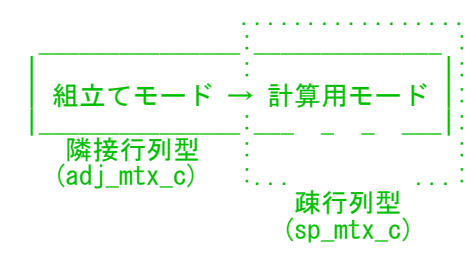
```
!*****************************************************************************************
!*** 疎行列に対する「隣接行列(ADJacency MaTriX)」を作成（組立てモード→計算用モード）  ***
!***                   使用データ構造：隣接行列を「リスト表現」で圧縮するために,        ***
!***                    ・組立てモード：「2分探索木の森(Forest of binary search trees)」 ***
!***                    ・計算用モード：「圧縮行格納(Compressed Row Storage, CRS)形式」  ***
!*****************************************************************************************
!
! 用途）「全体剛性行列」や「全体質量行列」等の
!       疎行列型 sp_mtx_c を使うための準備
!       （「継承」の概念で考えると,                  ）
!       （本adj_mtx_c型はスーパークラスであり        ）
!       （sp_mtx_c型はそのサブクラス.                ）
!       （adj_mtx_c型 変数からsp_mtx_c型 変数        ）
!       （へのダウンキャスト時にモードを移行.        ）
!
! 流用）有限要素のマルチカラー化
!
```



```
0018:
0019: !*******************************************
0020: ! 「隣接行列(Adjacency matrix)」の概要：
0021: !
0022: !    各有限要素の「要素自由度」は，それぞれ対応する「全体自由度」に接続されている．
0023: !    これを全体自由度の視点からみると，それらには有限要素を介して互いに接続関係にある
0024: !    ものがある．そこで，全体自由度に適当に名前（現実的には連番）を付けることにすれば，
0025: !    下図のように，これらの全接続関係をリーグ表で一覧にできる．このように，互いに接続
0026: !    関係にあるか否かを 0 or 1 の論理値で表したリーグ表を「隣接行列」という．
0027: !    （自分自身への接続を否 0 とする流儀が多いが，ここでは 1 とする）
0028: !
0029: !    有限要素法における隣接行列は，全体自由度の接続関係が無向なので対称行列であり，かつ
0030: !    全体自由度の総数に比べて接続関係にあるものが非常に少ない，すなわち，隣接行列の成分の
0031: !    ほとんどが 0 なので，大規模な「疎行列」となっている．この隣接行列が重要である理由は，
0032: !    「要素剛性行列」や「要素質量行列」を足し込んで組立てる「全体剛性行列」や「全体質量
0033: !    行列」の非ゼロ成分の位置を表わしているためである．それゆえ，隣接行列は疎行列の効率的な
0034: !    計算アルゴリズム（特にそれを係数行列とする連立１次方程式の求解）と，その記憶のための
0035: !    データ構造の基礎となる．なお，計算アルゴリズムのことを考えて，非正の全体自由度番号
0036: !     ― 拘束自由度 ― は隣接行列に含めないものとする．
0037: !
0038: !    続いて，隣接行列の特徴に言及しておく．隣接行列は全体自由度への番号の振り順（並び順）
0039: !    により見かけ上変化するが，全体自由度間の接続関係には変化がないのでその本質は変わらない．
0040: !    この特徴を活かして，大規模な疎行列に関して，計算アルゴリズムと記憶法がより効率的となる
0041: !    データ構造とするために，事前に全体自由度番号の「付け替え(renumbering)」＋αの工夫をする
0042: !    ことが多い．すなわち，下記の手順である（本モジュールにおいても，この手順を踏む）．
0043: !
0044: !      1) 無拘束の全体自由度にとりあえず1から順に適当に連番を振って，隣接行列を作る．
0045: !      2) この隣接行列を用いて，目的に応じて全体自由度への番号の振り順（並び順）を替える．
0046: !      3) 疎行列の計算に適したデータ構造（CRSなど）に変換する．
0047: !
0048: !    例)                    1 2 3 4 5 6 7  番号          1 2 3 4 5 6 7  新番号
0049: !        f_____e            a b c d e f g  名前          e d f g a b c  名前
0050: !       /      |\          1 a[ 1 1       1 1 ]      1 e[ 1 1 1 1       ]
0051: !      /       | \         2 b[ 1 1 1       1 ]      2 d[ 1 1   1       ]
0052: !   a/________g|__\d        3 c[   1 1         ]      3 f[ 1   1   1     ]
0053: !     \      /             4 d[       1 1   1 ]      4 g[ 1 1   1 1 1   ]
0054: !      \    /              5 e[       1 1 1 1 ]      5 a[     1 1 1 1   ]
0055: !       \  /               6 f[ 1       1 1   ]      6 b[       1 1 1 1 ]
0056: !        `b/_____c         7 g[ 1 1   1 1   1 ]      7 c[           1 1 ]
0057: !     1-D部材から成る構造       対応する隣接行列          番号付替え後の隣接行列 (RCM)
0058: !       （ネットワーク図）
0059: !
0060: !        特徴：
0061: !          ・図中のラベル a, b,… は全体自由度の名称．これらへの連番の振り順（並び順）は任意
0062: !          ・全体自由度番号の振り順（並び順）により，隣接行列は見かけ上変化するが，本質は不変．
0063: !          ・隣接行列は対称
0064: !
0065: !    補足）有限要素法概念の多くは「グラフ理論」という学問の枠組みが適用できる
0066: !      ・「グラフ」は「頂点(vertex)」とそのつながり方を表す「枝(arc,branch)」で構成される．
0067: !        【注：節点(node)や辺(edge)ともよばれるが，有限要素の用語との混用を避けるため用いない】
0068: !          有名なものには各種ネットワーク（基地が「頂点」，ケーブルが「枝」）や
0069: !          カーナビ（交差点が「頂点」，道路が「枝」）などがある．
0070: !          例1）各全体自由度を「頂点」と考えると，有限要素を介したそれらの接続関係は「枝」
0071: !          例2）各有限要素を「頂点」と考えると，全体自由度を介したそれらの接続関係は「枝」
0072: !      ・頂点が互いに枝で直結関係であることを「隣接する(adjacent)」という．
0073: !      ・注目する頂点につながっている枝の数を，その頂点の「次数(degree)」という．すなわち，
0074: !        自己ループの枝がない場合，隣接する頂点の数である．
0075: !      ・頂点がすべて互いに隣接関係にある集合を「クリーク(clique)」という．
0076: !          例）各有限要素が接続している全体自由度（頂点）の集合は「クリーク」
0077: !      ・隣接する頂点をそれぞれ相異なる色で塗分けることを「点彩色」という．
0078: !          例1）地図の4色塗分け問題（国土が頂点，国境を介した接続関係が枝）
0079: !          例2）並列処理のために有限要素集合を色分け
0080: !                    （有限要素を「頂点」，全体自由度を介した接続関係が枝）
0081: !          例3）レジスタ割付けのためにデータ集合を色分け
0082: !                              （データ変数が頂点，式を介した依存関係が枝）
0083: !
0084: !                          参考文献：石畑清「アルゴリズムとデータ構造」，p223～233，岩波書店
0085: !                                    http://en.wikipedia.org/wiki/Adjacency_matrix
```

```
0085:
0086: !
