない関数は Fortran95 の組み込み関数です．最初の USE ....というのはモジュール（後述）の呼び出しです．NUM_KINDS というモジュール（自作）に変数の精度が定義してあり，今回は作業精度（wp）に倍精度（dp）を用い，整数は 4 バイト（i4b）としています．MYLIB にはいくつかの自作ルーチンが登録されており，そのなかの GET_SIZE_FROM と GET_DATA_FROM を使うことを宣言しています．コーディングスタイルに関しては http://www.mri-jma.go.jp/Project/mrinpd/coderule.html や Numerical Rec-

プログラム例 1

```
PROGRAM PLOTDATA
!
!   コメントは！で始まります.
!   1行は132文字までです.
!   行頭を 6 文字空ける必要はありません.
!   7 文字以上の変数名が使えます.
!   小文字が使えます.
!
  USE NUM_KINDS, ONLY : wp=>dp, i4b              ! 自作
  USE MYLIB,     ONLY : GET_SIZE, GET_DATA       ! 自作
!
  IMPLICIT NONE  ! 行の途中からもコメントが書けます.
! IMPLICIT NONE は必ずつけましょう
!
  INTEGER(i4b)           :: ntime, nx
  REAL(wp), ALLOCATABLE :: time(:), x(:), Temp(:,:)
!
```


J. Plasma Fusion Res. Vol.81, No.5 (2005) 398-409

```
! Fortran は変数，関数の大文字，小文字を区別しないので Temp，temp，TEMP は
! 同義です．このため temporal のつもりで REAL(wp) :: temp という変数を新たに
! 加えようとするとエラーが起こります．上記の Temp が温度としての意味ならば
! temperature(:,:) と定義したほうが良いかもしれません．
!
  CHARACTER(LEN=64) :: fname
! CHARACTER*64 や filename*64 の書き方はやめましょう．
!
その他の宣言や前処理
!
  CALL GET_SIZE(fname, ntime, nx)          ！自作
  ALLOCATE(Temp(ntime, nx), time(ntime), x(nx))
!
  CALL GET_DATA(fname, time, x, Temp)      ！  自作
!
データ処理，表示
!
  DEALLOCATE(time, x, Temp)
!
END PROGRAM PLOTDATA
```

ipe in Fortran90 を参考にしています．Fortran90 の文法や FORTRAN77 との違いは http://www.ip.media.kyoto-u.ac.jp/htomita/が参考になります（Fortran90 プログラミングという書名で培風館より出版されています）．この例で重要なのは，ALLOCATABLE 宣言，ALLOCATE 文と DEALLOCATE 文です．ALLOCATABLE で宣言された配列は割り付け配列と呼ばれ，ALLOCATE で確保し，DEALLOCATE で開放することができます．Fortran95 では，使われなくなった割り付け配列は自動的に開放されるので，DEALLOCATE はしなくても良い場合がありますが，DEALLOCATE し忘れたときの保険程度に考え，基本的には ALLOCATE したものは DEALLOCATE したほうが良いと思います．上の例では配列の構築に GET_SIZE_FROM と GET_DATA_FROM と 2 回手続きを行っています．これは割り付け配列は宣言したルーチンで ALLOCATE しなければならないからです．もし 1 回ですませたい場合は，モジュールでグローバル変数として管理するか，ポインタを使います．割り付け配列は，主にメインルーチンでデータ用の配列を確保するのに用いられます．サブルーチンで作業用配列が動的に必要な場合は，もっと簡単な自動配列を用いることができます．プログラム例 2 に自動配列の例を示します．この例では coef と yfit が自動配列です．同じ

プログラム例 2

```
SUBROUTINE PLOT_POLYFIT(x, y, ndeg)
!
  USE NUM_KINDS, ONLY : wp=>dp, i4b
  USE LIBFIT,    ONLY : POLYFIT    ！自作
!
  IMPLICIT NONE
!
  INTEGER(i4b), INTENT(IN) :: ndeg
  REAL(wp),   INTENT(IN) :: x(:), y(:) ！形状引継ぎ配列
!   実際にこのルーチンを呼び出すときは形状引継ぎ配列を使用しているため明示的
!   インターフェイスが必要になる場合があります．
!
  REAL(wp) :: coef(0:ndeg), yfit(SIZE(y))   ！自動配列
!
  CALL POLYFIT(x, y, coef, yfit)              ！自作
!
データ表示
!
  RETURN
END SUBROUTINE PLOT_POLYFIT
```

ことを割り付け配列でも実現できますが自動配列の方が簡単（ALLOCATE，DEALLOCATE の必要がない）なうえ，配列の大きさによってはヒープ領域でなくスタック領域を使用します．上の例は FORTRAN77 とたいして違わないように見えますが，FORTRAN77 では ndeg と SIZE(y)がコンパイル時に決まっていない場合はエラーになります．

ここまで Fortran の動的配列対応に触れてきましたが，配列に密接に関連した機能である配列演算について説明したいと思います．私が Fortran95 を使う最大の理由は，この配列演算にあります．プログラム例 3 にいくつかの例を示します．これに加えて Fortran95 では豊富な配列処理関数 SIZE，SUM，MINVAL，MINLOC，MATMUL，MERGE，PACK，TRANSPOSE などが用意され，コーディングがかなり簡単になりました．配列は FORTRAN で発明されただけに，配列操作は Fortran に一日の長があるように感じます．このように配列演算を用いると，Do-Loop がいらなくなりコードの行数が減るのに加えて，紙に書いたアルゴリズムをほぼそのままコーディングできるようになり，バグが混入する危険性を少なくすることができます（後述の並列化うんぬんより実はこの点が一番重要だと思います）．

