FEMのデータ構造*

梅谷　信行

平成23年4月9日

目 次

*これは忘れっぽい著者が昔勉強したことを書き留めておく備忘録です．きっと間違えを多く含んでいます．すいません．ご指摘ご意見などありましたら教えてもらえると有り難いです．

1 概要

有限要素法では解析領域を単純な形状である要素に分割 (メッシュ分割) して、要素の中で積分を計算し、連立一次方程式を作り、それを解くことで解を求める。基本的に解析領域の単純な形への分割は、ある要素がどの節点を持っているかという配列とある節点の座標を持っている配列によって表現さる。これらをメッシュデータと呼ぶことにする。

しかしながら、連立一次方程式の作成やメッシュの可視化では、メッシュデータは直接使うことは少なく、別のデータに変換することで、初めて使えるようになる。例えば、連立一次方程式の場合、要素によってどの点とどの点が関係づけられるかという、節点から節点へのデータを持つ必要があるが、これはメッシュデータを変換しなければ得られない。

ここでは、要素の型がすべて同じ場合（単一メッシュ）について例を挙げる。

簡単のため２次節点は考慮しないで、節点は要素の角上にのみ存在すると考える。また、幾何学的な側面から有限要素法の節点を点と呼ぶことがあるが、ここでは両者は大きな違いはない。

2 メッシュデータ

FEM ではある要素がどの節点を持つかという配列を point in element を省略して *inpoel* と呼ぶことにする（有限要素法の教科書では大抵 lnods と呼ばれる）例えば

要素 ielem の中の要素内節点番号 ipoel の節点番号 ipoin は次のように配列 *inpoel* から取り出すことができる。

ipoin=*inpoel*[ielem][ipoel]

総要素数を nelem、要素内節点の数を npoel とすると、この配列 *inpoel* の大きさは nelem × npoel である。

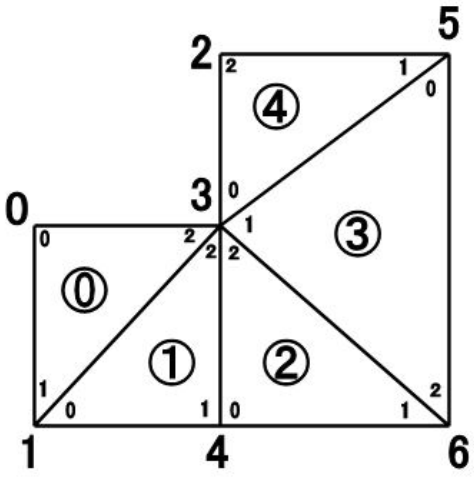


図 1: メッシュデータの例, ◯で囲まれた数字は要素番号，それ以外の小さな数字は要素内節点番号，大きな数字は節点番号

たとえば、図 1 のようなメッシュの場合（但し、◯で囲まれた数字は要素番号であり、小さい文字で表している数字は要素内節点番号である。）

inpoel[3][0] = 5、*inpoel*[3][1] = 3、*inpoel*[3][2] = 6

のようになる。

節点の座標を格納する配列を *coords* と呼ぶ。節点 ipoin の idim 座標は次のように書ける

coords[ipoin][idim]

総節点数を npoin、次元数を ndim とすると、この配列 *coords* の大きさは npoin × ndim である。

例えば、ndim=2 の場合、節点 ipoin の x 座標 x、 y 座標 y は次のように書ける

x = *coords*[ipoin][0]
y = *coords*[ipoin][1]

上で述べたメッシュデータからいかに派生型のデータを作成するかという点について述べる。

3　点周りの要素

ある要素がどの点をもっているかという情報は配列 *inpoel* が持っている。反対にある点がどの要素に属しているかという情報を持っていれば、後に述べる要素周りの要素や点周りの点を調べる時などに非常に役に立つ。次のようなメッシュの場合は点周りの要素は次のようになる。