0087: !*********************************************
0088: !  【検討の過程】「リスト表現」による隣接行列の圧縮と，その状態下での組立て：
0089: !
0090: !     隣接行列は大規模な疎行列なので，2次元配列による素直な表現は必要メモリ量の視点から
0091: !     現実的でない．そこで，下図のように行毎に非ゼロ成分位置の列番号のみを取出す
0092: !     「リスト表現」で圧縮を図る．その具体的な実装（データ構造）については後述する．
0093: !     なお，各行のことを「隣接リスト(Adjacency list)」という．
0094: !     すなわち，隣接行列のリスト表現は隣接リスト群で構成されている．
0095: !
0096: !       ___ ___ ___ ___                  ___ ___ ___ ___ ___
0097: !     1 a |_1_|_2_|_6_|_7_|            1 e |_1_|_2_|_3_|_4_|
0098: !     2 b |_1_|_2_|_3_|_7_|            2 d |_1_|_2_|_4_|
0099: !     3 c |_2_|_3_|___|               3 f |_1_|_3_|_5_|___ ___
0100: !     4 d |_4_|_5_|_7_|___            4 g |_1_|_2_|_4_|_5_|_6_|
0101: !     5 e |_4_|_5_|_6_|_7_|          5 a |_3_|_4_|_5_|_6_|
0102: !     6 f |_1_|_5_|_6_|___ ___        6 b |_4_|_5_|_6_|_7_|
0103: !     7 g |_1_|_2_|_4_|_5_|_7_|      7 c |_6_|_7_|
0104: !         リスト表現による圧縮              リスト表現による圧縮
0105: !           （番号付替え前）                （番号付替え後）
0106: !
0107: !     ところで，このリスト表現による圧縮は，完成した隣接行列(疎行列)に対するものであり，
0108: !     作成途中については何ら考えていない．当然，作成過程もリスト表現による圧縮下の
0109: !     ままで行うべきであるものの，少し工夫が要る．すなわち，隣接行列(疎行列)の作成は下図
0110: !     に示すように，リーグ表に逐一ビットを立てることで要素毎に組立てていく過程であるが，
0111: !     リスト表現の具体的な実装を考慮すると，組立てが完了するまで必要メモリ量が不明な
0112: !     ことに注意しなければならない．くわえて，最終的に行毎の列番号は昇順としておくほうが
0113: !     データ処理の観点から好ましい．
0114: !                          隣接行列                  リスト表現（圧縮表現）
0115: !       要素ab             1 2 3 4 5 6 7
0116: !                          a b c d e f g             ___ ___
0117: !                  1 a [ 1 1               ]       1 a |_1_|_2_|
0118: !                  2 b [ 1 1               ]       2 b |_1_|_2_|
0119: !                  3 c [                   ]       3 c |
0120: !                  4 d [                   ]       4 d |
0121: !                  5 e [                   ]       5 e |
0122: !                  6 f [                   ]       6 f |
0123: !                  7 g [                   ]       7 g |
0124: !
0125: !       要素af             1 2 3 4 5 6 7
0126: !                          a b c d e f g             ___ ___ ___
0127: !                  1 a [ 1 1       1       ]       1 a |_1_|_2_|_6_|
0128: !                  2 b [ 1 1               ]       2 b |_1_|_2_|
0129: !                  3 c [                   ]       3 c |
0130: !                  4 d [                   ]       4 d |
0131: !                  5 e [                   ]       5 e |___ ___
0132: !                  6 f [ 1         1       ]       6 f |_1_|_6_|
0133: !                  7 g [                   ]       7 g |
0134: !
0135: !       要素ag             1 2 3 4 5 6 7
0136: !                          a b c d e f g             ___ ___ ___ ___
0137: !                  1 a [ 1 1       1 1     ]       1 a |_1_|_2_|_6_|_7_|
0138: !                  2 b [ 1 1               ]       2 b |_1_|_2_|
0139: !                  3 c [                   ]       3 c |
0140: !                  4 d [                   ]       4 d |
0141: !                  5 e [                   ]       5 e |___ ___
0142: !                  6 f [ 1         1       ]       6 f |_1_|_6_|
0143: !                  7 g [ 1           1     ]       7 g |_1_|_7_|
0144: !
0145: !          :                   :                       :
0146: !
0147: !     そこで，以下では隣接リストによる圧縮表現の具体的な実装（データ構造）について，
0148: !       A) 組立て過程
0149: !       B) 完成後の疎行列計算
0150: !     という2つの視点から述べていく．くわえて昨今の計算機事情として，次の視点も考える．
0151: !       C) メモリ共有型並列化の難易度
0152: !
0153: !
```

```
0154:
0155: !*********************************************
0156: !  【検討の過程】リスト表現による隣接行列の圧縮に対する具体的な実装（データ構造）の候補：
0157: !
0158: !    まず，リスト表現に対する素直な実装としてギザギザ(jagged)配列が挙げられる．
0159: !    このギザギザ配列は次のようにして実現される．
0160: !
0161: !       TYPE :: jag
0162: !          INTEGER, ALLOCATABLE :: DOF(:)
0163: !       END TYPE
0164: !       TYPE(jag) :: row(n)    ! n:隣接行列の次元数
0165: !
0166: !    第 i 行第 j 列目の非ゼロ要素と 記憶位置 p の関係は次の通り．
0167: !       j = row(i)%DOF(p)
0168: !
0169: !    しかし，ギザギザ配列は組立て過程と疎行列計算の双方ともに問題があるので，候補から
0170: !    外すほうがよい．それらの問題点とは，すなわち，
0171: !
0172: !      A) 組立て過程における問題点：
0173: !
0174: !         1) 事前に必要メモリ量が不明であるにも関わらずメモリを割付けねばならない．
0175: !            これには次の2つの割付け方があるが，いずれも少々不恰好．
0176: !              a) 2パス化：組立てに先立ち，全要素をループして必要メモリ量の上限を見積もる
0177: !                           （上限は，その全体自由度を共有している全要素の総自由度数で抑えうる）
0178: !              b) 再割付け(re-malloc)によりメモリを倍々にしていく
0179: !
0180: !         2) 各行の列番号が重複しているときの扱い方．これには次の2つが考えられる．
0181: !              a) 重複を無視してとにかく終端に追記し，組立て完了後にマージを兼ねて昇順整列処理
0182: !                 （単純で処理速度は速かろうが，泥臭いアルゴリズム）
0183: !              b) 重複を毎回確認する．この確認をO(n)からO(log(n))へと効率化するには，昇順となる
0184: !                 位置への追記と二分探索を組合せる
0185: !                 （教科書的なアルゴリズム．ただし，配列を使うとデータ移動が多くなるので，
0186: !                  構造型とポインタで実現する「2分探索木(Binary Search Tree)」を使うのが常套）
0187: !
0188: !      B) 疎行列計算における問題点：
0189: !
0190: !           計算速度の点からはほぼ問題ないものの，疎行列ライブラリに受け渡すデータ構造としては
0191: !           一般的でなく互換性に欠ける．
0192: !
0193: !    一方で，C) のメモリ共有型の並列化については問題ない．すなわち，B) 疎行列計算は読み出しのみ
0194: !    なので並列化は容易である．また，A) 組立て過程では書き込み，すなわちデータ競合発生の可能性
0195: !    があるが，全有限要素の集合を適切に色分け（Multi-coloring）しておけば並列化時のデータ競合は
0196: !    回避できる．さらに，メモリの再割付けも疎行列の行番号毎に個別に処理できるので並列化できる．
0197: !