配列演算は並列処理する仕様になっており，コンパイラや CPU がベクトル演算や並列処理をサポートしている場合はパフォーマンスの向上が期待できます．ここで並列処理に関して注意しなければならない点があります．

```
indx(1:10)=(/ 1,2,3,4,5,6,7,8,9,10 /)
```

のとき

```
indx(2:9) = indx(3:10) − indx(1:8)
```

と

```
DO i = 2, 9
  indx(i) = indx(i+1) − indx(i − 1)
ENDDO
```

では結果が異なります．配列演算では `indx = (/ 1,2,2,2,2,2,2,2,2,10/)` であり，Do-Loop では `indx`

プログラム例 3

○配列全体に 0 を代入

```
ex1: DO i = 1, SIZE(a)
       a(i) = 0.0_wp
     ENDDO
ex2: a = 0.0_wp
ex3: a(:) = 0.0_wp
ex4: a(1:SIZE(a)) = 0.0_wp
```

上記 4 例は全く同じ結果となります．ex2-4 までが配列演算（代入）です．
書き方のスタイルは趣味の問題ですが，ex2 では a がスカラーか配列か（人間が）
分かりにくいので，私は ex4 のような書き方をします．

○配列同士の足し算

```
a(1:n) = b(1:n) + 3.0_wp*c(1:n)
```

○配列要素同士の掛け算

```
a(1:n) = b(1:n)*c(1:n)
```

○配列要素の基本数学関数演算

```
a(1:n, 1:m) = SIN(a(1:n, 1:m)) + EXP(2.0_wp/b(1:n, 1:m))
```

○偶数番目を 0 にする

```
a(2:n:2) = 0.0_wp
```

○順番をひっくり返す

```
a(1:n) = a(n:1:-1)
```

○ある配列の 1,5,12,56,87 番目の要素を取り出したい

```
indx(5) = (/1,5,12,56,87/)
a(1:5) = b(indx)
```

○絶対値が 0 より充分大きい時だけ要素の逆数を求める

```
WHERE(ABS(a) > SQRT(EPSILON(1._wp))) a = 1.0_wp/a
```

○ベクトルの内積

```
DOT_PRODUCT(a, b)
```

これは `SQRT(SUM(a*b))` と同じ結果となります．

= (/1,2,2,3,3,4,4,5,5,10/)となります．配列演算は“一括”で処理されるのに対して，Do-Loop は“逐次”計算が行われるので，i－1とi+1番目の要素を使って再定義されたi番目の要素が，次のステップのi－1で再び使われます．配列演算は微分方程式を解くような場合に特に有効です．

実験データを解析する場合，数値を文字にしたり，コンマの位置を見つけたり，単位に括弧をつけたりなどの文字列処理が必要になることが多いです．Fortran は，文字処理は苦手と思われがちですが，Fortran95 の文字列処理関数は使いやすくパワフルです．プログラム例 4 に文字処理関数の例を示します．この例の機能を用いれば，様々なフォーマットのデータファイルに対応できます（しかし Fortran を用いるのであれば，データファイルのヘッダー

プログラム例 4

```
CHARACTER(LEN=28) :: line
CHARACTER(LEN=12) :: key, strval
INTEGER(i4b)      :: max_len, ilen, ipos, ival
!
  line    = 'datasize = 128'
  max_len = LEN(line)               ! max_len = 28
  ilen    = LEN_TRIM(line)          ! ilen = 14
  ipos    = INDEX(line, '=')        ! ipos = 10
  key     = line(1:ipos-1)          ! key = 'datasize  '
  strval  = line(ipos+1:ilen)       ! strval = line(ipos+1:)でも良い
  IF (.NOT. ISDIGIT(strval)) STOP
! strval に数字空白以外の文字が含まれていたら stop
  READ(strval, *) ival              ! ival = 128
  ival = ival + 1
  WRITE(strval, *) ival             ! strval = '  129  '
  line = TRIM(key)//' = '//&
        &ADJUSTL(strval)            ! line = 'datasize = 129    '
!   行は＆で連結できます．
  line = TOUPPER(line)              ! line = 'DATASIZE = 129    '
!
!=====================================================================
!
  FUNCTION ISDIGIT(str) RESULT(stat)
!
!   入力文字 str が数字空白のみを含めば．TRUE.を，それ以外は．FALSE.を返す．
!
    USE NUM_KINDS, ONLY : i4b
!
    IMPLICIT NONE
!
    CHARACTER(LEN=*), INTENT(IN) :: str
    LOGICAL :: stat
!
    stat = .FALSE.
    IF (LEN_TRIM(str) == 0) RETURN
    IF (VERIFY(str, " +-1234567890") == 0) stat = .TRUE.
  RETURN
END FUNCTION ISDIGIT
!
!=====================================================================
!
FUNCTION TOUPPER（istr）  RESULT（ostr）
!
!    入力文字を大文字にします．TRIM した長さの文字を返すため
!    明示的インターフェイスが必要になる場合があります．
!
    USE NUM_KINDS, ONLY : i4b
```

```
!
    IMPLICIT NONE
!
    CHARACTER(LEN=*), INTENT(IN) :: istr
    CHARACTER(LEN=LEN_TRIM(istr)) :: ostr
!
    INTEGER(i4b), PARAMETER :: ascii_a = ICHAR('a') !  = 97
    INTEGER(i4b), PARAMETER :: ascii_z = ICHAR('z') !  = 127
    INTEGER(i4b)            :: i, ilen, ascii
!
    ostr = ''
    ilen = LEN_TRIM(istr)
    IF (ilen == 0) RETURN
!
    DO i = 1, ilen
      ascii = ICHAR(istr(i:i))
      IF (ascii_a <= ascii .AND. ascii <= ascii_z) THEN
        ascii = ascii - 32
      ENDIF
      ostr(i:i) = CHAR(ascii_code)
    ENDDO
    RETURN
END FUNCTION TOUPPER
```