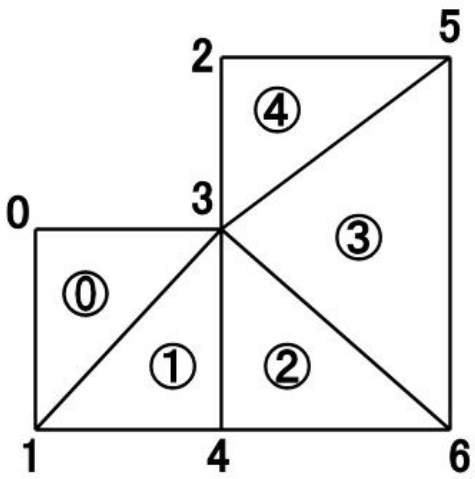


図 2: メッシュデータの例, ◯で囲まれた数字は要素番号，それ以外は節点番号

3.1　点周りの要素のデータ構造

1 つの点が属する要素の集合を Element Surrounding Point を略して *elsup* という配列に格納する。

点番号	この点を囲む要素番号
0	0
1	0,1
2	4
3	0,1,2,3,4
4	1,2
5	3,4
6	2,3

一つの要素がいくつの要素に囲まれているかというのは節点によって異なるために、Index 配列配列を利用する。Index 配列とはある節点の周りの要素が格納されている *elsup* の場所の先頭を格納している配列で、この Index 配列を *elsup_ind* とすると、節点 ipoin に囲まれている要素の番号 ielem0 は

ielem0 = *elsup*[ielsup]　　(ielsup=*elsup_ind*[ipoin] 〜 *elsup_ind*[ipoin+1]-1)

によって与えられる。
例えば図 2 ようなメッシュの場合は

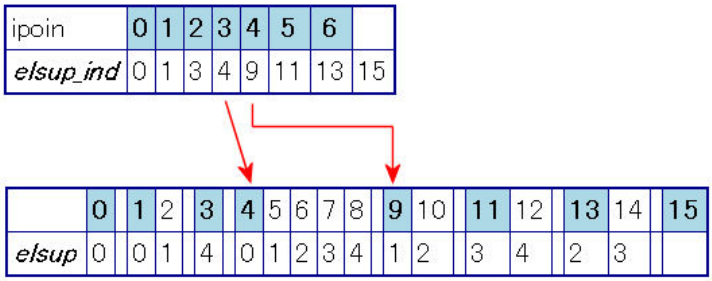
nelsup = 15
elsup_ind = {0, 1, 3, 4, 9, 11, 13, 15 }
elsup = { 0, 0, 1, 4, 0, 1, 2, 3, 4, 1, 2, 3, 4, 2, 3 }

のようになる。
例えば、節点番号 3 の周りの要素番号が知りたければ *elsup_ind* [3]=4、*elsup_ind* [3+1]=9 より
配列 *elsup* の 4 番から 9-1=8 番を見ればよい。*elsup* の 4 番から 8 番には 0,1,2,3,4 が入っているはずである。

3.2　点周りの要素の作り方

ここでは配列 *elsup_ind*、*elsup* と値 nelsup を具体的に求める方法を説明する。
これらのデータは次のようなステップを用いて生成することができる。

1. Step1:節点がいくつの要素に囲まれているかを *elsup_ind* に格納
2. Step2:*elsup_ind* を作る



3. Step3:nelsup を求め、*elsup* のメモリを確保
4. Step4:*elsup_ind* を変更しながら、*elsup* を作る
5. Step5:*elsup_ind* を元に戻す

3.2.1 Step1:節点がいくつの要素に囲まれているかを *elsup_ind* に格納

領域を確保し、初期化した後に、点がいくつの要素に囲まれているかを *elsup_ind* に格納する

ここで注意するべき点は点 ipoin を囲む要素の数を *elsup_ind* [ipoin+1] に格納していることである。

```
elsup_ind = new int [npoin+1];
for(int ipoin=0;ipoin<npoin+1;ipoin++){
   elsup_ind[ipoin] = 0;
}
for(int ielem=0;ielem<nelem;ielem++){
   for(int ipoel=0;ipoel<npoel;ipoel++){
      int ipoin0 = inpoel[ielem][ipoel];
      elsup_ind[ipoin0+1]++;
   }
}
```