0198: !
0199: !    そこで，上記のA)，B) の問題について，ギザギザ配列を工夫することで，いずれか一方のみを主に
0200: !    解決できる実装を次のア）とイ）それぞれに挙げる（基本的に，あちらを立てればこちらが立たず）．
0201: !
```

ア）単一 1次元配列への展開「圧縮行格納(Compressed Row Storage, CRS)」

改善点：B) 疎行列ライブラリに受け渡すデータ構造として一般的なものとなる.
改悪点：C) 単一配列では行毎のメモリ割付けが不可ゆえ，組立て過程の並列化が難しくなる.

ア-1）補助配列に開始・終了位置を格納する実装　（番号付替え前を例として）

```
 1_________4_________7_________10________13__14__15_____17____
|_1_|_2_|_6_|_7_|_x_|_x_|_1_|_2_|_3_|_7_|_x_|_x_|_2_|_3_|_4_|_5_|_7_|_x_|

    19________22________25____27____29_____________33____________ ←位置
   |_4_|_5_|_6_|_7_|_x_|_x_|_1_|_5_|_5_|_x_|_1_|_2_|_4_|_5_|_7_|_x_|_x_|_x_|←列番号

 a   b   c   d   e   f   g
 1___2___3___4___5___6___7__    ← 行番号
|_1_|_7_|13_|15_|19_|25_|29_|   ← 開始位置
|_4_|10_|14_|17_|22_|27_|33_|   ← 終了位置
```

長所：開始・終了位置を指定するため，行毎の列番号を自由な位置に格納できる.
　　蛇足ながら，この実装では泥臭いながらも全体自由度番号の付替えも，同一配列内で処理できる.
短所：組立て時のメモリ割付けは，上述の A-1-a の2パス化以外にない.
　　（このとき，A-2 は a) の選択が自然．また1パス目も一応並列化できるハズ…）
長所：A-1-a と A-2-a によると，泥臭いながらも組立て過程と番号付替えに対応できる.

補足）本データ構造が最初の実装であった（組立てと計算をin-placeで兼用できるため）
　　（番号付替え後，メモリアクセス順を考えて，列毎のデータの記憶位置について，
　　前詰めはできるが前後入替までは難しい）
　　なお，改訂後の実装では疎行列計算専用に用いることとなった（後述の結論）

ア-2）補助配列に開始位置のみを格納する実装　（番号付替え後を例として）

```
 1_______________5_________8_________11_________________
|_1_|_2_|_3_|_4_|_1_|_2_|_4_|_1_|_3_|_5_|_1_|_2_|_4_|_5_|_6_|

                              16______________23______________27_____29          ←位置
                             |_3_|_4_|_5_|_6_|_4_|_5_|_6_|_7_|_6_|_7_| x        ←列番号

 e   d   f   g   a   b   c
 1___2___3___4___5___6___7____    ← 行番号
|_1_|_5_|_8_|11_|16_|23_|27_|29_|  ← 開始位置
```

長所：疎行列計算のための最終データ構造として最も効率的
短所：メモリに一切の空がないため，メモリの空を利用してやり繰りが必要となる
　　組立て過程と番号付替えへの対応は絶望的.

0251:
0252: ¦　　イ）行毎の隣接リストを構造型とポインタにて「2分探索木(Binary Search Tree)」で構成し，
0253: ¦　　各木の根をポインタ配列につなぎとめることで，「森」とする実装
0254: ¦　　（2分探索木の詳細については，BS_tree_class を参照のこと）
0255: ¦　　（素直な「連結リスト」では追記はO(1)と有利だが，探索がO(n)と不利になるので却下）
0256: ¦
0257: ¦　　注目する全体自由度 ：　a(1)　　2　・・・　5　・・・　7
0258: ¦　　（ポインタ配列）　　|　　|　・・・　|　　|
0259: ¦　　根(root)：　f(6)　　1　　6　　5
0260: ¦　　　　　　　／＼　　　＼　　＼　　／＼
0261: ¦　　葉(leaf)：a(1)　g(7)　　2　　5　　1　7
0262: ¦　　　　　　　＼　　　／＼　　／＼　　＼
0263: ¦　　　　　　　b(2)　　3　7　　4　7　　4
0264: ¦　　　　　　　　　　　　　　　　　　　／
0265: ¦　　　　　　　　　　　　　　　　　　　2
0266: ¦
0267: ¦　　注）2分探索木には，根から葉を伸ばしていくときに左右に2分し，左側の子は親より小さい値，
0268: ¦　　右側の子は親より大きい値になるように，格納していく．これからも分かるように，
0269: ¦　　データの格納順により木の茂り方が変わる．
0270:
0271: ¦　　改善点：A）組立て過程が，極めて教科書的・簡潔・スマートなアルゴリズムとできる．
0272: ¦　　改悪点：B）疎行列計算用のデータ構造としてますます一般的でなくなる．
0273: ¦
0274: ¦　　イ-1）素直に構造型とポインタで実装
0275: ¦　　長所：データ構造も教科書的，簡潔＆スマートとなる．ユーザがメモリ管理を一切気にせず
0276: ¦　　ともよく，処理系が裏でヒープメモリ域を管理してくれる．C）の並列化も容易．
0277: ¦　　（並列化の結果，効率が良いかどうかは別問題．マルチスレッド時，スレッド毎に
0278: ¦　　個別のヒープメモリを管理するようになっていれば効率が良い．しかし，
0279: ¦　　単一ヒープメモリ域を共有管理していれば，lock機構による逐次処理をせざるを
0280: ¦　　得ず，効率が劇的に悪化）
0281: ¦　　短所：構造型とポインタの機能を，ユーザが巨大な単一配列上で管理する場合に比べて
0282: ¦　　処理速度が 数十倍（1桁強）落ちる．
0283: ¦　　（もっとも，有限要素解析，特に非線形解析を考えると，この性能悪化は許容範囲）
0284: ¦
0285: ¦　　イ-2）構造型とポインタをあえて巨大な単一配列で実装
0286: ¦
0287: ¦　　短所：単一配列でユーザがメモリ管理するため，逐次処理となる．よってC）の並列化は困難．
0288: ¦　　長所：素直に構造型とポインタで実現する場合と比べて，処理速度の低下はない．
0289: ¦　　補足）本データ構造は，組立部の2つ目の実装であった（組立てが教科書的とできるため．
0290: ¦　　速度低下は避けたかったのでこのデータ構造としたが，最終的に並列化でつまずいた．
0291: ¦　　スレッド毎に独立な巨大配列を準備する羽目になるが，あまりに煩雑ゆえ諦めた）
0292: ¦
0293: ¦
0294: ¦　　ウ）番外編：列方向にも2分探木やハッシュテーブルを用いて，行毎の隣接リストの実装をつなぎ
0295: ¦　　とめる案も考えうるが，次元が決まっており，基本的にすべての行に平均的にデータが入って
0296: ¦　　いく有限要素法では実用的でない．
0297: ¦
0298: ¦
0299: ¦　以上の考察に加えて，全体自由度番号の付替えは上記の ア-1）以外は基本的に out-place となる
0300: ¦　ことを踏まえて，本クラスのデータ構造は以下で実現する．
0301: ¦　　i）組立て過程はイ-1）による．組立て完了後に全体自由度番号の付替えを行ったのち，
0302: ¦　　ii）（ダウンキャスト時に）疎行列計算に適した ア-2）に変換する．（逆変換しないものとする）
0303: ¦
0304: ¦　なお，疎行列の組立てでなく隣接行列の組立てというステップをわざわざ別途切り出したのは，
0305: ¦　動的計算や特に非線形有限要素計算への対応を念頭に置いているためである．これらの計算では
0306: ¦　隣接行列が同一であるものの，複数の疎行列あるいは時々刻々変化する接線剛性行列
0307: ¦　― 全ポテンシャルエネルギの2階微分，ヘッセ行列 ― が出てくる．これらに対しては隣接行列
0308: ¦　をまず定義しておき，その後に要素剛性行列 Ke あるいは 要素整合質量行列 Me を繰返し加算
0309: ¦　していくほうが合理的であろう．
0310: ¦

```
0311:
0312:
0313: MODULE adj_mtx_class
0314:
0315:    USE BS_tree_class
0316:    USE sort_and_search_module
0317:    IMPLICIT none
0318:    PRIVATE
0319:
0320:    TYPE, PUBLIC :: adj_mtx_c
0321:       PRIVATE                                ! nDOF_p: Num. of global DOFs of Plus index
0322:       ! 組立てモードのために，リスト表現を2分探索木の森で
0323:       TYPE(tree_c), ALLOCATABLE :: tree(:) !(nDOF_p  )  2分探索木の根を記憶するポインタ配列
0324:       ! 計算用モードのために，リスト表現を圧縮行格納(CRS)形式で（疎行列に結合される）