や設定ファイルにはNAMELISTを使う方が便利です).

Fortran95の新機能であるダイナミックアロケーション，配列演算，文字列処理関数を用いれば，コードを書くという点では最近のインタプリタ言語並に簡単になったと思います．まだ変数宣言の手間などに違いがありますが，変数宣言と変数の型チェックは“見つけることが非常に困難なバグ”[A3]の混入を防ぐために培われたノウハウなので，暗黙の型宣言（IMPLICIT REAL*8 (a-h, o-z)など）を使わずINPLICIT NONEを使い，がんばって変数宣言しましょう．

3．Fortran95コンパイラ

ここまでFortran95を薦めてきましたが，Fortranコンパイラは高価です．しかしこの春gcc4.0の登場と共にg77に変わってgfortranがFortran95をサポートします（http://gcc.gnu.org/fortran/index.html）．またgccではありませんがスタンドアローンのFortran95コンパイラとしてg95（http://www.g95.org）があります．（実はg95とgfortranは元が同じで分岐したものです.）g95の方がFortran95標準への対応が少し進んでいるようで，私はLinux版のg95を愛用しております．g95はWindows版alpha版Mac版もあるのでどんなプラットフォームでも試してみることができます．g95のホームページにはg95でコンパイル可能な様々なライブラリやコードへのリンク（http://www.g95.org/g95_status.html）がありますが，その中でトカマクの乱流コードGYROへのリンクがあるのは大変興味深いです．日本のOpen Source Projectに対する寄与は海外に比べてまだまだ少ないと感じます．いつも恩恵にあずかるばかりではなく例えばg95を使ったコードやユーティリティを開発して公開するなどの恩返しも必要なのではと思います．

4．ライブラリ

4.1 グラフィックライブラリー

実験解析では実験データを可視化することが非常に重要です．Fortranには描画機能がないためライブラリを使うことになります．ここで紹介するライブラリはFortranから呼び出して使うものです．

・OpenGLのFortran90インターフェイス：http://math.nist.gov/f90gl
・広く普及しているplplot: http://plplot.sourceforge.net
・美しく高機能なDISLIN: http://www.mps.mpg.de/dislin

私はg95と相性が良く，Windows版もあるDISLINを使っています．Fig. 1にDISLIN+g95で作ったアプリケーションの一例を示します．画面だけでなくpsやpngフォーマットでも出力できるので論文やプレゼンテーションにも使えそうです(当然ですがps出力の場合，図のような枠やボタンは含まれずグラフの部分だけになります)．通常の2次元グラフに加えてCONTOURやSURFACEプロットもできます．DISLINにはPerl版などもありますが，Fortran版はインタープリタ版と同様の手軽さで利用できます．

4.2 数値計算ライブラリ

Fortran90/95では，ほとんどのFORTRAN77ライブラリが使用可能です．しかしそのためFortran95ネイティブのライブラリはまだまだ少ないのが現状です．フォーマットや，古いFORTRAN77の命令を新しいものに置き換えたりしたものは多くありますが，Fortran95の並列処理を活かしたり，最新のアルゴリズムを用いたものはいまのところ商用ライブラリに限られています．ここでは可搬性が高くほとんどのコンパイラでコンパイル可能と思われるフ