3.2.2 Step2:*elsup_ind* を作る

Step1 の *elsup_ind* を前から順番に足しあわせていくことで *elsup_ind* を作っている。

```
for(int ipoin=0;ipoin<npoin;ipoin++){
   elsup_ind[ipoin+1] += elsup_ind[ipoin];
}
```

3.2.3 Step3:nelsup を求め、*elsup* のメモリを確保

```
nelsup = elsup_ind[npoin];
elsup = new int [nelsup];
```

3.2.4 Step4:*elsup_ind* を変更しながら、*elsup* を作る

もともと *elsup_ind* は *elsup* の Index を格納する配列である。

つまり、ipoin を囲む要素番号は配列 *elsup* の *elsup_ind* [ipoin] 番目から始まるというように、配列 *elsup_ind* は先頭を指し示す配列であった。しかし、ここでは先頭を指し示す配列としてではなく、*elsup* のどこに値をいれるかという配列として用いる。一度値を入れると *elsup_ind* の値を 1 つ増やすことで上書きしないようにしている。この操作によって *elsup_ind* を変更していくと、最終的に *elsup_ind* の値は前に 1 つずれることになる。

```
for(int ielem=0;ielem<nelem;ielem++){
   for(int ipoel=0;ipoel<npoel;ipoel++){
      int ipoin0 = inpoel[ielem][ipoel];
      int ielsup0 = elsup_ind[ipoin0];
      elsup[ielsup0] = ielem;
      elsup_ind[ipoin0]++;
   }
}
```

3.2.5 Step5:*elsup_ind* を元に戻す

elsup_ind を元に戻す。*Step4* によって配列 *elsup_ind* の値は 1 つ前にずれていることになる。そこでこれを元通りに戻すために配列の値を後ろにずらす。配列の 0 番目には 0 が入っていることがわかるので最後にこれをセットする。

```
for(int ipoin=npoin;ipoin&gt;0;ipoin--){
   elsup_ind[ipoin] = elsup_ind[ipoin-1];
}
elsup_ind[0] = 0;
```

以上により点を囲む要素のデータを構築することができる。

4 点周りの点

点周りの点とは要素によってどの点とどの点が接続しているかという情報であり、有限要素法において連立 1 次方程式を作成する際に行列の非零成分の場所を

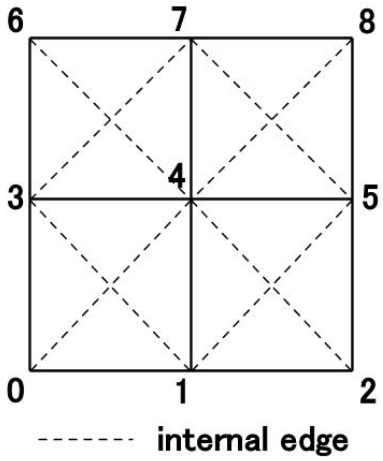


図 3: ４角形要素の内部辺の例

得る時に使う。これはあくまで要素によって関係づけられている節点のことで、要素の辺による接続とは一般的には異なっている。

例えば上のようなメッシュの場合、点０は要素の辺で接続されている点１、点３に加えて点４も要素によって接続されている。実際０と４の間には辺がないが、仮想的に辺があるとして要素の internal edge と言う。上の図では点４は自分以外の全ての節点と接続していることになる。三角形要素や四面体要素は interanal edge は持たず、すべての節点同士が辺で接続されている。

4.1 点周りの点のデータ構造

点がどの点と要素によって接続しているかという情報を Point Surrounding Point を略して *psup* という配列に格納する。

一つの点がいくつの点と接続しているかというのは節点によって異なるために、これもまた、データの先頭の場所を表す、Index 配列つきの 1 次元配列で表す。この Index 配列を *psup_ind* とすると節点 ipoin に接続している節点の番号 ipoin0 は

ipoin0 = *psup*[ipsup]　　(ipsup=*psup_ind* [ipoin] 〜 *psup_ind* [ipoin+1]-1)