0325:       INTEGER, POINTER :: ind(:) => NULL() !(nDOF_p+1)  第i行目の隣接リストの格納開始位置
0326:       INTEGER, POINTER :: DOF(:) => NULL() !(nnz     )  列の全体DOF番号（隣接リスト）群
0327:    END TYPE adj_mtx_c                        !         FORALL( p=ind(i):ind(i+1)-1 ) DOF(p)
0328:
0329:    PUBLIC :: init
0330:    INTERFACE init
0331:       MODULE PROCEDURE init_adj_mtx
0332:    END INTERFACE
0333:
0334:    PUBLIC :: final
0335:    INTERFACE final
0336:       MODULE PROCEDURE final_adj_mtx
0337:    END INTERFACE
0338:
0339:    PUBLIC :: add_clique
0340:    INTERFACE add_clique
0341:       MODULE PROCEDURE add_clique_adj_mtx
0342:    END INTERFACE
0343:
0344:    PUBLIC :: associate
0345:    INTERFACE associate
0346:       MODULE PROCEDURE associate_adj_mtx
0347:    END INTERFACE
0348:
0349:    PUBLIC :: renumber
0350:    INTERFACE renumber
0351:       MODULE PROCEDURE renumber_adj_mtx
0352:    END INTERFACE
0353:
0354:    PUBLIC :: do_coloring
0355:    INTERFACE do_coloring
0356:       MODULE PROCEDURE do_coloring_by_Welsh_Powell
0357:    END INTERFACE
0358:
0359:    PUBLIC :: wrt
0360:    INTERFACE wrt
0361:       MODULE PROCEDURE wrt_adj_mtx
0362:    END INTERFACE
0363:
0364: CONTAINS
0365: !
```

```
0366:
0367: !*****************************************************************************************
0368:    SUBROUTINE init_adj_mtx ( this, nDOF_p )
0369:
0370:       ! 隣接行列 this の初期化（隣接行列は，正(plus)の連番が振られた全体DOFだけで構成）
0371:
0372:       TYPE(adj_mtx_c), INTENT(inout) :: this
0373:       INTEGER,         INTENT(in   ) :: nDOF_p ! 最大の全体DOF番号（=隣接行列の次数）
0374:
0375:       CALL final_adj_mtx( this )               ! フェイルセイフ
0376:       ALLOCATE( this%tree(nDOF_p) )            ! 隣接リスト(2分探索木)の「森」を準備
0377:
0378:    END SUBROUTINE init_adj_mtx
0379:
0380: !*****************************************************************************************
0381:    SUBROUTINE final_adj_mtx ( this )
0382:
0383:       TYPE(adj_mtx_c), INTENT(inout) :: this
0384:
0385:       INTEGER :: i
0386:
0387:       IF ( ASSOCIATED(this%DOF) ) DEALLOCATE(this%DOF)
0388:       IF ( ASSOCIATED(this%ind) ) DEALLOCATE(this%ind)
0389:
0390:       IF ( ALLOCATED(this%tree) ) THEN
0391:          DO i = 1, SIZE(this%tree)
0392:              CALL final( this%tree(i) )
0393:          END DO
0394:          DEALLOCATE(this%tree)
0395:       END IF
0396:
0397:    END SUBROUTINE final_adj_mtx
0398:
0399: !*****************************************************************************************
0400:    SUBROUTINE add_clique_adj_mtx ( this, DOFe2g )
0401:
0402:       ! 組立てモードにて，隣接行列 this にクリークを追記（ブーリアン加算）
0403:
0404:       TYPE(adj_mtx_c), INTENT(inout) :: this
0405:       INTEGER,         INTENT(in   ) :: DOFe2g(:)! 要素DOF番号から全体DOF番号への変換テーブル
0406:
0407:       INTEGER :: nDOF_p                  ! 最大の全体DOF番号（=隣接行列の次数）
0408:       INTEGER :: nDOFe                   ! 要素DOFの数
0409:       INTEGER :: cnt                     ! 正番号が振られた全体DOFの数（ cnt < nDOFe ）
0410:       INTEGER :: buf( SIZE(DOFe2g) )     ! 正の全体DOF番号のみの格納バッファ（自動割付け配列）
0411:       INTEGER :: ie, i                   ! 要素DOF番号，全体DOF番号
0412:
0413:       IF ( .NOT. ALLOCATED(this%tree) ) &
0414:                    & STOP "add_clique: 隣接行列が初期化されていないか，組立てモードでない"
0415:
0416:       nDOF_p = SIZE(this%tree)
0417:       nDOFe  = SIZE(DOFe2g)
0418:
0419:       ! まず，正の全体DOF番号だけを buf(1:cnt) に抽出
0420:       cnt = 0
0421:       DO ie = 1, nDOFe
0422:          i = DOFe2g(ie)
0423:          IF ( i > nDOF_p ) STOP "add_clique: 全体自由度番号がinitで指定した値を超えている"
0424:          IF ( i > 0 ) THEN
0425:             cnt = cnt +1
0426:             buf(cnt) = i                 ! DOFe2g 中の全体DOF番号に重複があっても問題なし
0427:          END IF
0428:       END DO
0429:
0430:       ! 次に，バッファに抽出した全体DOF番号を 隣接リスト（2分探索木）の森に登録
0431:       DO ie = 1, cnt
0432:          i = buf(ie)                     ! 第 i 行（ 1 <= i <= nDOF_p は保証済み）
0433:          CALL insert( this%tree(i), buf(1:cnt) )
0434:       END DO
0435:
0436:    END SUBROUTINE add_clique_adj_mtx
0437: !
```

```
0438:
0439: !*****************************************************************************************
0440:    SUBROUTINE associate_adj_mtx ( this, ind, DOF )
0441:
0442:       ! 隣接行列 this 中のCRS形式に, ポインタ DOF(:) & ind(:) を結合する.