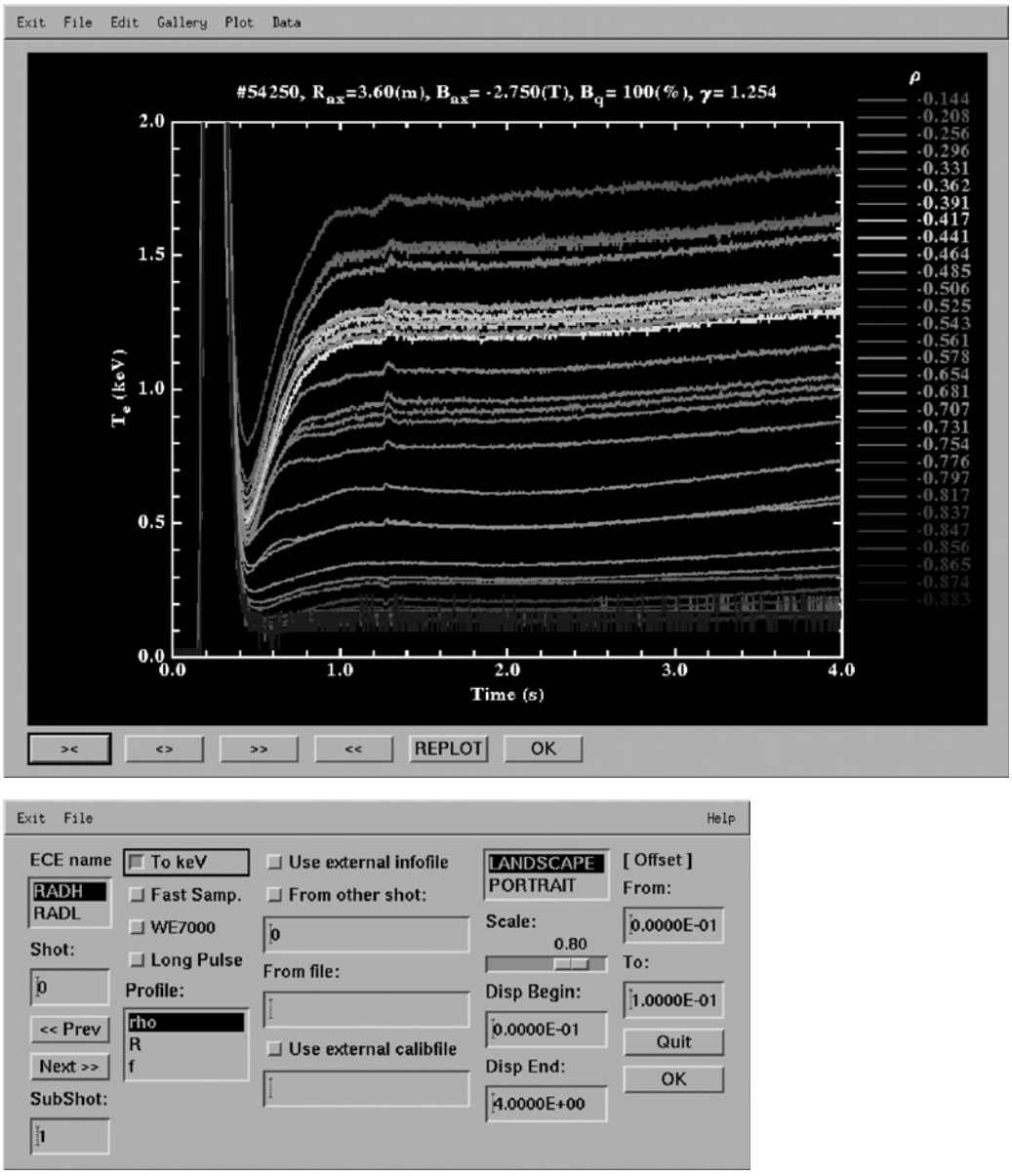

Fig. 1　DISLIN を用いた実験データビューアの例（Imagemagic というアプリケーションで撮ったスクリーンショット）．DISLIN を用いれば通常のグラフのみならず，WIDGET や MENU を簡単に作ることができます．

リーフォーマット書式のものを紹介します．

1：http://users.bigpond.net.au/amiller/非常に多くの FORTRAN77 ライブラリが Fortran90/95 のフリーフォーマットに書き換わっています．

2：http://www.pdas.com/fmm.htm 基本的な行列演算，スプライン補間，非線形方程式ソルバ，数値積分，常微分方程式ソルバのシンプルなパッケージ．ちょっと Fortran95 を使ってみたい方にお手軽なライブラリとして適していると思います．その他のライブラリに関する情報は

http://www.cts.com.au/tomasz.html
http://www.netlib.org/
http://www.personal.psu.edu/faculty/h/d/hdk/fortran.html

などを参考にしてください．

5．モジュール

今まで紹介してきた新機能のみでも十分なのですが，実は Fortran95 の真髄はモジュールにあります．モジュールとは手続きとデータをパッケージ化したものであり，SUBROUTINE や FUNCTION よりも大きなプログラム単位になります．今までのコーディングでは，あるまとまった処理をするときどのように SUBROUTINE に分けるか，という設計を行いますが，Fortran95 ではまずどのような機能を持つモジュールが必要か，という設計を行います．このモジュールの登場によりコードの保守，管理，再利用が大幅に楽になりました．モジュールの機能のなかで最も単純で重要なものは，グローバル変数の管理です．FORTRAN77 のコードで可読性を悪化させる最大の犯人は COMMON 文（と EQUIVALENCE 文）です．これが IN-

