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

前回は、Fortran90/95と一部にFortran2003を用いて、抽象データ型（Abstract Data Type, ADT）プログラミングの考え方と実例を示しました。これは、オブジェクト指向プログラミングから動的多態性と継承を除いたものです。データを流していくプログラミングのスタイルに慣れていらっしゃる方には、取っ付きづらい概念かもしれません。しかし、内部のデータを隠蔽 ― カプセル化 ― し、公開インターフェイスのみを介して外部とやり取りするというこのスタイルは、一度慣れてしまうと便利なものです。なぜならば、開発するプログラムには「ブツ（名詞）に、～する（動詞）」という言語化がダイレクトに反映されるためです。したがって、対象とする問題を分割統治し損なわないかぎり、数年前の自分 ― 赤の他人 ― が書いたプログラムも読み返し易い（？）ように感じています。

今回は、このADTプログラミングの考え方に基づき、FEM計算で必須の疎行列、すなわち全体剛性行列**K**と全体整合質量行列**M**の組立てまでを述べます。行列の疎性を活かした可逆圧縮のデータ構造には色々とありますが、ここでは圧縮行記憶（Compressed Row Storage, CRS）形式を対象とします。この圧縮データ構造は、非ゼロ成分のみを記憶するもののなかで、基本的なものの1つでしょう。

ところで、FEM創成期に諸先輩方が著された教科書には、少し不満があります。振り返ってみて、それは仕方がないと納得できることです。しかし仮に「疎行列やその組立て過程は、グラフ理論なる学問体系に他ならないので、そちらを学習しなさい」と強調しておいてくださっていれば…、という思いです。小国力編著の本[1]にも、グラフ理論はあまり知られていない旨の記述があります。全体自由度番号の付替えにはじまり、ウェーブフロント法やマルチフロンタル法、さらには消去木とシンボリックLU分解[2]など、いずれもこの理論の枠組みのようです。これらは我流の泥臭いアルゴリズムでも対応できるでしょうが、教科書的なアルゴリズムがあれば、そちらを学習するほうがきっと効率的です。

グラフ理論は図1に示すようなネットワーク ― 単にグラフという ― を分析する道具であり、対応するアルゴリズムが色々と提案されています。この応用は多岐に渡っており、カーナビなどの経路探索や、コンパイラによるレジスタ割付け解析などに使われているようです。グラフは、データ構造とアルゴリズムに関する教科書[3]の後ろの方に載っています。配列とリストにはじまり、ソート、2分探索、スタックとキュー、木、などを経てグラフに至ります。なお、筆者らがグラフというキーワードを知ったのは、電磁場解析で辺要素[4]を使わざるを得なかったためです。

本稿では、第2章で疎行列の組立て過程をグラフ視点から俯瞰したのち、第3章で2分探索木のADTプログラミングを、第4章でCRS形式までのADTプログラミングを述べます。なお、今回の逐次実行プログラムソースは、実用版を含め本学会誌HPに置いています。

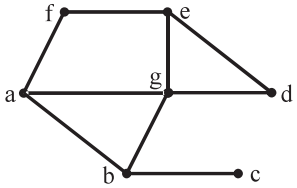

図1　頂点（vertex）と枝（edge）で定義されるグラフ（ネットワーク、路線、管網、フレームなど）

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　全体剛性行列の組立ての概要と全体設計

まず，基本事項の確認からはじめさせてください．骨組みのマトリクス構造解析や電気回路の節点解析などでは，対象とする系の変位や電位などの未知量を求めるために，「要素」と「節点」という概念を導入し，図 1 のようなネットワーク図を描きます. そのうえで，系を要素にバラして ― よって節点の概念を要素節点と全体節点に分化させて ― 考えます．すなわち，

1) 節点内力の概念を導入し，各要素が自由体として満たすべき平衡方程式を立て，
2) こちらの都合で一度バラしてみたものの，本来は要素節点と全体節点で変位や電位などが一致している，という適合条件を課し，
3) 全体節点において，力や電流などがつりあっている，という平衡条件を課すことで，

系全体が満たすべき平衡方程式を立て，未知変数について解くことになります．

線型問題であれば，系全体の係数行列 $\mathbf{K}$ は e 番目要素に関する係数行列を $\mathbf{K}_e$ として，数式上

$$\mathbf{K} \leftarrow \sum_{e=1}^{n_{\mathrm{elm}}} \mathbf{L}_e^{\mathrm{T}} \mathbf{K}_e \mathbf{L}_e \tag{1}$$

と記述できます．ここで，n_{elm} は全要素数，$\mathbf{L}_e$ とその転置 $\mathbf{L}_e^{\mathrm{T}}$ は e 番目要素に関する論理型の長方形行列であり，それぞれ上記手順 2), 3) に起因して出てくるものです．改めて，$\mathbf{K}$ は一般に大規模な疎行列，$\mathbf{K}_e$ はせいぜい数十次元の密行列であることに注意ください．非線型問題であれば，$\mathbf{K}$ はニュートン法のためのヤコビ行列であり，反復求解過程と増分過程で繰返し作成されます．

ここから具体例として，図 1 を何らかの物理問題として，かつ節点がスカラ自由度である場合，すなわち節点と自由度を同一視できる場合について，話を進めます．まず単純化のため，行列 $\mathbf{K}_e$ と $\mathbf{K}$ について，それらの成分がゼロか否かについてのみ注目してみます．成分がゼロか否かを 0 と 1 に対応づけて，それぞれ論理型行列 $\mathbf{A}_e$ と $\mathbf{A}$ とします．$\mathbf{K}_e$ は一般に密行列なので，

$$\mathbf{A}_e = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{2}$$

とすべて 1 にしておくのが最も安全です．$\mathbf{A}$ については，式(1)の総和 Σ を論理和 ∪ で置き換えて，

$$\mathbf{A} \leftarrow \bigcup_{e=1}^{n_{\mathrm{elm}}} \mathbf{L}_e^{\mathrm{T}} \mathbf{A}_e \mathbf{L}_e \tag{3}$$