0443:       ! なお, 初めてよばれたときに 組立てモードから計算用モードに移行する（不可逆）.
0444:       ! （複数の全体剛性行列や全体質量行列間でこれらを共有することで, 同一問題なことを保証）
0445:
0446:       !       （継承と考えてサブクラスからCALLされたとみる場合, スーパークラスである
0447:       !        adj_mtx_c型 をダウンキャストしていることになる）
0448:       !
0449:       ! 参考：F90だけでなくJavaでも, オブジェクトを同型に代入する場合, POINTER属性のものは
0450:       !       データそのものでなく, データ記憶域を指すポインタのみがコピーされる.
0451:       !       なお, ALLOCATABLE 属性のものはデータそのものがコピーされる
0452:
0453:       TYPE(adj_mtx_c),  INTENT(inout) :: this
0454:       INTEGER, POINTER, INTENT(out  ) :: ind(:) ! ポインタ結合状態へのINTENT指定は F2003規約
0455:       INTEGER, POINTER, INTENT(out  ) :: DOF(:)
0456:
0457:       IF (       .NOT. ASSOCIATED(this%DOF )   &
0458:          & .AND. .NOT. ALLOCATED (this%tree) ) &
0459:          & STOP "associate: 隣接行列が初期化されていない"
0460:
0461:       ! 初めて呼び出されたときのみ, 「組立てモード」から「計算用モード」に不可逆で移行
0462:       IF ( ALLOCATED(this%tree) ) CALL transform_mode ( )
0463:
0464:       DOF => this%DOF(:)
0465:       ind => this%ind(:)
0466:
0467:    CONTAINS
0468: !
```

```
0469:
0470:       SUBROUTINE transform_mode ( )
0471:
0472:          INTEGER :: nDOF_p                ! 最大の全体DOF番号 (=隣接行列の次数)
0473:          INTEGER :: nnz                   ! Num. of Non-Zeros for allocating DOF(1:nnz)
0474:          INTEGER :: i                     ! 隣接行列の第 i 行
0475:          INTEGER, ALLOCATABLE :: buf(:)   ! 第i行の隣接リストを木から取出すためのバッファ
0476:          INTEGER :: deg1_i                !        〃          長さ = 木の頂点数 = (次数 +1)
0477:          INTEGER :: max_deg1              ! 最大の 隣接リスト長さ (バッファ割付け用)
0478:          INTEGER :: p, q, start
0479:
0480:          nDOF_p = SIZE(this%tree)
0481:
0482:          ! 非ゼロ成分数 nnz を得る (ついでに隣接リストを取出すためのバッファ長 max_deg1 も)
0483:          nnz      = 0
0484:          max_deg1 = 0
0485:          DO i = 1, nDOF_p
0486:             deg1_i = get_ndat( this%tree(i) )
0487:             IF ( deg1_i == 0 ) STOP "associate: 全体自由度への番号付けにトビがあり連番でない"
0488:             ! 上記の例外処理とクリークの性質により, 隣接行列の全対角項の存在が保証できる
0489:             nnz = nnz + deg1_i
0490:             IF ( max_deg1 < deg1_i ) max_deg1 = deg1_i
0491:          END DO
0492:          ! 注) 上記ループ内で this%ind(i+1) = nnz +1 とできそうだが, SYM_MTX 時には無理.
0493:
0494:          ! --- プリプロセッサは SJIS の 5C 文字問題に対応していないので要注意！ ---
0495: #ifdef SYM_MTX
0496:          ! 対称行列の場合, nnz を上三角の非ゼロ要素数に書き換えておく.
0497:          nnz = ( nDOF_p + nnz ) / 2
0498:          ! 無向グラフ (隣接行列は対称) では ( nDOF_p + nnz ) は 必ず偶数.
0499:          ! 理由 : ( nnz - nDOF_p ) は非対角項の非ゼロ数は, 上三角・下三角 (対称)
0500:          !        それぞれの非対角項の非ゼロ数の和なので, 偶数.
0501:          !        よって, nDOF_p + nnz = ( nnz - nDOF_p ) + 2 * nDOF_p も偶数.
0502: #endif
0503:
0504:          ALLOCATE( this%ind(nDOF_p+1) )
0505:          ALLOCATE( this%DOF(nnz     ) )
0506:
0507:          ALLOCATE( buf(max_deg1) )
0508:          p = 1                            ! 第1行目の開始位置は 1
0509:          this%ind(1) = p
0510:          DO i = 1, nDOF_p
0511:             CALL get_dat( this%tree(i), buf, deg1_i )
0512: #ifdef SYM_MTX
0513:             ! 対称行列の場合, 上三角のみ取出す ( i は必ず buf(1:deg1_i) 中に存在)
0514:             start = binary_search( i, buf(1:deg1_i) )
0515: #else
0516:             start = 1
0517: #endif
0518:             DO q = start, deg1_i
0519:                this%DOF(p) = buf(q)       ! p = this%ind(i) + q - start としても同じ
0520:                p = p +1
0521:             END DO
0522:             this%ind(i+1) = p
0523:             !==========>>>>>>>>>>
0524:             CALL final( this%tree(i) )    ! ifortでは, ここがOMP並列化時に遅くなる.
0525:             !<<<<<<<<<<==========         ! (マルチスレッド用 KMP_FREE() の仕様)
0526:          END DO
0527:          ! 終了時には, p == nnz +1 となる
0528:          DEALLOCATE( buf )
0529:
0530:          DEALLOCATE( this%tree )
0531:
0532:          WRITE (*,'("associate: 次元数 =",I10," ,非ゼロ数 =",I12,2X,"(行あたり : ",F5.1,")")') &
0533:                & nDOF_p, nnz, nnz/REAL(nDOF_p)
0534:
0535:       END SUBROUTINE transform_mode
0536:
0537:    END SUBROUTINE associate_adj_mtx
0538: !
```

```
0539:
0540: !*****************************************************************************************
0541:    SUBROUTINE wrt_adj_mtx  (this, fl_name )
0542:
0543:       ! 隣接行列this を Coodinate(COO)形式 ― ija形式 ― で出力
0544:
0545:       TYPE(adj_mtx_c), INTENT(in   ) :: this
0546:       CHARACTER(*),    INTENT(in   ) :: fl_name
0547:
0548:       INTEGER :: uid = 30
0549:       INTEGER :: nDOF_p                  ! 最大の全体DOF番号 (=隣接行列の次数)
0550:       INTEGER :: i, j                    ! 隣接行列の i行 j列
0551:       INTEGER, ALLOCATABLE :: buf(:)     ! 第i行の隣接リストを木から取出すためのバッファ
0552:       INTEGER :: deg1_i                  !        〃        長さ = 木の頂点数 = (次数 +1)
0553:       INTEGER :: max_deg1                ! 最大の 隣接リスト長さ (バッファ割付け用)
0554:       INTEGER :: p, q
0555:
0556:       OPEN (UNIT=uid, FILE=fl_name )
0557:
0558:       IF ( ALLOCATED(this%tree) ) THEN    ! 組立てモードの場合
0559:
0560:          ! 隣接リストから全成分を取出すためのbuf(:)を準備
0561:          nDOF_p = SIZE(this%tree)
0562:          max_deg1 = 0
0563:          DO i = 1, nDOF_p
0564:             deg1_i = get_ndat( this%tree(i) )
0565:             IF ( max_deg1 < deg1_i ) max_deg1 = deg1_i
0566:          END DO
0567:          ALLOCATE( buf(max_deg1) )
0568:
0569:          DO i = 1, nDOF_p
0570:             CALL get_dat( this%tree(i), buf, deg1_i )
0571:             DO q = 1, deg1_i
0572:                j = buf(q)
0573:                WRITE(uid,'(2I10)') i, j
0574:             END DO
0575:          END DO
0576:
0577:       ELSE IF ( ASSOCIATED(this%DOF) &    ! 計算用モードの場合
0578:         & .AND. ASSOCIATED(this%ind) ) THEN
0579:
0580:          nDOF_p = SIZE(this%ind) -1
0581:          DO i = 1, nDOF_p
0582:             DO p = this%ind(i), this%ind(i+1)
0583:                j = this%DOF(p)
0584:                WRITE(uid,'(2I10)') i, j
0585:             END DO
0586:          END DO
0587:
0588:       ELSE
0589:          STOP "adj_mtx_c型の変数が初期化されていない"
0590:       END IF
0591:
0592:       CLOSE(uid)
0593:
0594:       WRITE(*,'("Coodinate(COO)形式の隣接行列をMATLABで可視化するには, 次のコマンド:")')
0595:       WRITE(*,'("ij = load(''",A,"'');")') fl_name
0596:       WRITE(*,'("K_pp = sparse(ij(:,1), ij(:,2), NaN );")')
0597:       WRITE(*,'("spy(K_pp);")')
0598:
0599:    END SUBROUTINE wrt_adj_mtx
0600: !