CLUDE 文と一緒に使われるとソースを読むのがいやになってきます．反論があるかもしれませんが，Fortran95 では COMMON 文と INCLUDE 文はできるだけ避け，モジュールと USE を使ってコーディングするようにしましょう[A2]．以下にモジュールを使った例をいくつか示します．最初にプログラム例 5 に既出の NUM_KINDS の中身を示します．FORTRAN77 では，倍精度実数を宣言するのに REAL（8）や REAL＊8 や DOUBLE PRECISION を使ったりしますが，REAL（2）と指定するコンパイラもあります．また整数も INTEGER が 4 バイトだったり，8 バイトだったりします．そのコンパイラ間の違いを吸収するために Fortran95 では KIND，SELECTED_INT_KIND 関数が用意されています．NUM_KINDSのようなモジュールを用意しておけば，コンパイラに依存せずに倍精度や 4 バイト整数が指定できます．今後64bitCPU が一般化され，整数がデフォルト 8 バイトになったりすると，今まで単に INTEGER だけで宣言してきたプログラムが壊れる可能性もあり，このようなモジュールの有用性は増します．また，このようなコンパイラや CPU による違いを吸収する機能として Fortran95 には EPSILON, TINY, RADIX, MAXEXPONENT などの数値モデル問い合わせ関数が用意されています．これらを用いれば，より可搬性の高いコードを作れます（LAPACK95（http://www.netlib.org/lapack95）でもそのようにコーディングされています）．COMMON 文の代替としてのモジュール変数利用の例をプログラム例 6 に示します．この例ではモジュールのもう一つの重要な機能であるスコープの限定の例にもなっています．モジュール変数やモジュール手続きは USE 文によってスコープが限定できるため，名前空間の汚染が防げます（名前空間の汚染とは例えばライブラリ A に GET_DATA という関数があり，ライブラリ B にも GET_DATA という関数があった場合，コンパイラはライブラリ A とライブラリ B を同時にはリンクできない）．このため自作した関数やサブルーチンはすべてモジュール化することを薦めます（明示的イン

プログラム例 5

```
MODULE NUM_KINDS
  IMPLICIT NONE
!
  PRIVATE
!   モジュール変数はデフォルトで非公開とします．PRIVATE 宣言が無い場合や，
! PUBLIC 宣言があればモジュール変数は公開です．
!
  INTEGER, PARAMETER :: i4b = SELECTED_INT_KIND(9)
  INTEGER, PARAMETER :: i2b = SELECTED_INT_KIND(4)
  INTEGER, PARAMETER :: i1b = SELECTED_INT_KIND(2)
  INTEGER, PARAMETER :: sp  = KIND(1.0)
  INTEGER, PARAMETER :: dp  = KIND(1.0D0)
!
  PUBLIC :: i4b, i2b, i1b, dp, sp
! i4b, i2b, i1b, dp, sp を公開します．
END MODULE NUM_KINDS
```

プログラム例 6

```
MODULE MYGLOBAL
  USE NUM_KINDS, ONLY : wp=>dp, i4b
!
  IMPLICIT NONE
!
  PRIVATE
!
  INTEGER(i4b), PARAMETER :: max_size = 1024*1024
  INTEGER(i4b), PARAMETER :: max_num  = 32
!
  INTEGER(i4b) :: data_size   = 0
  INTEGER(i4b) :: channel_num = 0
  INTEGER(i4b) :: rawdata(1:max_size, 1:max_num) = 0
  REAL(wp)     :: time(1:max_size) = 0.0_wp
!
```

```
  PUBLIC :: data_size, channel_num
  PUBLIC :: rawdata, time
!
! max_size と max_num は公開(PUBLIC)されていないので他のサブルーチンからは
!   使えません.
!
END MODULE MYGLOBAL
!
!=====================================================================
!
SUBROUTINE READ_RAWDATA(filename)
  USE NUM_KINDS, ONLY : wp=>dp, i4b
  USE MYGLOBAL, ONLY : datasize, channel_num, rawdata  ! 自作
  USE MYFILEIO, ONLY : NEWUNIT                         ! 自作
!
  IMPLICIT NONE
!
  CHARACTER(LEN=*), INTENT(IN) :: filename
!
  INTEGER(i4b) :: u
  REAL(wp)     :: time
! USE 文で time は宣言されていないので，この time は MYGLOBAL 内の time ではない.
!
  u = NEWUNIT()
  OPEN(u, TRIM(filename), STATUS='OLD')
ファイル操作
!
  READ(u,*) time
  READ(u,*) datasize, channel_num
  DO i = 1, data_size
    READ(u,*) rawdata(i,1:channel_num)
  ENDDO
!
ファイル操作
  CLOSE(u)
!
  RETURN
END SUBROUTINE READ_RAWDATA
!
!---------------------------------------------------------------------
!
FUNCTION NEWUNIT() RESULT(u)
  USE NUM_KINDS, ONLY : wp=>dp, i4b
!
  IMPLICIT NONE
!
  INTEGER(i4b) :: u
!
  LOGICAL :: stat
!
  u = 10
  DO                                ! 無限ループ
    INQUIRE(UNIT=u, OPENED=stat)
    IF (.NOT. stat) EXIT            ! Do Loop は EXIT で抜けます
    u = u + 1
  ENDDO
  RETURN
FUNCTION NEWUNIT
```

ターフェイスも不要になります)．以上はモジュールの最も初歩的な使い方です．モジュールの機能は非常に大きく，演算子のオーバーロードなどの機能もあります．モジュールと，構造体，ポインタを駆使することでC++のクラスに近い機能を実現することができます．Fortran95でのオブジェクト指向プログラミングはまた機会があったら紹介したいと思います．