によって与えられる。

4.2 点周りの点の作り方

ここでは配列 *psup_ind*、*psup* と値 npsup を具体的に求める方法を説明する。
これらのデータは次のようなステップを用いて生成することができる。

4.2.1 Step1

点と点との接続の数 npsup を求めるステップ
ある節点 ipoin が属する要素 jelem0 について、その要素の中の節点 jpoin0 が既にカウントされているかどうかを配列 *lpoin* を見ることで調べる。既にカウントされていたら *lpoin* [jpoin]=ipoin となっている。そうでなければカウンタ icoun0 を 1 つ増やし、この節点が２重にカウントしないために *lpoin* [jpoin0]=ipoin を代入する。

```
lpoin = new int [npoin];
for(int ipoin=0;ipoin<npoin;ipoin++){
   lpoin=-1;
}
int icoun0 = 0;
for(int ipoin=0;ipoin<npoin;ipoin++){
   lpoin[ipoin] = ipoin;
   for(int ielsup=elsup_ind[ipoin];ielsup<elsup_ind[ipoin+1];ielsup++){
      int jelem0 = elsup[ielsup];
      for(int ipoel=0;ipoel<npoel;ipoel++){
         int jpoin0 = inpoel[jelem][ipoel];
         if( lpoin[jpoin0] != ipoin ){
            icoun0++;
            lpoin[jpoin0] = ipoin;
         }
      }
   }
}
npsup = icoun0;
```

4.2.2 Step2

メモリを確保するステップ
Step1 で求めた npsup を元に配列 *psup* のメモリを確保する

```
psup_ind = new int [npoin+1];
psup = new int [npsup];
```

4.2.3 Step3

Step1 と同じ手順で *psup_ind*、*psup* を作るステップ

```
for(int ipoin=0;ipoin<npoin;ipoin++){
   lpoin=-1;
}
icoun0 = 0;
psup_ind[0] = 0;
for(int ipoin=0;ipoin<npoin;ipoin++){
   lpoin[ipoin] = ipoin;
   for(int ielsup=elsup_ind[ipoin];ielsup<elsup_ind[ipoin+1];ielsup++){
      int jelem0 = elsup[ielsup];
      for(int ipoel=0;ipoel<npoel;ipoel++){
         int jpoin0 = inpoel[jelem0][ipoel];
         if( lpoin[jpoin0] != ipoin ){
            psup[icoun0] = jpoin0;
            icoun0++;
            lpoin[jpoin0] = ipoin;
         }
      }
   }
   psup_ind[ipoin+1] = icoun0;
}
delete[] lpoin;
```

4.2.4 Step4

Step3 でもとめた psup のデータは昇順になっていないので、適当な関数を呼んでソートする。下はC＋＋の標準で用意されているソート関数を使った例である。下の関数を使うためには、STL アルゴリズムヘッダ *algorithm* をインクルードする必要がある。C 言語なら qsort 関数を使うのがよい。

```
for(int ipoin=0;ipoin<npoin;ipoin++){
   int ipsup_begin = psup_ind[ipoin];
   int ipsup_end = psup_ind[ipoin+1];
   std::sort( &psup[ipsup_begin], &psup[ipsup_end] );
}
```

5　要素周りの要素

5.1　要素周りの要素のデータ構造

要素がどの要素に接しているかという情報は Elemnt Surrounding Element を略して *elsuel* という配列に格納する。

一つの要素が接している要素の数は一定であり、要素の面（２次元の場合は辺）の数 nfael に等しい。よって次のように２次元配列として値を格納する。要素 ielem の面 ifael に接する要素 ielem0 は

ielem0=*elsuel*[ielem][ifael]

