チュートリアル

Fortran（FORmula TRANslation）は半世紀以上の歴史を持ち、一部では時代遅れと言われながらも、今なお数値計算に利用する研究者が多いプログラミング言語です。Fortran90/95の機能や使用例を改めて理解したいという研究者のために、Fortran90/95による近年の有限要素法プログラムを題材にして、岐阜大学の永井学志先生と橋本一輝氏に解説をお願いいたしました。今回のチュートリアルでは、有限要素法における全体剛性行列の作成方法について解説していただきます。なお、チュートリアル記事は1ページ目のみを本誌に掲載し、続きは日本計算工学会HP上で公開していますので、そちらも併せてご参照ください。

有限要素計算における全体剛性行列の作成法 ― 疎行列データ構造の視点から ―

永井 学志　橋本 一輝

1　はじめに

数年前より、いまどきのパソコンを対象として、いまさら新しくFEMプログラムを作っています。みなさまもご存知なように、どのように安いパソコンでも、今世紀に入ってからはマルチコア化とSIMDベクトル化がなされています。この状況は我々にとってうれしいものの、その性能を上手く使いこなすには、プログラミングの難易度があがっている ― 多階層の並列化が必須となった ― ように思います。

このような状況で最近のパソコン性能を享受するには、汎用FEMソルバとそのユーザサブルーチン、あるいは弱形式を直接に記述できるPDEソルバを使うのが、最も効率的でしょう。一方で、時代遅れを自覚しつつも、自分で本格的なFEMプログラムを組んでみたいという方もまだいらっしゃるのでは、とも思っています。ただ、ソースが付属した教科書の多くはFEM第一世代の諸先輩方によるものゆえ、最近の計算機環境に対応しきれない部分があります。

そこで、最近の計算機環境を踏まえつつ、私自身がFEMプログラミングの過程で学習したことを、恥を忍んでお話してみます。「そんなことも知らなかったの？」、「もっと改良できるよ」などのコメントは歓迎です。話の最後には、Fortran90/95＋α による抽象データ型に加えて、OpenMP並列化のFEMソースを公開します。

筆者紹介

ながい がくじ
岐阜大学 機械工学科 准教授。1995年 東京理科大学 建築学科卒、2000年 東京工業大学 環境物理工学専攻 博士課程修了。

はしもと かずき
名古屋大学 複雑系科学専攻 修士課程在学中。2013年 岐阜大学 数理デザイン工学科卒。

2　結局、全体剛性組み立て部とKu=f求解部

最近の計算機環境に対応したFEMプログラムと、第一世代プログラムとの違いは、a）全体剛性行列**K**と節点外力ベクトル**f**の組み立て部と、b）連立1次方程式**Ku=f**の求解部にあります。すなわち、a）組み立て部には要素のマルチカラー化による並列化と、b）**Ku=f**求解部には汎用な直接求解ライブラリの使用もしくは反復求解の並列化が主題となります。

もう少し詳しく述べると、両者a）、b）はひとつながりゆえ、それらの効率化のためには、**K**の疎行列性のデータ圧縮形式が重要です。ついつい計算アルゴリズムのみに注目しがちですが、メモリの使い方に関係するデータ構造も大切です。アルゴリズムとデータ構造の両輪で成っているのが、プログラムですので ― A＋D＝P ―。

そう言いつつも、全体剛性行列**K**をその引数が指定するデータ圧縮形式で作るのは、敷居が高いようです。現諏訪東京理科大学の河合先生が、「行方向圧縮記憶（CRS）形式を作れる時点で、ソルバに相当詳しいのですよ」と言っていたのを思い出します。

このチュートリアルでは、上述のa）、b）を帰結として、c）全体剛性行列**K**のCRS形式を作るための準備である、要素－自由度間コネクティビティのデータ解析と、d）その抽象データ型としてのカプセル化 ― ブラックボックス化 ― について述べます。