と表記できます．この例での結果は，

$$\mathbf{A} = \begin{array}{c} \\ \mathrm{a(1)} \\ \mathrm{b(2)} \\ \mathrm{c(3)} \\ \mathrm{d(4)} \\ \mathrm{e(5)} \\ \mathrm{f(6)} \\ \mathrm{g(7)} \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} 1 & 1 & & & & 1 & 1 \\ 1 & 1 & 1 & & & & 1 \\ & 1 & 1 & & & & \\ & & & 1 & 1 & & 1 \\ & & & 1 & 1 & 1 & 1 \\ 1 & & & & 1 & 1 & \\ 1 & 1 & & 1 & 1 & & 1 \end{bmatrix} \end{array} \tag{4}$$

となります．ここで，行列 $\mathbf{A}$ の空欄成分は 0 であり，行と列それぞれの並び順は全体節点の名称の辞書順 ― a, b, c, …, g ― です．

式(3)の組立ての実際は後回しにして，その結果である式(4)を解釈してみます．この式は，全体節点間のつながり関係を真偽，1 と 0 として，リーグ表にしただけです．たとえば，図 1 のグラフでは，全体節点 a(1) は要素を介して，自分自身を含め全体節点 a(1), b(2), f(6), g(7) につながっています. この意味で, 式(2)は e 番目要素の要素節点同士がすべて相互につながっていることを，式(3)はこの関係の全体節点への変換と統合を表しています．

ここで今後の説明を容易にするため，グラフ理論 [3] における用語をいくつか紹介させてください．論理型行列 $\mathbf{A}$ を「隣接行列（adjacency matrix)」と称します．また，通常は自由度 ― ここでは節点 ― を「頂点（vertex)」，2 頂点間のつながり ― ここでは要素 ― を「枝（branch)」と称します．さらに，全成分が 1 である $\mathbf{A}_e$ に対応するグラフを「完全グラフ」，その頂点群を「クリーク（clique : 小集団)」と称します．なお，この例の場合，図 1 のグラフから直接的に隣接行列を構成するほうが容易です．しかし，一般の有限要素計算への応用においては，一度メッシュ絵を離れて隣接行列から新たにグラフを思い描くほうが好都合です．

本題に入ります．本稿の主題は式(1)の疎行列 $\mathbf{K}$ の組立て過程 $\sum_{e=1}^{n_{\mathrm{elm}}}$ です．主題の多くは，いま式(3)の隣接行列 $\mathbf{A}$ の組立て過程 $\bigcup_{e=1}^{n_{\mathrm{elm}}}$ に要約されました．また，非線型計算では，式(3)の組立ては基本的に一度で済みますが，式(1)の組立ては繰返し必要です．そこで，これら 2 つの組立て過程をそのまま親子 2 段構えで継承し，親である隣接行列（ADJacency MaTriX）に関するモジュール adj_mtx_class と，子である疎行列（SParse MaTriX）に関するモジュール sp_mtx_class に切り分けます．

両モジュールは，前回で述べた ADT プログラミングの考え方で記述します．すなわち，データ構造とアルゴリズムの抽象化により，抽象データ型とそのインターフェイスとして公開する総称名サブルーチン ― 抽象データ「ブツ」に寄り掛かる作用素「～する」 ― と，カプセル化すべき実装本体とに分離できます．そこで，実装本体は次章以降で述べるとして，ここでは使用者の視点から使い勝手を考え ― 実際には作成者の視点から計算効率も考え ―, 両モジュール内の抽象データ型 adj_mtx_c と sp_mtx_c と, それぞれに対する作用素の用法を先に定義します．具体的に，次の program1 のような用法にて定義します. Program1 ではすべてのサブルーチン CALL が作用素に該当し, それぞれ第 1 引数が作用を受ける抽象データです．

（Program1：逐次実行版の主プログラム）

```
1:PROGRAM main_serial_ver
2:  USE constants       ! 倍精度実数型の定数DPの定義
3:  USE adj_mtx_class   ! adj_mtx_c型に関するADT定義
4:  USE sp_mtx_class    ! sp_mtx_c 型に関するADT定義
```

```
 5:       :
 6:  IMPLICIT none
 7:       :
 8:  TYPE(adj_mtx_c)    :: A_pp          !隣接行列
 9:  TYPE( sp_mtx_c)    :: K_pp, M_pp    !疎行列
10:       :
11:  INTEGER, PARAMETER :: nelm  = 9     !図1の要素数
12:  INTEGER, PARAMETER :: nDOF_p= 7     !正の全体DOF数
13:  INTEGER, PARAMETER :: nDOF_n= 0     !非正の　〃
14:  INTEGER, PARAMETER :: nDOFe = 2     !要素DOF数
15:  INTEGER            :: DOFe2g(nDOFe) !DOFの変換TBL
16:  REAL(DP)           :: Ke(nDOFe,nDOFe),& !要素剛性
17:                        & Me(nDOFe,nDOFe)   !要素質量
18:       :
19:  !【第1段：親】隣接行列A_ppの作成
20:  CALL init( A_pp, nDOF_p )
21:  DO k = 1, nelm                        ! ∪_{e=1}^{n_elm}
22:     DOFe2g(:) = …
23:     CALL add_clique( A_pp, DOFe2g )
24:  END DO
25:  ! 確認用途：完成したA_ppのファイル出力
26:  CALL wrt( A_pp, 'A_pp.dat' )
27:       :
28:  !【第2段：子】疎行列K_ppとM_ppの作成
29:  CALL init( K_pp, A_pp )
30:  CALL init( M_pp, A_pp )
31:  DO k = 1, nelm                        ! Σ_{e=1}^{n_elm}
32:     DOFe2g(:) = …
33:     Ke(:,:)   = …
34:     Me(:,:)   = …
35:     CALL add_clique( K_pp, Ke, DOFe2g )
36:     CALL add_clique( M_pp, Me, DOFe2g )
37:        :
38:  END DO
39:  ! 確認用途：完成したK_ppのファイル出力
40:  CALL wrt( K_pp, 'K_pp.dat' )
41:       :
42:END PROGRAM main_serial_ver
```