```

```
0601:
0602: !*****************************************************************************************
0603:    SUBROUTINE renumber_adj_mtx ( this, node2DOFg ,edge2DOFg_ )
0604:
0605:       ! 組立てモードにて，自由度に振っている全体DOF番号を付替えて，その結果を
0606:       ! 隣接行列 this と 変換テーブル node2DOFg(:,:), edge2DOFg_(:) に反映させる.
0607:       ! （付替えは，とりあえず Reverse Cuthill-Mckee (RCM) ordering のみ）
0608:
0609:       TYPE(adj_mtx_c),    INTENT(inout) :: this
0610:       INTEGER,            INTENT(inout) :: node2DOFg(:,:) !(nDOFn,nnode)  節点DOF番号
0611:       INTEGER, OPTIONAL,  INTENT(inout) :: edge2DOFg_(:)  !(nedge)        辺  DOF番号
0612:
0613:       INTEGER :: nDOF_p  ! > 0   最大の全体DOF番号 ( plus の全体DOF番号は非変位拘束を表現)
0614:       INTEGER :: nDOF_m  ! < 0   最小の    〃      ( minus    〃           変位拘束  〃  )
0615:       INTEGER :: nDOFn, nnode, nedge
0616:       INTEGER ::  i,  j
0617:       INTEGER :: io, in                          ! i-th Original/Old, i-th New
0618:       INTEGER,       ALLOCATABLE :: o2n(:)       ! 全体DOF番号の付替え前(old)から新(New)への変換
0619:       INTEGER,       ALLOCATABLE :: n2o(:)       ! その逆変換
0620:       TYPE(tree_c), ALLOCATABLE :: tree_tmp(:)
0621:
0622:       IF ( .NOT. ALLOCATED (this%tree) ) THEN  ! フェイルセイフ
0623:          IF (    ASSOCIATED(this%DOF ) ) THEN
0624:             STOP "renumber: 隣接行列が組立てモードのときのみ有効（計算用モードに移行済み）"
0625:          ELSE
0626:             STOP "renumber: 隣接行列が初期化されていない"
0627:          END IF
0628:       END IF
0629:
0630:       ! 全体DOF番号を付替えるために，順引き o2n(:) を作成
0631:       nDOF_p = SIZE(this%tree)
0632:       nDOF_m = MINVAL(node2DOFg)                    ! 引数を減らした副作用（効率悪化は軽微）
0633:       IF ( PRESENT(edge2DOFg_) ) THEN
0634:          nDOF_m = MIN ( nDOF_m, MINVAL(edge2DOFg_) )
0635:       END IF
0636:       ALLOCATE( o2n(nDOF_m:nDOF_p),&
0637:                 & n2o(     1:nDOF_p) )
0638:       CALL mk_RCM_table( this%tree, n2o )           ! 付替えアルゴリズムはとりあえずRCMで
0639:       ! 注) in = n2o(io) の io, in は1対1の対応にあることが前提（このときFORALLもDOも同じ）
0640:       FORALL (in=nDOF_m:     0) o2n(    in ) = in  ! 非正の全体DOF番号はそのまま
0641:       FORALL (in=     1:nDOF_p) o2n(n2o(in)) = in  ! 参考) f(f^{-1}(y)) = y
0642:       DEALLOCATE( n2o )
0643:
0644:       ! 付替え結果 o2n(:) を 隣接リストthis%tree(:) （隣接行列の行と列）に反映
0645:       ALLOCATE( tree_tmp(nDOF_p) )
0646:       !$OMP PARALLEL DO PRIVATE( in )
0647:       DO io = 1, nDOF_p
0648:          in = o2n(io)
0649:          tree_tmp(in) = this%tree(io)                  ! 行の移動
0650:          CALL renumber( tree_tmp(in), o2n(1:nDOF_p) )  ! 列の入替え
0651:       END DO
0652:       !注) 構造体の成分のうちPOINTER属性があるものは，ポインタ値のみコピーされる
0653:       !     （比較：ALLOCABLE属性があるものは，新たに配列がALLOCATEされ中身が全コピーされる）
0654:       !注) 順番に巧く玉突きしていけばtree_tmp(:)は不要なようにみえるが，以外と難しい.
0655:       !    これは，4x4マス目上で15枚パネルを逐一スライドさせ並び替えるパズルが難しいのと同じ.
0656:       CALL MOVE_ALLOC( tree_tmp, this%tree )        ! F2003規約
0657:       ! this%treeが指していたメモリ域は解放され，新たにtree_tmpの指していたメモリ域を指す
0658:
0659:       ! 付替え結果 o2n(:) を node2DOFg_DB(:,:) と edge2DOFg_DB_(:) に反映
0660:       nDOFn = SIZE(node2DOFg,1)
0661:       nnode = SIZE(node2DOFg,2)
0662:       FORALL ( i=1:nDOFn, j=1:nnode ) node2DOFg(i,j) = o2n( node2DOFg(i,j) )
0663:       IF ( PRESENT(edge2DOFg_) ) THEN
0664:          nedge = SIZE( edge2DOFg_ )
0665:          FORALL ( i=1:nedge ) edge2DOFg_(i) = o2n( edge2DOFg_(i) )
0666:       END IF
0667:
0668:       DEALLOCATE( o2n )
0669:
0670:    END SUBROUTINE renumber_adj_mtx
0671: !
```

```
0672:
0673: !*****************************************************************************************
0674:    SUBROUTINE mk_RCM_table ( tree, n2o )
0675:
0676:       ! Reverse Cuthill-Mckee (RCM) ordering により頂点番号付替えのテーブルを作る(MaKe).
0677:       ! 変換テーブル n2o(1:nDOF_p) は 付替え後(new) から付替え前(org)を与える.