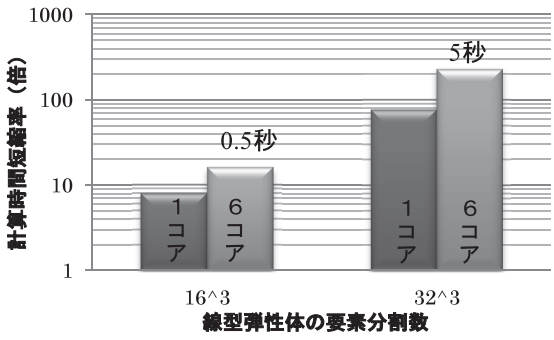


図1　CRS形式＋PARDISO直接求解ライブラリによる高速化（Intel Core i7-3930K）

全体を通して先に，図 1 に本稿の結論 ― 本稿を斜め読みすることのメリット ― を示しておきます．この図は，Intel Core i7-3930K マシンによる連立 1 次方程式の直接求解の計算速度を比較したものであり，スカイライン圧縮形式を自作の逐次実行ソルバで求解した場合を基準として，CRS 形式を Intel MKL Ver. 10.3 版の PARDISO [1] ソルバで求解した場合の計算速度比を示しています．自由度数が大きくなると，2 桁を超える計算時間の短縮が達成されています．自作の逐次実行ソルバは書籍 [2] を参考にしたものなので，それほど変な処理をしていないと思います．逆に，PARDISO が如何に職人芸の賜物であるかがお分かり頂けるのではないでしょうか？ 32^3 要素分割では，自作では求解に 20 分を要していたものが PARDISO ではわずか 5 秒で完了しています．おそらく 1 桁弱はメモリ共有型の並列化によるもの，もう 1 桁強は疎行列圧縮形式の効率化によるものだろうと想像しています．

以降では，この PARDISO のような汎用の直接解法ライブラリや，自作の反復求解ソルバにとって重要な，全体剛性行列 **K** の CRS 形式を上手く作るための方策について解説します．そのために，上記 a)～d)の項目について，ほぼ逆順で述べていくことにします．

3. 抽象データ型プログラミング

Fortran90/95 規約 [3),4)] による抽象データ型（Abstract Data Type, ADT）プログラミング [5),6)] の作法は，本稿において必須の考え方ではありません．しかし，この作法に一度慣れてしまうと，プログラミングの基本である「分割して統治せよ」の概念に素直に従うものゆえに，分割した際にその境界（インターフェイス）の定義さえ ― 実はこれが厄介なのですが… ― を強固にしておけば，非常に使い勝手の良いものです．Fortran でいうところの副プログラム単位（サブルーチンや関数の集合）間の独立性が格段に上がります．

抽象データ型プログラミングは，オブジェクト指向プログラミングの元となった考え方であり，現在ではそのサブセットと見なされています．すなわち，オブジェクト指向プログラミングの 3 大要素 ― カプセル化・継承・多態性 ― のうち，1 つ目のカプセル化に重点を置いたものです．Fortran90/95 規約では言語仕様のゆえか，カプセル化に加えて静的多態性 ― Fortran では総称名とよぶ ― も同時に考えているようです．

なお，静的多態性や総称名というと，ややこしく聞こえますが，絶対値を返す組込み関数 ABS などでよくお世話になっている仕組みです．この絶対値関数を自作しようとすると，引数が整数型，実数型，複素数型などのデータ型毎に，個別の関数を作る必要に気付きます．一方で，組込み関数 ABS ではデータ型を気にせずともよいことに疑問を感じた方もおられるかと思います．実際のところ，Fortran ではコンパイル時に ABS(a)の引数 a のデータ型をみて，基本整数型であれば IABS(a)を，倍精度実数型であれば DABS(a)を呼び出すように処理しています．このように総称名とは，引数のデータ型などに依存しないで，同じ機能に同じ FUNCTION 名・SUBROUTINE 名を付けておく仕組みです．