ここで，プログラム中の自由度（DOF: Degree Of Freedoms）に関して，補足させてください．全体自由度番号は正負に渡る連番として，ディレクレ条件に関して非正の番号（対応して接尾語_n）を，それ以外に正の番号（接尾語_p）を付けます．計算ではディレクレ条件以外の自由度についてのみ疎行列が必須なので，数式上

$$\mathbf{A} = \begin{array}{c} \\ \text{ディレクレ}\leq 0 \\ \text{非ディレクレ}\ 0< \end{array} \begin{array}{c} \begin{array}{cc} \text{ディレクレ}\leq 0 & \text{非ディレクレ}\ 0< \end{array} \\ \begin{bmatrix} \mathbf{A}_{\mathrm{nn}} & \mathbf{A}_{\mathrm{np}} \\ \mathbf{A}_{\mathrm{np}}^{\mathrm{T}} & \mathbf{A}_{\mathrm{pp}} \end{bmatrix} \end{array} \tag{5}$$

と分割して，$\mathbf{A}_{\mathrm{pp}}$ に関してのみ考えます．また，配列による変換テーブル DOFe2g(:) ― 要素自由度番号（Element DOF num.）から全体自由度番号（Global DOF num.）を引く ― の命名法に関しても補足させてください．変換は函数 $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$ の言い換えに過ぎないので，$\boldsymbol{f}$ に対応する配列名を x2y(from x to y)とすることで，可読性を上げたつもりです．

話を戻して，Program1 を見ていきましょう．まず，3,4 行目でモジュール adj_mtx_class と sp_mtx_class の使用を宣言します．8，9 行目でそれぞれの抽象データ型 adj_mtx_c と sp_mtx_c で，隣接行列 A_pp と疎行列 K_pp，M_pp を宣言します．そのうえで，20 行目で隣接行列 A_pp を初期化（CALL init）し，21～24 行目で小集団（完全グラフ）を加算（CALL add_clique）します．26 行目は必須でないですが，完成した隣接行列 A_pp のファイル出力（CALL wrt）です．疎行列 K_pp，M_pp についても同様に，29, 30 行目で初期化（CALL init）し，31～38 行目で要素剛性行列 Ke と整合質量行列 Me の小集団を加算（CALL add_clique）します．

より詳細に見ると，29, 30 行目の CALL init では，隣接行列 A_pp を元に疎行列 K_pp，M_pp を初期化しています．元々，疎行列(1)から隣接行列(3)を切り出して考えたので，元に戻す処理が必要です ― オブジェクト指向でいうところの「継承」と「ダウンキャスト」―．ここで，抽象データ型 adj_mtx_c と sp_mtx_c の内訳を先読みしておきます．疎行列 K_pp，M_pp は同じ隣接行列 A_pp からの派生であるため，あえてデータのカプセル化を POINTER で破り，両疎行列から A_pp の同じデータ実体を指すようにします ― 詳細は次章 ―．このことのメリットは，メモリ効率の良さ以上に，複数の疎行列が対象としている問題の同一性が言えることです．たとえば，時刻歴解析のニューマーク β 法では，スカラ値を c として K_pp+c*M_pp を計算する作用素 add が必要ですが，両疎行列は同一の隣接行列 A_pp のデータ実体を指していますので，対象問題の同一性は既に保障されています．以上が使用者向けの話です．

3　結局，抽象データ型としての隣接行列の組立て

3.1　隣接行列と疎行列のデータ関係の切り分け

ADT プログラミングに向けて，隣接行列と疎行列のデータ関係の切り分けと，疎行列 sp_mtx_c 型の内部データ構造の設計からはじめます ― ここからは使用者でなく作成者の視点 ―．出発点は，式(4)の隣接行列 **A** に等価な圧縮表現の 1 つ，

$$\begin{array}{ll} \text{a(1)} & \{1,2,6,7\}, \\ \text{b(2)} & \{1,2,3,7\}, \\ \text{c(3)} & \{2,3\}, \\ \text{d(4)} & \{4,5,7\}, \\ \text{e(5)} & \{4,5,6,7\}, \\ \text{f(6)} & \{1,5,6\}, \\ \text{g(7)} & \{1,2,4,5,7\} \end{array} \tag{6}$$

です．この表現は，隣接行列 **A** の成分が 1 の列位置を行毎にまとめて集合{…}とし，さらにそれらを縦に並べたものです．グラフ理論 [3] では各集合{…}を「隣接リスト（adjacency list）」と称します．この名称は，図 1 のグラフと式(6)を見比べると，実に自然です．全体節点に関して，a(1)とつながっているものをすべてリストアップすると，自分自身を含めて a(1), b(2), f(6), g(7)です．