6．PCで数値シミュレーション

Fortranの本来の使命である処理速度に関しての進展について議論してみたいと思います．処理速度は言語によって決まるものではなく，アーキテクチャ，アルゴリズム，コーディング，コンパイラの性能などによって決まりますが，ここでは言語の仕様とコンパイラの最適化について考えます．Fortran95は並列処理を意識した唯一の言語です．配列演算の項でも述べましたが，配列演算を用いた部分は並列処理可能であることがコンパイラには分かるので，コンパイラは最適化しやすくなります．Fortran95には既出の配列演算以外にも並列化を意識した機能があります．

```
  DO i = 1, n
    y(i) = 5.0_wp*MYFUNC(x(i), pi) + &
         & 3.0_wp
  ENDDO
...
  FUNCTION MYFUNC(x, a) RESULT(y)
...
```

上の例では自作関数MYFUNCに副作用があるか(xの値を変える，グローバル変数の値を変える等）どうか判別できないので，実際はMYFUNCに副作用がなくてもこのループを並列化することは困難です．MYFUNCがFortranの組み込み関数のSINなどであれば，コンパイラはSINに副作用がないことを知っているので，並列化してくれるかもしれません(くどいようですがコンパイラやCPUに依存します)．上記のルーチンを以下のように書き変えた場合はどうなるでしょうか

```
  FORALL(i=1:n)
     y(i) = 5.0_wp*MYFUNC(x(i), pi) + &
          & 3.0_wp
  END FORALL
...
  PURE FUNCTION MYFUNC(x, a) RESULT(y)
...
```

FORALL文によってコンパイラは並列処理可能であることがわかります．FORALLは並列演算を一般化したものでハイパフォーマンスフォートランとの互換性のため導入されました．配列演算が暗黙的並列処理と呼ばれるのに対してFORALLは明示的並列処理と呼ばれます．すべての配列演算はFORALL文で表せますが，FORALL文を配列演算では表せない場合があります．

`a(1:n−1) = a(2:n)*b(1:n−1)`は
`FORALL(i=1:n−1) a(i) = a(i+1)*b(i)`と同義ですが

```
FORALL(i=1:n, j=1:m)
  a(i,j) = REAL(i,wp)/REAL(j,wp)
  (a(i,j) = i/j と書いてはいけません.)
END FORALL
```

は配列代入では書けません．今までのDO...ENDDOのループをFORALL...END FORALLに変えることで明示的に並列処理ブロックを定義できます．上記例でPUREがついている関数MYFUNCは純関数と呼ばれ，副作用がありません．Fortran95のすべての組み込み関数は純関数です．FORALL...END FORALL内には純関数しか書けません．また純関数の一種で要素別（ELEMENTAL）関数というのも定義できます．要素別関数ではスカラー引数に配列を渡すことができます（戻り値も配列になります)．シングルスカラープロセッサのPCでは，明示的に並列処理を定義してもパフォーマンスの向上は期待できないかもしれません(CPUとコンパイラの組み合わせによっては擬似ベクトル化，擬似並列化ができるものもありますが)．しかしPCのCPUにも変化の兆しがあります．いまやCPUのクロック速度は頭打ちで，今後はマルチコア化が進むそうです．マルチコアCPUとFortran95言語仕様は非常に相性が良い気がします（コンパイラやOSが対応すればですが)．

最後に

私がFortran95を薦めるのは，プラズマ核融合関連のユーザーが増え有用なライブラリやツールが公開されれば，今後自分の解析の幅が広がったり，開発効率が向上したりするのでは，という下心があるためです．実は資産の共有ができるのならFortran95でなくても他の解析ツールのマクロやモジュールでも良いと思っています．ただ可搬性，汎用性，パフォーマンス，導入コスト，習得しやすさ，そして最大の利点である“見つけるのが非常に困難なバグ”［A3］の混入を比較的防ぎやすいという点でg95/gfortranが良いように思います．ツールは何であれ，資産が蓄積され，それが他の分野でも有用であれば，それはプラズマ核融合分野のアクティビィティの高さを示すことにつながると思います．

今回例で示したソースのフルバージョンを公開します．公開する以下のソースはすべてモジュールライブラリです．

nfnum.f95	整数，実数の精度指定およびpiなどの定義．すべてのモジュールに必要．
nfutil.f95	現時刻の所得やポインター関連のユーティリティ．
nfphys.f95	よく用いられる物理定数．
nffio.f95	file関連ユーティリティ．NEWUNIT，FINDFILEなど
nfstr.f95	文字列処理ユーティリティ．TOUPPER，TOLOWER，DELCHARなど

Appendix

A 1　Fortran と C/C++ の違いは？

細かな違いはたくさんありますが，一番大きな点はC/C++は汎用，Fortranは科学技術計算用という設計思想で

しょうか．本文でも述べましたが，Fortran は数式を使うアルゴリズムをほぼそのままコーディングできるように，という思想で設計されています．例えば配列要素の記憶順序が列順（数学の行列の表し方と同じ，これに対して C/C++ は行順で，ファイルからのデータ読み込みに有利），多次元配列を動的に確保できる（C/C++ は 1 次元配列のみ），配列インデックスの上限下限が定義できる（C/C++ ではインデックスの開始は 0 で固定），複素数を言語としてサポートしている，などでしょう．これらに加えて言語としての並列化のサポートなどがあります．コーディングのしやすさとは関係ありませんが，Fortran と C/C++ の思想の違いを示す面白い例がありますので紹介します．

```
n=10;
for (i=1;i<=n;i++){
  n－－;
  printf(”%d¥n”,i);
}
```