3.1 本題に入る前に，モジュールと構造体の概説

本題に入る前に，Fortran90/95 規約について 2 つだけ準備をさせてください．

1 つ目の準備では，MODULE 宣言による副プログラム単位の概念 ― 最近の言語のクラスに対応 ― を紹介します．これは次のような目的で用います．

i) 複数の副プログラム単位間で変数・定数群を共有
ii) 構造体群を定義（次段落にて紹介）
iii) CONTAINS 宣言以下に FUNCTION・SUBROUTINE 群を定義

なお，以降では Fortran90/95 規約の予約語は，基本的に大文字で表記します．i)の用法として，MODULE 名を constants として，基本的な定数群を次の program1 のように定義しておきます．

（Program1：基本的な定数群の定義例）

```
1:MODULE constants
2:  IMPLICIT none
3:  INTEGER, PARAMETER :: DP  = &
4:    & SELECTED_REAL_KIND(2*PRECISION(0.0))
5:  REAL(DP),PARAMETER :: PI = 3.14159265358979_DP
6:    ! 倍精度型の定数の最後には _DPが必須
7: END MODULE constants
```

ここで，3, 4 行目は倍精度型実数を定義するためのオマジナイ ― 種別型パラメータ DP の定義 ― で，5 行目中の REAL(DP)がその使用例です．これらの定数を別のプログラム単位で用いるためには，次で示すように，変数・定数を宣言する前に USE constants と宣言します．

2 つ目の準備では，TYPE 宣言による構造体の概念 ― Fortran にもようやく導入されました！ ― を紹介します．構造体は，データ型を組み合わせまとめて，新たに 1 つのデータ型とするものです．もっとも，この TYPE 構造体を，著者らが 5 年ほど前に試してみたところ，Intel Fortran でさえまったく最適化が掛からないという経験をしています．TYPE による構造体型の宣言は，TYPE 名を struct として，次の program2 のようにします ― 学生情報の一例 ― ．

（Program2：TYPE による構造体の宣言例）

```
1:  USE constants                    ! program1を使用
2:      :
3:  TYPE struct
4:    INTEGER  :: ID_No          ! 学籍番号
5:    INTEGER  :: birthday(3)    ! 誕生の年月日
6:    REAL(DP) :: hight          ! 身長
7:    REAL(DP), ALLOCATABLE :: score(:) ! Fortran2003
8:      :
9:  END TYPE struct
```

ここで,5 行目はあえて見やすさを優先し,Fortran90/95 の推奨記述法に従っていません．すなわち，INTEGER, DIMENSION(3) :: birthday とすべきところを，INTEGER ::birthday(3)としています．また，7 行目もあえて使いやすさを優先し，Fortran2003 規約も併用することで，構造体内の動的割り当て配列を宣言しています．この構造体 struct 型を用いるためには，次の program3 のように宣言します．

（Program3：TYPE による構造体の使用例）

```
 1:  TYPE(struct)              :: you
 2:  TYPE(struct)              :: couple(2)
 3:  TYPE(struct), ALLOCATABLE :: student(:)
```

you の各成分を指定するには，%記述子を用いて，you%ID_No や you%birthday(1)，you%birthday などと記述します．また，動的割り当て配列 student を用いるためには，次の program4 のようにします．

（Program4：構造体とその成分の動的割り当て例）

```
 1:  ALLOCATE( student(3) )                  ! 割り当て例
 2:  DO i = 1, 3
 3:    ALLOCATE( student(i)%score(132) )  ! 割り当て例
 4:  END DO
```

ここで，2 重で ALLOCATE していることにご注意ください．蛇足ですが，これを用いると jagged 配列 ― ギザギザ配列，C 言語でのポインタ配列まがい ― を実装できるようになります．

3.2 ようやく本題，抽象データ型プログラミング

前節 3.1 の MODULE 宣言と TYPE 宣言を組み合わせることで，Fortran90/95 による抽象データ型プログラミングの枠組みを，次の program5 に示します．これで 1 ファイル ADT_class.f90 とします．