隣接行列 $\mathbf{A}_{\mathrm{pp}}$ を元に初期化される疎行列 $\mathbf{K}_{\mathrm{pp}}$ などでは，計算効率の視点から圧縮データ構造として CRS 形式が好まれます．CRS 形式は，表現(6)を区切りのための補助配列とともに 1 次元配列に展開・拡張したものといえます．具体的に sp_mtx_c 型の内訳となる CRS

形式は，次の program2 のようになります．

（Program2：sp_mtx_c と adj_mtx_c の関係）

```
 1:MODULE sp_mtx_class
 2:   USE constants       ! 倍精度実数型の定数DPの定義
 3:   USE adj_mtx_class   ! adj_mtx_c型に関するADT定義
 4:         :
 5:   IMPLICIT none
 6:   PRIVATE                              !基本は非公開
 7:         :
 8:   TYPE, PUBLIC :: sp_mtx_c             !抽象データ型
 9:     PRIVATE
10:     INTEGER, POINTER :: ind(:) => NULL()   !区切り
11:     INTEGER, POINTER :: DOF(:) => NULL()   !列位置
12:     REAL(DP), ALLOCATABLE :: a(:)    !疎行列の成分
13:           :
14:   END TYPE sp_mtx_c
15:         :
16:   PUBLIC init                           !総称名の公開
17:   INTERFACE init                        !作用素として
18:     MODULE PROCEDURE init_mtx_CRS    !(源氏名の正体)
19:   END INTERFACE
20:         :
21:CONTAINS
22:         :
23:   SUBROUTINE init_mtx_CRS ( this, adj )
24:     TYPE( sp_mtx_c), INTENT(inout) :: this
25:     TYPE(adj_mtx_c), INTENT(inout) :: adj !隣接行列
26:           :
27:     CALL associate( adj, this%ind, this%DOF )
28:     !この自作ダウンキャスト作用素はadj_mtx_class
29:           :
30:   END SUBROUTINE int_mtx_CRS
31:         :
32:END MODULE sp_mtx_class
```

前章の最後で先読みしたように，16～30 行目の初期化 init 中，27 行目 CALL associate により adj_mtx_c 型データ実体を指すようにしています．たとえば表現(6)の場合，列位置の配列 this%DOF(:)には(/1, 2, 6, 7, 1, 2, 3, 7, 2, 3, 4, 5, 7, 4, 5, 6, 7, 1, 5, 6, 1, 2, 4, 5, 7/)，区切りのための補助配列 this%ind(:)には(/1, 5, 9, 11, 14, 18, 21, 26/)が入っています．

しかし，上述のように組立て終えたのちに CRS 形式とするのは問題ないですが，隣接行列 $\mathbf{A}_{\mathrm{pp}}$ の組立て過程 $\bigcup_{e=1}^{n_{\mathrm{elm}}}$ を CRS 形式で行うのは無理があり，これを抽象データ型 adj_mtx_c の内訳とはできません．この問題は組立て過程

$$
\begin{array}{cc}
 & e=\mathrm{fe}(1) \\
\mathrm{a}(1) & \cup\{\,\} \\
\mathrm{b}(2) & \cup\{\,\} \\
\mathrm{c}(3) & \cup\{\,\} \\
\mathrm{d}(4) & \cup\{\,\} \\
\mathrm{e}(5) & \cup\{6,5\}, \\
\mathrm{f}(6) & \cup\{6,5\}, \\
\mathrm{g}(7) & \cup\{\,\}
\end{array}
\Rightarrow\cdots\Rightarrow
\begin{array}{c}
e=\mathrm{fe}(1)\sim\mathrm{ed}(4) \\
\cup\{6,1\}, \\
\cup\{\,\}, \\
\cup\{\,\}, \\
\cup\{5,4\}, \\
\cup\{6,5;\ 5,7;\ 5,4\}, \\
\cup\{6,5;\ 6,1\}, \\
\cup\{5,7\}
\end{array}
\Rightarrow\cdots \qquad (7)
$$

を考えると自明です．ここで，要素番号 e の番号並びは，要素名を頂点名 a, b, c,…を 2 つ書き連ねて，図 1 の上から右向きに fe(1), fa(2), eg(3), ed(4), …, bc(9) としています ― 辞書順にすると次節の説明が最悪ケースに（笑） ―．式(7)で注意頂きたいことは，組立て過程 $\bigcup_{e=1}^{n_{\mathrm{elm}}}$ を完了しないかぎり各隣接リスト{…}の成分数は未知なので，CRS 形式の 1 次元配列 this%DOF(:)に前もって展開できないという点です．各隣接リスト{…}は集合なので，並び順と重複に意味がないことにも注意ください．なお，組立て過程 $\bigcup_{e=1}^{n_{\mathrm{elm}}}$ （program1 の 19～24 行目）の前に全要素をループで走査すれば，各隣接リスト{…}の成分数の上限だけはおさえられて，配列を割付けることができますが，泥臭いデータ構造とアルゴリズムになってしまいます．