この結果は 1,2,3,4,5 となります．一方 Fortran では

```
n = 10
DO i = 1, n
  n = n － 1
  WRITE(*,*) i
END DO
```

この結果は 1,2,3,4,5,6,7,8,9,10 となります．Fortran では Do-Loop が何回回るかは，Do-Loop に最初に入ったときに決まってしまうからです．ちなみにこのループを抜けた後は i=11（10 ではない），n=0 となっています．このことから以下の例の結果も容易に想像できると思います．

C の例

```
n=10;
for (i=1;i<=n;i++){
  if(i==4) i=n－1;
  printf(”%d¥n”,i);
}
```

Fortran の例

```
n = 10
DO i = 1, n
  IF (i==4) i=n－1
  WRITE(*,*) i
END DO
```

C の例では結果は 1,2,3,9,10 となります．一方 Fortran ではコンパイルエラー（Do-Loop 内ではその制御変数を変えてはいけない）になります．ループの最初でループ回数を決めてしまうので，途中で制御変数を変えられては困ります．納得！です．

A 2　なぜ COMMON 文を使ってはいけないのか？

そもそも割り付け配列は COMMON BLOCK に入れることができません．このため使おうと思っても自然と COMMON 文の出番は少なくなりますが，それでも使う場合には COMMON 文には大きな問題があることを知っておかなければなりません．COMMON 文はなかなかエラーを出さないため，“見つけるのが非常に困難なバグ”の温床となりやすいのです．

例えば main で COMMON/TEST/A，B と定義して

```
SUBROUTINE SUB1
    COMMON/TEST/B，Y
```

と書いてもエラーにはなりません．このコードを別の人が見た時，作者はわざとそうしているのか（もしそうなら一度お会いして理由を伺いたいものです），エラーなのかわかりません．またこの仕様のため COMMON/TEST/A，B の B がどこで変更されたかを追跡するのに，単純に B だけをサーチするだけでは安心できません．さらに COMMON BLOCK はすべてユーザーに公開されるので，ユーザー（多くの場合，半年後の作者）は，誤って変更したくない変数を変えてしまうかもしれません．例えば

```
COMMON/DATA2D/nx, ny, nz, X(100), &
& Y(100), Z(100,100), nx_max, ny_max
```

という COMMON BLOCK があるとします．ここで，nx_max，ny_max は配列のサイズを表し，変えたくない（公開したくない）変数とします（COMMON BLOCK の変数は PARAMETER 属性を持てないためすべて変更可能）．この時 SUBROUTINE で nx だけを使いたいときでも，危険な変数 nx_max，ny_max をユーザーに公開しなければなりません．さらにこの DATA2D という COMMON BLOCK には問題があります．もし配列のサイズを100から200に変えたい場合はどうするのでしょう？すべての DATA2D を参照するサブルーチンで変更作業を行わなければなりません．そしてもし変更し忘れの個所があっても，多くの場合コンパイルエラーになりません．従来，このようなバグを防ぐため COMMON 文は INCLUDE 文と共に使われてきました．しかし，実は INCLUDE 文は FORTRAN77 標準ではなく，その正式サポートは Fortran90 からです．同じ Fortran90 標準を使うのなら，スコープの限定ができ，変数の公開，非公開が管理できるモジュールを使ったほうが安全です．

グローバル変数はモジュールで管理しよう，と言っていますが，なるべくグローバル変数を使わない方がコードの可読性は良くなります．グローバル変数が必要な場合はプログラムの設計が悪い場合がほとんどです．SUBROUTINE の引数を少なくするためだけにグローバル変数を導入するのには反対です．グローバル変数よりも，20個も引数のある SUBROUTINE の方がまだ良いと思います（引数の数については Appendix 4 参照）．

A 3　見つけるのが非常に困難なバグとは？

コンパイルエラー，実行時エラーが出ない，結果は一見正しそうだが良く見ると有効数字の 3 桁目以降が異なる，またはある条件では間違っている．このような場合，バグを探し出すのは非常に困難です．よくある例ですが，あるインタプリタ言語で，実数のつもりの a に対して，a＝1 としてしまったため a が整数として扱われた（Fortran95 では型不一致のエラーが出るか，警告を出すか，または実数に

変換して計算する)．C で if(i==0){...とするところで if(i=0){...としてしまった（C では i= 0 という代入演算は常に真を返す．Fortran95 ではコンパイルエラーが出る)．FORTRAN で DO 10 I = 0．10 としてしまったため，ロケットの制御が不能になった(古い FORTRAN の仕様では空白が意味を持たなかったので Do-Loop の誤りではなく，DO10I という暗黙的に実数として宣言された変数に 0.1 を代入する，と解釈されてしまった（注：この逸話は有名な作り話です)．Fortran95 ではコンパイルエラーが出ますので心配いりません)．FORTRAN はその長い歴史で様々なバグと戦ってきました．そのノウハウが Fortran95 の言語仕様に活かされています．私はいくつかの言語を使ってみて，Fortran が“発見困難なバグ”の混入を最も防ぎやすいと思います．