0678:       ! (反復法を適用するとき, データの局在性を向上させる. METIS内蔵のPARDISOなどでは不要)
0679:
0680:       !     参考文献：http://en.wikipedia.org/wiki/Cuthill%E2%80%93McKee_algorithm
0681:
0682:       TYPE(tree_c), INTENT(in   ) :: tree(:) !(nvtx)
0683:       INTEGER,      INTENT(out  ) :: n2o (:) !(nvtx) 付替え後の頂点番号から前への変換テーブル
0684:
0685:       ! 説明：RCM法の基本的な考え方はCuthill-Mckee(CM)法にある. そこでCM法について説明する.
0686:       !       CM法は幅優先探索(Breadth First Search)の一種であり, 頂点番号の付替えには
0687:       !       First In, First Out であるキュー (queue, 待ち行列) を用いる. 新しいデータは
0688:       !       キューn2o(:)の末尾(tail)に順に追加される. キューに入れられて処理待ちのデータ
0689:       !       (前線) は, キューの先頭(head)から順に処理される. キューを実現するための配列
0690:       !       n2o(:) 上に残されたデータ並びが, CM法による頂点番号付替えの結果となる.
0691:       !                              head -->          tail -->
0692:       !                            __v__ __ __ __ __ __v__ __ __      ← 付替え後の頂点番号
0693:       !     n2o(:) = |__|__|__|__|_i|_x|_x|_j|_x|_x|__|__|__|    ←   〃   前      〃
0694:       !                      処理済          処理待ち     ^次のデータが入る位置
0695:       !
0696:       !        5 f_____e(3)       本CMアルゴリズムをグラフと共に示すと, 以下のようになる.
0697:       !          /(2)   |\        0. 頂点c の次数が最小ゆえ, ここから訪問, キューの末尾に追記.
0698:       !       枝/       | \       1. キューの先頭にある頂点c に注目し, この全隣接頂点を訪問.
0699:       !    3 a/____4_g|___\d          初訪問の頂点b を, 次数の昇順にてキューの末尾に追記.
0700:       !    (3)\       /(4)  (2)  2. キューの先頭にある頂点b に注目し, この全隣接頂点を訪問.
0701:       !        \     /              初訪問の頂点a,g を, 次数の昇順にてキューの末尾に追記.
0702:       !         2`b/____c 1       3. キューの先頭にある頂点a に注目し, この全隣接頂点を訪問.
0703:       !           (3)   (1)          初訪問の頂点f を, …       ― 以下同様
0704:       !     頂点の( )内は次数
0705:       !
0706:       !       RCM法は, CM法による付替えの結果を逆順にするだけである.
0707:
0708:       INTEGER :: nvtx
0709:       INTEGER, ALLOCATABLE :: buf(:)      ! 第i行の隣接リストを木から取出すためのバッファ
0710:       INTEGER, ALLOCATABLE :: deg1(:)     !                   にある頂点の (次数 +1)
0711:       INTEGER :: deg1_i                   !         〃        長さ = 木の頂点数 = (次数 +1)
0712:       INTEGER :: max_deg1                 ! 最大の 隣接リスト長さ (バッファ割付け用)
0713:       INTEGER :: min_deg1, min_loc        ! 最小次数の頂点を Cuthill-Mckee の開始点とする
0714:       LOGICAL, ALLOCATABLE :: visited(:)  ! (nvtx) 頂点の訪問済フラグ
0715:       INTEGER :: head, tail               ! キューの先頭位置と末尾位置(次に入る位置)
0716:       INTEGER :: i, j                     ! 頂点番号 (隣接行列の i行 j列 )
0717:       INTEGER :: q, cnt
0718:
0719:       nvtx = SIZE(tree)
0720:
0721:       ! 簡便な付替え開始点として, 最小次数の頂点 min_loc を探す.
0722:       ! (ついでに, 隣接リストを取出すためのbuf(:)と, 対応する (次数+1)のdeg1(:) を準備)
0723:       max_deg1 = 0
0724:       min_deg1 = HUGE(min_deg1)
0725:       DO i = 1, nvtx
0726:          deg1_i = get_ndat( tree(i) )
0727:          IF ( max_deg1 < deg1_i ) max_deg1 = deg1_i
0728:          IF ( min_deg1 > deg1_i ) THEN
0729:             min_deg1 = deg1_i
0730:             min_loc  = i
0731:          END IF
0732:       END DO
0733:       ALLOCATE( buf (max_deg1), &
0734:               & deg1(max_deg1)  )
0735:
0736:       ! 幅優先探索によりグラフ全頂点を訪問していく
0737:       ALLOCATE( visited(nvtx) )
0738:       visited(:) = .FALSE.                ! 全頂点に未訪問のフラグを立てる
0739:       j          = min_loc                ! 付替え開始点から訪問開始
0740:       visited(j) = .TRUE.                 ! 「訪問済み」フラグを立てて,
0741:       tail = 1
0742:       n2o(tail)  = j                      ! キューの末尾に入れる
0743:       tail = tail +1
0744: !
```

```
0745:
0746:       DO head = 1, nvtx                    ! 幅優先探索
0747:
0748:          ! 以下の条件に該当するのは, 非連結グラフ（領域が分離）の場合のみ
0749:          IF ( head >= tail ) THEN          ! 連結グラフでは head = nvtx +1 で初合致
0750:             min_deg1 = HUGE(min_deg1)
0751:             DO i = 1, nvtx
0752:                IF ( .NOT. visited(i) ) THEN     ! 未訪問の頂点から再開始点を選ぶ
0753:                   deg1_i = get_ndat( tree(i) )
0754:                   IF ( min_deg1 < deg1_i ) THEN
0755:                      min_deg1 = deg1_i
0756:                      min_loc  = i
0757:                   END IF
0758:                END IF
0759:             END DO
0760:             j          = min_loc              ! 頂点 j を初めて訪問.
0761:             visited(j) = .TRUE.               ! 「訪問済み」フラグを立てて,
0762:             n2o(tail)  = j                    ! キューの末尾に入れる
0763:             tail = tail +1
0764:          END IF
0765:
0766:          i = n2o(head)                        ! キューの先頭にある頂点 i に注目
0767:          CALL get_dat( tree(i), buf, deg1_i )
0768:          cnt = 0                              ! 初訪問の頂点をカウント
0769:          DO q = 1, deg1_i
0770:             j = buf(q)
0771:             IF ( .NOT. visited(j) ) THEN      ! 頂点 j への訪問が初めてならば,
0772:                visited(j) = .TRUE.            ! 「訪問済み」フラグを立てて,
0773:                cnt = cnt +1
0774:                buf (cnt) = j                  ! 初訪問の頂点だけを抜き出す
0775:                deg1(cnt) = get_ndat( tree(j) )  ! その頂点の次数
0776:             END IF
0777:          END DO
0778:          ! 初訪問の隣接頂点 buf(1:cnt) を, deg1(1:cnt) をキーとして昇順ソート
0779:          CALL sort ( buf(1:cnt), deg1(1:cnt) )  ! 優先順位づけされた幅優先探索
0780:          n2o(tail:tail+cnt-1) = buf(1:cnt)      ! キューの末尾に入れる
0781:          tail = tail +cnt
0782:
0783:       END DO
0784:       DEALLOCATE (visited, deg1, buf )
0785:
0786:       ! RCM法による頂点番号の付替えは, CMによるものを逆順とすることで得られる.
0787:       CALL flip ( n2o )
0788:
0789:    CONTAINS
0790:
0791:       SUBROUTINE flip ( a )
0792:          ! a(1:n) = a(n:1:-1) を一時配列を作ることなく行う.