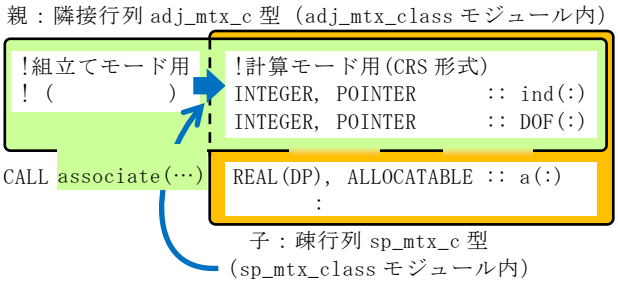

図 2　隣接行列 adj_mtx_c 型と疎行列 sp_mtx_c 型の設計

そこで，ADT プログラミングの考え方では使用者が抽象データ型の内訳を知り得ないことを逆手にとります．すなわち，隣接行列 adj_mtx_c 型の内訳として，図 2 に模式的に示すように，データ構造を 2 つ，組立てモード用の何らかのデータ構造と，計算モード用の CRS 形式とを用意します．そのうえで，隣接行列 A_pp を元に初めて疎行列をダウンキャスト初期化するタイミング ― program1 の 29 行目で CALL init(K_pp, A_pp)，これから program2 の 16～30 行目が呼出され，さらに同 27 行目で CALL associate ― にて，内部データ構造を組立てモード用から計算モード用に，不可逆で移行させます．

3.2　抽象データ型としての隣接行列の組立て

ここでは，モジュール adj_mtx_class における抽象データ型 adj_mtx_c と，それに対する作用素 init, add_clique, associate について，実装本体を考えます．データ構造は，前節で 2 つのモードに切り分けたので，組立てモードに特化しましょう．式(7)から分かるように，隣接行列を可逆圧縮した状態のまま組立てることから，工夫が必要です．式(6),(7)では縦方向と横方向があるため，2 層構造として考えるのが自然です．また，組立て過程 $\bigcup_{e=1}^{n_{\mathrm{elm}}}$ に要する時間は後の非線型計算を考えると無視できるので，行毎に伸びていくメモリ割付けの空間効率を重視しましょう．

組立て過程 $\bigcup_{e=1}^{n_{\mathrm{elm}}}$ の例(7)などをメモリ効率よく実現するには，配列表現でなくリスト表現の類が適しています．リストは構造型宣言 TYPE とポインタ宣言 POINTER を組み合わせて，再帰的に数珠つなぎしていくものです．ここで，配列表現に比べてリスト表現による初期化がどれほど遅くなるのか，連結リスト（linked list） ― 線形リスト ― の場合で実験してみ

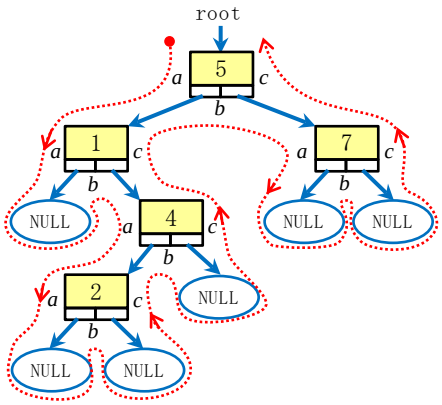

図 3　頂点 g(7) の隣接リスト組立てに対する 2 分探索木

ました．約 50 倍遅くなりました．原則的に逐一 ALLOCATE 命令でシステム側にメモリ割当てを要求するので仕方ないことです．しかし，4byte データを 25M 回（約 100Mbyte）繰返し割当てるために要する時間はわずか約 2 秒でしたので，無視します．

いま，隣接リストの集合 ∪{...} 毎に考えます．組立て過程(7)と完成版(6)を比べると，常に昇順を保ちつつ，clique（小集団）の加算毎に重複分については無視し，新規分のみをしかるべき位置に挿入していくのが得策でしょう．この作業に最適なデータ構造は，「2 分探索木（binary search tree）」[3] です．これも TYPE と POINTER の組み合わせで実現されるものです．なお，Java 言語では TreeSet という木のクラスがあらかじめ備わっています．

2 分探索木を完成させた一例を，図 3 に示します．この例は，式(7)の最終行 g(7)を完成させた隣接リスト ∪{5,7; 1,7; 7,4; 7,2} に対するものです．この木の作り方は，図中に示すように最上部を根（root）として，再帰的に 2 分しつつ下に木を茂らせていきます．このとき，右下に直上よりも大きな値を，左下に直上よりも小さな値を入れます．この例で最初から描くと，1) 値 5：根が指す直下のメモリ域に挿入，2) 値 7：根から辿り値 5 の右側ポインタが指すメモリ域に挿入，3) 値 1：根から辿り値 5 の左側ポインタが指すメモリ域に挿入，4) 値 7：根から辿り値 5 の右側を辿ると値 7 は挿入済みなので探索終了，…となります．描画上は 2 分ですが，処理上は，小さい場合，等しい場合，大きい場合の 3 分であることに注意ください．

式(7)全体では，このような木の「森（forest）」となります．2 分探索木のメリットは，整列済みの各リストの成分数を m ― 疎行列では数十程度 ― とすれば，探索に平均 $O(\log_2 m)$，その後の挿入には $O(1)$ の軽い処理回数となることです．なお，配列では工夫すれば探索を $O(\log_2 m)$ とできるものの，その後のしかるべき位置への挿入には再割付けの手間を除いても $O(m)$，逆に連結リストではしかるべき位置への挿入には $O(1)$ なものの，探索に $O(m)$ を要します．

図 3 のような 2 分探索木があるとき，図中に示すように，根の左側をスタートして木を常に左手に見ながら一巡して根に戻ってくることを「横断（traversal）」と称します．このとき，図中の各成分周りに付した領域 *a*, *b*, *c* のうち，領域 *b* を通り掛け（in-order）する際にその成分を順次ピックアップしていくと，(/ 1, 2, 4, 5, 7 /) と昇順整列済みのデータ列を得ることができます．したがって，組立てモード用のデータ構造を 2 分探索木としておき，初めて associate が呼出された際に，CRS 形式に移行させるのは容易です．これで図 2 中の空欄（ ）が「2 分探索木の森」で埋まり，模式図が完成しました．