A 4　構造体はどんな時使うのか？

グローバル変数を使うぐらいなら20個も引数のある SUBROUTINE を使おう，と言いましたが，やはり20個の引数というのは少々使い勝手が悪いと思います．そんな時，構造体を用いて関連した変数をパッケージ化すれば，20個の引数は数個の構造体で表されるでしょう．それでは構造体の例を見てみましょう（プログラム例 A4)．ここではまだ POINTER の説明をしていないのですが，この例では”割り付け配列”とほぼ同義と思ってよいでしょう

プログラム例 A 4

```
!  構造体の宣言
TYPE ADCDATA
  CHARACTER(LEN=64)  :: diagnostic  = ''      ! 計測器の名前，ECE など
  INTEGER(i4b)       :: shotno     = 0        ! ショット番号
  INTEGER(i4b)       :: data_length = 0
! ADC の 1ch の取り込みデータ長
INTEGER(i4b)         :: channel_num = 0       ! 計測器の持つチャンネル数
REAL(wp)             :: t_sample   = 0.0_wp   ! サンプリング時間
REAL(wp)             :: t_delay    = 0.0_wp   ! データ取り込み開始時間
INTEGER(i4b), POINTER:: chans(:)   => NULL()
! 取り込みたい ADC のチャンネル番号．通常 chans = (/1,2,3,.../)であるがチャンネル不良,
! 増設，などで必ずしも 1 番から連続ではなくてもよい．その場合は
! chans = (/3, 5, 12, 13, 14,.../)のようになります．
  REAL(wp),   POINTER :: param(:)  => NULL()
! チャンネルの持つパラメータ．そのチャンネルのエネルギーや計測位置．
! REAL ではなく CHARACTER のほうが汎用性は高いかもしれません．
  REAL(wp),   POINTER :: time(:) => NULL()    ! 時間
  REAL(wp),   POINTER :: volt(:,:) => NULL()  ! 計測器の出力
END TYPE
!
!-------------------------------------------------------------------------------
!
  INTEGER(i4b), PARAMETER :: nplot = 8
  TYPE(ADCDATA) :: plotdata(nplot)
!  構造体配列を割り付け配列にする事も可能です．
  REAL(wp) :: t_plot_begin, t_plot_end
  REAL(wp) :: dummy
  CHARACTER(LEN=128) :: dir
!
  plotdata(1:nplot)%shot = 12345
  plotdata(1)%diagnostic = "Ip"
  plotdata(2)%diagnostic = "Bolometer"
  plotdata(3)%diagnostic = "ECE"
...
!  構造体のメンバーは%で参照します．
その他設定
!
  DO i= 1, nplot
    CALL ADC_CHANNEL_SETUP(plotdata(i))      ! 自作
    CALL ADC_GET_DATA(plotdata(i))           ! 自作
    CALL ADC_CONV_TO_PHYS(plotdata(i))       ! 自作
```

```
  ENDO
!
  t_plot_begin = 0.0_wp
  t_plot_end   = 0.0_wp
  DO i = 1, nplot
    dummy = MINVAL(plotdata(i)%time)
    t_plot_begin = MIN(t_plot_begin, dummy)
    dummy = MAXVAL(plotdata(i)%time)
    t_plot_end = MAX(t_plot_end, dummy)
  ENDDO
!   構造体のメンバーには通常の配列演算が行えます.
データの表示
!
  dir = "/home/inagaki/＃12345"
  DO i   =   1,   nplot
    CALL ADC_SAVE(plotdata(i), dir)          ! 自作
    CALL ADC_CLEAR(plotdata(i))              ! 自作
  ENDDO
```

(割り付け配列は構造体のメンバーにはなれないのです．この問題は Fortran2003 では解決しており g95/gfortran でも拡張として取り入れられています．しかし可搬性を考えるなら現時点では POINTER を使った方が良いと思います)．この例で特に強調したいのは，channel_num や data_length が計測器ごとに異なる場合，構造体を使うメリットは大きいということです．diagnostic などは構造体を使わなくても diagnostic(nplot) と配列にすればよいですが，time については time(:,:) という 2 次元配列を使ってもうまくいきません．C の「ディスプレイ」(別名ダブルポインタ，ポインタへのポインタ）のような「異なる長さを持つ配列の配列」という機能が必要です．Fortran では構造体以外に「ディスプレイ」を実現する方法はありません．構造体を使わずに，上記例を実現させようとすると time_Ip(:), time_Bolo(:), time_ece(:), ...と個別に配列を用意しなければなりません．chans, param，volt についても同様です．これでは計測器を追加するとき非常に手間がかかります．構造体を使うことで引数が多くなるのを防げるうえに，コードの可読性，可搬性があがることがわかります．この例を見て C++ に詳しい方は“おやっ”と思われたかもしれませんが，例えば CALL ADC_GET_DATA(plotdata(i)) の部分を C++ の plotdata[i].get_data() に置き換えてみると，上記 Fortran のプログラムは C++ のクラスを用いたプログラムと良く似てきます（C++ では構造体はクラスの一種なので，当たり前と言われれば当たり前なのですが...)．加えて先ほどから強調しているモジュールによるスコープの管理，変数の隠蔽などの機能を用いれば，オブジェクト指向のプログラミングもある程度可能になります．