0793:          ! 理由) 上記の書き方だと, 大抵は一時配列を作成するので, n が大きくなると
0794:          !       スタックオーバーフローが発生する. 本ルーチンはこれを回避する.
0795:          INTEGER, INTENT(inout) :: a(:)
0796:          INTEGER :: n, i, j, tmp
0797:          n = SIZE(a)
0798:          DO i = 1, n/2
0799:             j = n+1 -i
0800:             tmp  = a(i)
0801:             a(i) = a(j)
0802:             a(j) = tmp
0803:          END DO
0804:       END SUBROUTINE flip
0805:
0806:    END SUBROUTINE mk_RCM_table
0807: !
```

```
0808:
0809: !*************************************************************************************************
0810:    SUBROUTINE do_coloring_by_Welsh_Powell ( this, vtx2col, ncol_ )
0811:
0812:       ! Welsh-Powell のアルゴリズムにより, グラフの頂点(VerTeX)を彩色(COLoring)する.
0813:
0814:       ! 用途：並列化のために有限要素集合をマルチカラー化
0815:       !   この用途において考えるグラフは, 疎行列を表わすためのものでないことに注意.
0816:       !   すなわち, 隣接行列は要素を介した全体自由度間の接続関係でなく, その逆関係
0817:       !    ― 全体自由度を介した要素間の接続関係 ― を表現するために用いる.
0818:       !   （したがって, 概念の混乱を避けるため, 疎行列ではあえて具体名で DOF(自由度)
0819:       !     としていたものを, 本ルーチンでは本来の用語 vtx(頂点) を用いる）
0820:
0821:       ! 説明：Welsh-Powell のアルゴリズムでは, 頂点v_p の次数を deg(v_p) として,
0822:       !             deg(v_1) >= deg(v_2) >= ....
0823:       !       と降順に並んでいるとき,  彩色に使用する色数 ncol_ の上限は
0824:       !             ncol_ <= max_k{ min(k,deg(v_k)+1) }
0825:       !       でおさえられる.                                          |
0826:       !                                                                |  ¥     k
0827:       !       例）3つの四角形要素                                      |_ ¥.／
0828:       !            _____ _____      max_k{ min(k,deg(k)+1) } = 2   ncol_|  ／`¥..deg(v_p)+1
0829:       !          |_1_|_2_|_1_|     deg(1)+1 = 3                       |／_____ `_______ k
0830:
0831:       ! 参考文献：Welsh, D.J.A., Powell, M.B.(1967),
0832:       !                  "An upper bound for the chromatic number of a graph
0833:       !                   and its application to timetabling problems",
0834:       !                   The Computer Journal 10 (1): 85-86
0835:       ! 参考文献： 船曳信生ら, グラフ理論の基礎と応用, 共立出版, 2012
0836:
0837:       TYPE(adj_mtx_c),    INTENT(in   ) :: this
0838:       INTEGER,            INTENT(out  ) :: vtx2col(:) ! (nvtx) 頂点から色への変換テーブル
0839:       INTEGER, OPTIONAL,  INTENT(out  ) :: ncol_      ! 使用した色数
0840:
0841:       INTEGER                :: nvtx        ! 全頂点数 (=最大の頂点番号) 〔全要素数〕
0842:       INTEGER, ALLOCATABLE   :: vtx (:)     ! (nvtx)  頂点番号〔要素番号〕
0843:       INTEGER, ALLOCATABLE   :: deg1(:)     ! (nvtx)  頂点の次数+1
0844:       INTEGER, ALLOCATABLE   :: buf(:)      ! 第i行の隣接リストを木から取出すためのバッファ
0845:       INTEGER                :: deg1_i      !       〃          長さ = 木の頂点数 = (次数 +1)
0846:       INTEGER                :: max_deg1    ! 最大の 隣接リスト長さ (バッファ割付け用)
0847:
0848:       LOGICAL, ALLOCATABLE   :: used(:)     ! (0:max_deg1) 色が隣接頂点で使用済みを表すフラグ
0849:       INTEGER                :: i, j        ! 頂点番号 (隣接行列の i行 j列)
0850:       INTEGER                :: k, q
0851:       INTEGER                :: ncol, icol  ! 使用色数, 色識別番号
0852:
0853:       IF ( .NOT. ALLOCATED(this%tree) ) STOP "do_coloring: グラフ彩色は組立てモードのみ"
0854:
0855:       nvtx = SIZE(this%tree)
0856:
0857:       ! 1) 頂点vtx(:)を, deg1(:)をキーとして昇順ソート
0858:       ALLOCATE( vtx (nvtx), &
0859:               & deg1(nvtx)  )
0860:       DO i = 1, nvtx
0861:          vtx (i) = i
0862:          deg1(i) = get_ndat( this%tree(i) )
0863:       END DO
0864:       CALL sort( vtx, deg1 )              ! vtx(:) を キーのdeg1(:)順で昇順ソート
0865: !
```

```
0866:
0867:       ! 2) 彩色は最大次数の頂点から降順で
0868:       max_deg1 = deg1(nvtx)                ! deg1(:) も昇順ソート済みゆえ, 最後が最大長
0869:       ALLOCATE( buf (  max_deg1), &
0870:              & used(0:max_deg1)  )         ! 色識別番号 0 を未着色に（プログラムの効率化）
0871:       used(:) = .FALSE.                    ! 全色を未使用に
0872:       vtx2col(:) = 0                       ! 未着色番号 0 を未訪問フラグ（番兵）に
0873:       ncol = 1
0874:       DO k = nvtx, 1, -1                   ! 次数の降順で回していく
0875:          i = vtx(k)                        ! i は頂点番号
0876:          CALL get_dat( this%tree(i), buf, deg1_i ) ! 頂点iの隣接リストから隣接頂点を取出す
0877:
0878:          ! 頂点iの隣接頂点で使用されている色を調査
0879:          used(1:ncol) = .FALSE.            ! フラグは "全色未使用"で初期化
0880:          DO q = 1, deg1_i                  ! j は頂点 i に 隣接する頂点の番号
0881:             j = buf(q)                     !   ( 隣接行列の(i,j)成分 )
0882:             icol = vtx2col(j)              ! icol = 0 の場合, 頂点 j は未着色（未訪問）だが,
0883:             used(icol) = .TRUE.            ! 条件分岐よりも used(0) に突っ込んで無処理で
0884:          END DO                            ! 終わらせるほうが楽.
0885:
0886:          ! 未使用色のうち, 最小の識別番号の色を探索
0887:          DO icol = 1, ncol
0888:             IF ( .NOT. used(icol) ) EXIT
0889:          END DO                            ! ループ完遂後は 制御変数icol == ncol+1
0890:          vtx2col(i) = icol
0891:
0892:          ! 上記で used(1:ncol) = .FALSE. として, 少し効率化するための付随処理
0893:          IF ( ncol < icol ) THEN           ! ここに該当すると全色使用済みゆえ色数を1増やす
0894:             ncol = icol
0895:             IF ( ncol > max_deg1 ) STOP "do_coloring: グラフ彩色のアルゴリズム上有り得ない"
0896:          END IF
0897:
0898:       END DO
0899:
0900:       IF ( PRESENT(ncol_) ) ncol_ = ncol
0901:
0902:    END SUBROUTINE do_coloring_by_Welsh_Powell
0903:
0904: END MODULE adj_mtx_class
```