そこで，隣接行列モジュール adj_mtx_class の本体実装では，2 分探索木（Binary Search Tree）モジュール BS_tree_class の抽象データ型 tree_c を下請けとして切り分けて，次の program3 のようにします．

（Program3：adj_mtx_class の実装）

```
 1:MODULE adj_mtx_class
 2:   USE BS_tree_class  ! tree_c型に関するADT定義
 3:         :
 4:   IMPLICIT none
 5:   PRIVATE                              !基本は非公開
 6:         :
 7:   TYPE, PUBLIC :: adj_mtx_c            !抽象データ型
 8:     PRIVATE
 9:     ! 組立てモード用（2分探索木の森）
10:     TYPE(tree_c), ALLOCATABLE :: tree(:)
11:     ! 計算モード用（CRS形式）
12:     INTEGER, POINTER :: ind(:) => NULL()   !区切り
13:     INTEGER, POINTER :: DOF(:) => NULL()   !列位置
14:   END TYPE adj_mtx_c
15:         :
16:   PUBLIC init                           !総称名の公開
17:   INTERFACE init                        !作用素として
18:     MODULE PROCEDURE init_adj_mtx
19:   END INTERFACE
20:
21:   PUBLIC add_clique                     !総称名の公開
22:   INTERFACE add_clique                  !作用素として
23:     MODULE PROCEDURE add_clique_adj_mtx
24:   END INTERFACE
25:
26:   PUBLIC associate                      !総称名の公開
27:   INTERFACE associate                   !作用素として
28:     MODULE PROCEDURE associate_adj_mtx
29:   END INTERFACE
30:         :
31:CONTAINS
32:         :
33:   SUBROUTINE init_adj_mtx ( this, nDOF_p )
34:     TYPE(adj_mtx_c), INTENT(inout) :: this
35:     INTEGER,         INTENT(in   ) :: nDOP_p
36:     ALLOCATE( this%tree(nDOF_p) )    !木の森を準備
37:   END SUBROUTINE init_adj_mtx
38:
39:   SUBROUTINE add_clique_adj_mtx ( this, DOFe2g )
40:     TYPE(adj_mtx_c), INTENT(inout) :: this
41:     INTEGER,         INTENT(in   ) :: DOFe2g(:)
42:     INTEGER :: buf(SIZE(DOFe2g))     !自動割付け配列
43:           :
44:     cnt = 0
45:     DO ie = 1, SIZE(DOFe2g)
46:        i = DOFe2g(ie)
```

```
47:      IF ( i > 0 ) THEN
48:        cnt = cnt +1
49:        buf(cnt) = i    !正の全体DOF番号のみを抽出
50:      END IF
51:    END DO
52:    DO ie = 1, cnt      !木々に複数データを一括挿入
53:      i = buf(ie)
54:      CALL insert( this%tree(i), buf(1:cnt) )
55:    END DO
56:  END SUBROUTINE add_clique_adj_mtx
57:
58:  SUBROUTINE associate_adj_mtx ( this, ind, DOF )
59:    TYPE(adj_mtx_c),  INTENT(inout) :: this
60:    INTEGER, POINTER, INTENT(out  ) :: ind(:)
61:    INTEGER, POINTER, INTENT(out  ) :: DOF(:)
62:         :
63:    IF ( ALLOCATED(this%tree) ) THEN !初CALL時のみ
64:      nnz = 0                        !  モード移行
65:      max_len = 0
66:      nDOF_p = SIZE(this%tree)
67:      DO i = 1, nDOF_p               !CRS作成の準備
68:        length = get_ndat( this%tree(i) )
69:        nnz = nnz + length      !Num. of Non-Zeros
70:        IF ( max_len < length ) max_len = length
71:      END DO
72:      ALLOCATE( this%ind(nDOF_p+1) )
73:      ALLOCATE( this%DOF(nnz)      )
74:      ALLOCATE( adj_list(max_len ) ) !取出用バッファ
75:      p = 1
76:      this%ind(1) = p
77:      DO i = 1, nDOF_p               !CRS作成
78:        CALL get_dat( this%tree(i), adj_list, length)
79:        !対称行列はプリプロセッサ命令 #ifdef で
80:        DO m = 1, length
81:          this%DOF(p) = adj_list(m)  !列位置
82:          p = p +1
83:        END DO
84:        this%ind(i+1) = p            !区切り
85:        CALL final( this%tree(i) )   !木の解放(必須)
86:      END DO
87:      DEALLOCATE( adj_list  )
88:      DEALLOCATE( this%tree )
89:    END IF
90:    ind => this%ind(:)          !区切りデータの共有
91:    DOF => this%DOF(:)          !列位置  〃     〃
92:  END SUBROUTINE associate_adj_mtx
93:       :
94:END MODULE adj_mtx_class
```

ここで，抽象データ型 adj_mtx_c の内訳は 7～14 行目であり，それに対する作用素 init の実装は 16～19，33～37 行目，add_clique は 21～24，39～56 行目，associate は 26～29，58～92 行目です．なお，疎行列の対称性を考慮する場合には，associate 内にプリプロセッサ命令#ifdef ～ #else ～ #endif を，SJIS の 5C 文字問題に注意しつつ書くのも一案です．

また，2 分探索木の実装は ADT の考え方によりほぼ隠されたことに注意ください．2 行目でそのモジュール BS_tree_class の使用を宣言したうえで，10 行目で adj_mtx_c の 1 成分として，その抽象データ型 tree_c の動的割付け配列である森 tree(:)を宣言しています．それに対する作用素の用法はそれぞれ，54 行目：木の探索および挿入の insert，68 行目：木の成分数を返す get_ndat，78 行目：木の全成分をバッファ配列に取出す get_dat，85 行目：木を解放する final です．