（Program5:抽象データ型プログラミングの枠組み）

```
 1:MODULE ADT_class
 2:  USE constants   ! program1を使用
 3:  IMPLICIT none
 4:  PRIVATE         ! 原則，本MODULE外に公開しない
 5:      :
 6:  ! 抽象データ型を定義するためのカプセル化
 7:  TYPE, PUBLIC :: ADT_c  ! 例外で外部への公開名
 8:    PRIVATE              ! 以降の成分は公開しない
 9:    REAL(DP) :: c
10:      :
11:  END TYPE ADT_c
12:      :
13:  ! 総称名 ― 静的多態性，作用素 ― の宣言1
14:  PUBLIC :: initialize
15:  INTERFACE initialize
16:    MODULE PROCEDURE sub_init1, sub_init2
17:  END INTERFACE
18:  ! 総称名の宣言2
19:  PUBLIC :: do_something
20:  INTERFACE do_something
21:    MODULE PROCEDURE sub_do1, sub_do2
22:  END INTERFACE
23:      :
24:  PUBLIC :: finalize
25:  INTERFACE finalize
26:    MODULE PROCEDUFRE sub_final
27:  END INTERFACE
28:      :
29:CONTAINS
30:  ! 総称名initialize を構成する1つの具体的記述
31:  SUBROUTINE sub_init1( this, ...)
32:    TYPE(ADT_c), INTENT(inout) :: this
33:      :
34:  END SUBROUTINE sub_init1
35:  !
36:  ! 総称名initialize を構成するもう1つの具体的記述
37:  SUBROUTINE sub_init2( this, ...)
38:    TYPE(ADT_c), INTENT(inout) :: this
39:      :
40:    CALL hoge( this%c, ...) !さらにサブルーチンCALL
41:      :
42:  END SUBROUTINE sub_init2
43:      :
44:  ! モジュール内のみで有効なサブルーチン記述
45:  SUBROUTINE hoge( c, ... ) !最適化強制のバックドア
46:    REAL(DP) :: c           !最適化にはFortran77必須
47:      :
48:  END SUBROUTINE hoge
49:      :
50:END MODULE ADT_class
```

新しい規約や慣例がたくさん出てきてややこしく見えますが，やりたいことの本質は極めて単純です．次段落以降で 1 つ 1 つ順を追って説明させてください．

先に，program5 の MODULE を使う立場から説明します．これは，オブジェクト指向の説明でよく出てくる，「ドア」を「開ける」,「冷蔵庫」を「開ける」等のお話です．どのような「ブツ」でもとにかく「開ける」という，まったく同じ動作で表しておくと便利という考え方です． Program5 では,「ブツ」の型定義は 7 行目の構造体 ADT_c 型であり，動作「～する」の定義は 14,19,24 行目の総称名 initialize, do_something, finalize です．具体的な使用例を，次の program6 に示します．

（Program6：抽象データ型モジュールの使用例）

```
 1:  USE ADT_class   ! program5を使用
 2:      :
 3:  TYPE(ADT_c) :: obj_A          ! 「ブツ」の宣言
 4:      :
 5:  CALL initialize  ( obj_A, ...)
 6:      :
 7:  CALL do_something( obj_A, ...)
 8:      :
 9:  CALL finalize    ( obj_A, ...)
10:      :
```

構造体 ADT_c 型の obj_A を，まずは initialize して，次に do_something して，最後に finalize すると記述します．また，違う構造体型の obj_B があっても，まったく同じように obj_B を initialize して，do_something して， finalize すると記述します.「～する」の SUBROUTINE もそうですが,「ブツ」obj_A, obj_B