蛇足ですが，コンパイラとシステムに負荷を掛けがちな処理として，42 行目の自動割付け配列 buf(SIZE(DOFe2g))と，第 54 行目の部分配列の実引数 buf(1:cnt)があります．前者について，自動割付け配列はスタックメモリに積まれることが多いですが，小さな配列なので特に問題ありません．後者について，部分配列を実引数とするとき配列記述子 ― 上下限やストライドなど ― が対応する仮引数に渡されますが，コンパイラがストライド 1 に限定した最適化を掛けるとき ― Intel fortran では手続き間最適化の –ipo オプション ―，速度低下の問題は起こりません．

3.3　最下請けとしての 2 分探索木

プログラムを完結させるため，2 分探索木モジュール BS_tree_class についても，実装本体の一例を次の program4 に示しておきます．抽象データ型 tree_c に対する作用素，insert と get_dat については，内部サブルーチンによる再帰版を示します．スタックメモリのオーバフローを恐れる場合には，煩雑ですが非再帰で書き換えられます．

（Program4：BS_tree_class の実装）

```
 1:MODULE BS_tree_class
 2:  IMPLICIT none
 3:  PRIVATE                          !基本は非公開
 4:  TYPE, PRIVATE :: vtx          !2分探索木の基本要素
 5:    INTEGER :: c
 6:    TYPE(vtx), POINTER :: L => NULL()
 7:    TYPE(vtx), POINTER :: R => NULL()
 8:  END TYPE vtx
 9:
10:  TYPE, PUBLIC :: tree_c          !抽象データ型
11:    PRIVATE
12:    TYPE(vtx), POINTER :: root => NULL()
13:    INTEGER :: nvtx = 0           !頂点数(for効率化)
14:  END TYPE tree_c
15:
16:  PUBLIC :: insert                !総称名の公開
17:  INTERFACE insert                !作用素
18:    MODULE PROCEDURE insert_data_R
19:  END INTERFACE
20:
21:  PUBLIC :: get_ndat              !作用素
22:
23:  PUBLICT get_dat                 !総称名の公開
24:  INTERFACE get_dat               !作用素
25:    MODULE PROCEDURE traversal_asend_R
26:  END INTERFACE
27:       :
28:CONTAINS
29:       :
30:  SUBROUTINE insert_data_R ( this, a )
31:    TYPE(tree_c), INTENT(inout) :: this
32:    INTEGER,      INTENT(in   ) :: a(:)!挿入データ
33:         :
34:    DO i = 1, SIZE(a)
35:      CALL insert_kernel( this%root )  !内部Sub.
36:    END DO
```

```
37:  CONTAINS    !内部Sub. (上位変数は有効)
38:    RECURSIVE SUBROUTINE inset_kernel ( ptr )
39:      TYPE(vtx), POINTER, INTENT(inout) :: ptr
40:      IF ( ASSOCIATED(ptr) ) THEN      !終端でない
41:        IF      ( a(i) == ptr%c ) THEN
42:          RETURN                       !探索終了
43:        ELSE IF ( a(i) <  ptr%c ) THEN
44:          CALL insert_kernel( ptr%L )  !左下へ探索
45:        ELSE
46:          CALL insert_kernel( ptr%R )  !右下へ探索
47:        END IF
48:      ELSE                             !終端に到達
49:        ALLOCATE( ptr )                !メモリ割付
50:        ptr%c = a(i)                   !挿入
51:        this%nvtx = this%nvtx +1
52:        RETURN
53:      END IF
54:    END SUBROUTINE insert_kernel
55:  END SUBROUTINE insert_data_R
56:
57:  FUNCTION get_ndat ( this ) RESULT( ans )
58:    TYPE(tree_c), INTENT(in   ) :: this
59:    INTEGER                     :: ans
60:    ans = this%nvtx
61:  END FUNCTION get_ndat
62:
63:  SUBROUTINE traversal_asend_R ( this, buf, n )
64:    TYPE(tree_c), INTENT(in   ) :: this
65:    INTEGER,      INTENT(out  ) :: buf(:)
66:    INTEGER,      INTENT(out  ) :: n
67:          :
68:    cnt = 0
69:    CALL trvsl_inorder( this%root )  !内部sub
70:  CONTAINS    !内部Sub. (上位変数は有効)
71:    RECURSIVE SUBROUINE trvsl_inorder( ptr )
72:      TYPE(vtx), POINTER, INTENT(in   ) :: ptr
73:      IF ( .NOT. ASSOCIATED(ptr) ) THEN !終端に到達
74:        RETURN
75:      ELSE                              !終端でない
76:        CALL trvsl_inorder( ptr%L )     ! a)左下へ
77:        cnt = cnt +1                    ! b)中央
78:        buf(cnt) = ptr%c                !      出力
79:        CALL trvsl_inorder( ptr%R )     ! c)右下へ
80:      END IF
81:    END SUBROUTINE trvsl_inorder
82:  END SUBROUTINE traversal_asend_R
83:        :
84:END MODULE BS_tree_class
```

4　抽象データ型としての疎行列の組立て

疎行列 sp_mtx_c 型に対する作用素 add_clique の実装を次の program5 のようにして，program2 に追加します．CRS 形式の区切り this%ind(:) と列位置 this%DOF(:) を元に疎行列を組立てるには，this%a(:) において疎行列 $\mathbf{K}_{pp}$ の (i,j) 成分が格納される位置 p を探索する必要があります．この探索には，整列済み配列データに対する「2 分探索 (binary search)」[3] を用います．