の内部が具体的にどう記述されているのか，知らないほうが使う立場からすると幸せです．

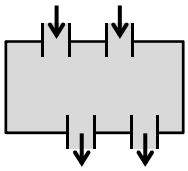
ここで，動作「～する」を実現するための SUBROUTINE の捉え方が，Fortran の一般的な手続き型プログラミングのものと少し違っていることにご注意ください．手続き型プログラミングでの SUBROUTINE の捉え方は，図 2 a)に示すように，数学での函数の捉え方と同じです．すなわち，いくつかのデータを入力すれば，内部であらかじめ決まった手順により加工されたものが出力されると考えます．そこに居る SUBROUTINE に対して適宜データを流していくと認識しているかと思います．一方で，抽象データ型プログラミングでの総称名 SUBROUTINE の捉え方は，図 2 b)に示すように，数学での作用素の捉え方と似ています．すなわち，それ単独では未完のままであり，データに寄生して初めて意味を成します．そこに居るデータに対して適宜 SUBROUTINE の 1 つが取り付き，仕事をしては離れていくと考えます．

今度は，program5 の MODULE を記述する側から説明します．基本的には本 MOUDLE 外には何も公開しないで自己完結 ― カプセル化 ― したいのです．しかし，例外的に許可したものだけ，外部に公開する ― インターフェイス ―．このようにしておくと，外部をあまり気にすることなく内部を自由に記述できます．これらを実現するのが，4, 7, 8, 14 行目などの PRIVATE, PUBLIC 宣言です．program5 で公開しているものは，上述した「ブツ」の型定義 ADT_c と，「～する」の定義 initialize, do_something, finalize のみです．不完全ながら，前者はオブジェクト指向でいうところのクラス，後者はメソッドに対応しています．なお，Fortran2003 では，よりオブジェクト指向に近いプログラミングができるようになっていますが，本稿では触れません．

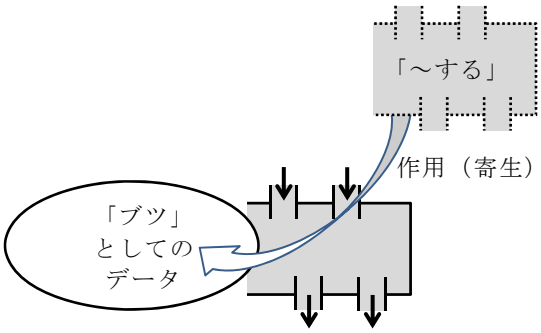
Fortran90/95 規約による抽象データ型プログラミングでは，重要な自主規制があります．すなわち，MODULE 外部へ公開する抽象データ型は常に 1 つであり，対応する抽象データは公開 SUBROUTINE･FUNCTION のすべてにおいて，常に第 1 引数 this としておきます．

改めて program5 を見ていきます．7～11 行目が MODULE 内で唯一の抽象データ型の定義であり，その型名 ADT_c を公開する一方で，各成分をカプセル化しています．14～17 行目が，対応する抽象データへの作用素としての SUBROUITNE initialize の公開宣言であり，さらにこれが総称名であり，その個別名は同 MODULE 内の SUBROUITNE sub_init1, sub_init2 であると INTERFACE 宣言にて定義しています．実際の記述は，29 行目の CONTAINS 宣言以下にあり，それぞれ 31～34 行目，37～42 行目にあります．これらの個別 SUBROUTINE は当然非公開なので，カプセル化されています．また，総称名は個別 SUBROUTINE の引数の違いにより，コンパイル時に静的に解決されます．

なお，計算効率を優先する場合には，40 行目と 45～48 行目で言及しているように，構造体の各成分を実引数として，下位の SUBROUTINE に引き渡しておきます．コンパイラは SUBROUTINE 毎に最適化を掛けることが多いようです．



a) 手続き型プログラミングの場合：函数



b) 抽象データ型プログラミングの場合：作用素

図 2　SUBROUTINE の捉え方

3.3　有限要素への抽象データ型の応用例

抽象データ型プログラミングの例として，program7,8 それぞれに，材料構成則と有限要素の枠組みを示してみます．著者らが抽象データ型をはじめた本当の動機は，研究室で色々な構成則や要素を試すにあたり，1 構成則 1 ファイル，1 要素 1 ファイルとして，ファイルを取り替えるだけでソース内部の変更は一切しないようにしたい，というものでした．どんどん増えてくる変数を引数に追加するのでなく，せっかく Fortran90 を使っているのだから，構造体の成分として入れてしまおうというのがはじまりです．実情は，概念が完全に固まっており，都合よいと思うところだけ控えめに抽象データ型にしています．

本稿を書き進めつつ，現状のコードでは，弾塑性構成則の内部変数や記憶しておくべき反復収束値の値，あるいはマルチフィジクスへの対応の弱さを残したままであったことに気づきました．具体的には，「各積分点での状態量」というあたりの「ブツ」と，「設定する」，「更新する」，「書き出す」というあたりの「～する」を定義しておけば，引数がよりシンプルになったかもしれません．今後の課題です．

（Program7：材料構成則の例）

```
1:MODULE material_class
2:
3:  USE constants
4:  IMPLICIT none
```

```
 5:  PRIVATE
 6:      :
 7:  INTEGER, PARAMETER, PUBLIC :: nstrn=3 !歪成分数
 8:      :
 9:  TYPE, PUBLIC :: mat_c    ! 物性値の記憶用
10:    PRIVATE
11:    REAL(DP) :: sy         ! 降伏応力
12:      :
13:  END TYPE mat_c
14:
15:  PUBLIC :: init           ! 初期化
16:  INTERFACE init
17:    MODULE PROCEDURE read_mat_data
18:  END INTERFACE
19:
20:  PUBLIC :: eval_Fin_and_K ! 内力と接線剛性等の評価
21:  INTERFACE eval_Fin_and_K
22:    MODULE PROCEDURE return_mapping
23:  END INTERFACE
24:      :
```

（Program8：有限要素の例）

```
 1:MODULE element_class
 2:
 3:  USE constants
 4:  USE material_class
 5:  IMPLICIT none
 6:  PRIVATE
 7:
 8:  INTEGER, PARAMETER, PUBLIC :: ndim   = 2 ! 次元数
 9:  INTEGER, PARAMETER, PUBLIC :: nnodee = 4 ! 節点数
10:
11:  TYPE, PUBLIC :: elm_c
12:    PRIVATE
13:    INTEGER :: matNo          ! 材料領域番号
14:    INTEGER :: nodeNo(nnodee) ! 全体節点への変換TBL
15:    REAL(DP):: eps_n(nstrn,4) ! ガウス点のひずみ
16:      :
17:  END TYPE elm_c
18:
19:  PUBLIC :: init           ! 初期化
20:  INTERFACE init
21:    MODULE PROCEDURE read_elm_data
22:  END INTERFACE
23:
24:  PUBLIC :: eval_Fin_and_K ! 内力と接線剛性の評価
25:  INTERFACE eval_Fin_and_K
26:    MODULE PROCEDURE eval_Fin_and_K_elm_c
27:  END INTERFACE
28:
29:  PUBLIC get_DOFg   ! 要素DOF→全体DOFの変換TBL
30:  INTERFACE get_DOFg
31:    MODULE PROCEDURE get_DOFe2g_TBL
32:  END INTERFACE
33:    :
```

4　おわりに

次号では，前章で述べた抽象データ型プログラミング作法を前提として，いよいよ第 1 章で言及した，c) 全体剛性行列 **K** の CRS 形式を作るための準備から述べていきます．すなわち，要素－自由度間コネクティビティのデータ解析から説明していきたいと思います．

参考文献

1) http://www.pardiso-project.org/ （2014.3 現在）
2) 寒川光，RISC 超高速化プログラミング技法，共立出版株式会社，1995
3) M. Metcalf el al., Fortran 95/2003 Explained, 2004
4) 富田博之，齋藤泰洋，改訂新版 Fortran90/95 プログラミング，培風館，2011
5) たとえば，Drew McCormack, Neglected FORTRAN ― Better use of f90 in scientific research ―, http://www.math.fsu.edu/acmath/Fortran90CourseNotes.pdf （2014.3 現在）
6) 竹内則雄，佐藤一雄，有限要素解析における Fortran90/95 とオブジェクト指向プログラミング, 計算工学講演会論文集, pp.199-202，vol.5 (2000 年 5 月)