2 分探索の一例として図 4 に，g(7)の隣接リストから値 5 の格納位置 4 を探索する様子を示します．再帰的に 2 等分を繰返して追い込んでいくことから，この探索は$O(\log_2 m)$であり，いまmは数十程度なので実質的に 10 回未満です ― それでも重い処理ですが… ―．

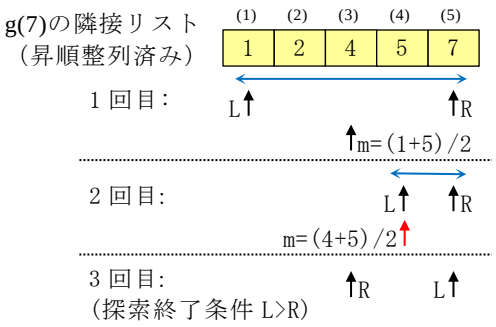

図 4　昇順整列済みデータに対する 2 分探索の例

22～26 行目にあるように，疎行列の第 i 行目のすべての列位置データは昇順にて this%DOF(this%ind(i):this%ind(i+1)-1)に格納されており，ここから CALL binary_search して第 j 列目のオフセット格納位置 loc を探索します．よって，$\mathbf{K}_{pp}$ の (i,j) 成分の格納位置は p = this%ind(i) -1 +loc と計算できます．

(Program5：sp_mtx_class への実装追加分)

```
 1:       :
 2:  USE sort_and_search_module, ONLY: binary_search
 3:       :
 4:  PUBLIC add_clique                  !総称名の公開
 5:  INTERFACE add_clique
 6:    MODULE PROCEDURE add_clique_CRS_mtx
 7:  END INTERFACE
 8:       :
 9:  SUBROUTINE add_clique_CRS_mtx ( this, Ke, DOFe2g )
10:    TYPE(adj_mtx_c), INTENT(inout) :: this
11:    REAL(DP),        INTENT(in   ) :: Ke  (:,:)
12:    INTEGER,         INTENT(in   ) :: DOFe2g(:)
13:         :
14:    nDOFe = SIZE(DOFe2g)
15:    DO ie = 1, nDOFe
16:      i = DOFe2g(ie)
17:      IF ( i > 0 ) THEN
18:        DO je = 1, nDOFe
19:          j = DOFe2g(je)
20:          !対称行列はプリプロセッサ命令 #ifdef で
21:          IF ( j > 0 ) THEN
22:            loc = binary_search( j,            &
23:                  & this%DOF(this%ind(i):      &
24:                  &          this%ind(i+1)-1)) )
25:            p = this%ind(i) -1 + loc
26:            this%a(p) = this%a(p) + Ke(ie,je)
27:          END IF
28:        END DO
29:      END IF
30:    END DO
31:  END SUBROUTINE add_clique_CRS_mtx
32:       :
```

2 分探索の関数 binary_search の本体は，program6 に示すように，ADT プログラミングの考え方でなく従来型として，とりあえず関数・サブルーチンの単なる寄せ集めモジュール sort_and_search_module に入れておきます．等号の存在ゆえに，名称にある 2 分でなく実質的に 3 分となるためか，短いながらもややこしいプログラムの代表例 [3] です．

（Program6:整列済み配列データに対する 2 分探索）

```
 1:MODULE sort_and_search_module
 2:  IMPLICIT none
 3:  PRIVATE
 4:  PUBLIC binary_search
 5:       :
 6:CONTAINS
 7:       :
 8:  FUNCTION binary_search ( j, a ) RESULT( loc )
 9:    INTEGER, INTENT(in   ) :: j    !探索すべき値
10:    INTEGER, INTENT(in   ) :: a(:) !昇順整列済み
11:         :
12:    left  = 1
13:    right = SIZE(a)
14:    DO WHILE ( left <= right )
15:      middle = ( left + right ) / 2
16:      IF ( j <= a(middle) ) right = middle -1
17:      IF ( a(middle) <= j ) left  = middle +1
18:    END DO
19:    it_exists = ( left - right == 2 ) !存在の有無
20:      IF ( it_exists ) THEN
21:         loc = middle             !格納された位置
22:      ELSE
23:         loc = left               !挿入すべき位置
24:    END IF
25:         :
26:  END FUNCTION binary_search
27:       :
28:END MODULE sort_and_search_module
```

5　おわりに

思っていた以上に，隣接行列の話が長くなりました．今回は，前回示した ADT プログラミングの考え方を復習しつつ，グラフや 2 分探索木を新たに導入しました．前回に続き，掲載が遅くなり申し訳ございませんでした．次回の最終回では，要素のマルチカラー化 ― ふたたびグラフの話 ― による作用素 add_clique 周りの Open-MP 並列化や，疎行列 sp_mtx_c 型に対する作用素 solve 内への PARDISO ソルバの組込み，さらに，いくつかの落穂 ― 連立 1 次方程式の反復求解と相性がよい，Reverse Cuthill-Mckee（RCM）法による全体自由度番号の並び替えなど ― を拾う予定です．

参考文献

1) 小国力編著，行列計算ソフトウェア ：WS、スーパーコン、並列計算機，丸善, 1991
2) たとえば，G.W. Stewart, Building an Old-Fashioned Sparse Solver, 2003, http://hdl.handle.net/1903/1312 (2014 年 6 月現在)
3) たとえば，石畑清，アルゴリズムとデータ構造，岩波書店, 1989
4) 五十嵐一ら，新しい計算電磁気学，培風館, 2003