によって与えられる。外側に面しているために他の要素に接していないような要素がある場合は普通ではとりえない値（例えば-1）を返すようにしておく

5.2 要素の面の番号づけ

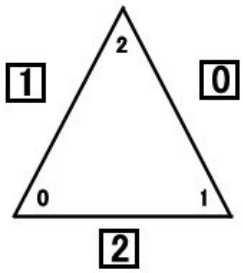
要素の面（２次元の場合は辺）の番号のつけ方は任意である。Point in Face を略して *lpofa* という配列に要素の面がどの節点を持っているかというデータをあらかじめ格納しておく

例えば要素の面 *ifael* の中の要素面内節点番号 ipofa が要素内節点番号 ipoel であるという情報は次のように配列 *lpofa* によって与えられる。

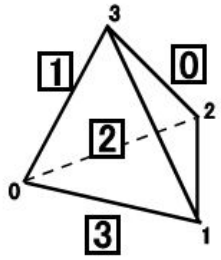
ipoel=*lpofa* [ifael][ipofa]

例えば次のように番号付けをする

5.2.1 三角形要素



面番号	面の中の点 0	面の中の点 1
0	lpofa[0][0]=1	lpofa[0][1]=2
1	lpofa[1][0]=2	lpofa[1][1]=0
2	lpofa[2][0]=0	lpofa[2][1]=1



面番号	面の中の点 0	面の中の点 1	面の中の点 2
0	lpofa[0][0]=1	lpofa[0][1]=2	lpofa[0][2]=3
1	lpofa[1][0]=0	lpofa[1][1]=3	lpofa[1][2]=2
2	lpofa[2][0]=0	lpofa[2][1]=1	lpofa[2][2]=3
3	lpofa[3][0]=0	lpofa[3][1]=2	lpofa[3][2]=1

5.2.2 ４面体要素

図 2 のメッシュを考える。但し、要素番号は○で囲まれており、要素内節点番号は小さい数字になっている。それ以外の数字は節点番号である。

この場合配列 *elsuel* は次のようになる。

$$elsuel\ [3][0] = 2,\quad elsuel\ [3][1] = -1,\quad elsuel\ [3][2] = 4$$

5.3 要素周りの要素の作り方

ここでは要素 ielem の面 ifael に接する要素、つまり、*elsuel*[ielem][ifael] を具体的に求める方法を説明する。*elsuel* は全ての要素の中の全ての面についてループさせてこれを調べることで求めることができる。

elsuel[ielem][ifael] は次のようなステップを用いて生成することができる。

1. Step1:大きさ npofa の配列 *inpofa* を確保する。また、大きさ npoin の配列 *lpoin* を確保して 0 でクリアする。

2. Step2:要素 ielem の面 ifael 上の節点を *inpofa* に格納して、その節点について *lpoin* を 1 にする。また、要素の面上のある 1 点を ipoin0 とする

3. Step3:ipoin0 の周りの要素要素 jelem0 の面 jfael0 上の点が全て要素 ielem の面 ifael に含まれるか *lpoin* を使って調べる。全て含まれていれば *elsuel*[ielem][ifael]

に jelem0 を代入して終了、一致していなければ新しい要素の面について調べる。

4. Step4:*lpoin* を元に戻す

Step2 から Step4 の手順を全ての要素 ielem=0〜nelem、ifael=0〜nfael の間を繰り返すことで全要素の全ての面に対して隣り合う要素を見つけることができる。
以下各ステップについて細かく説明する。

5.3.1 Step1

大きさ npofa の配列 *inpofa* を確保する。また、大きさ npoin の配列 *lpoin* を確保して 0 でクリアする。

```
inpofa = new int [npofa];
lpoin = new int [npoin];
for(int ipoin=0;ipoin<npoin;ipoin++){
   lpoin[ipoin] = 0;
}
```

5.3.2 Step2

要素 ielem の面 ifael 上の節点を *inpofa* に格納して、その節点について *lpoin* を 1 にする。また、要素の面上のある 1 点を ipoin0 とする

```
for(int ipofa=0;ipofa<npofa;ipofa++){
  int ipoi0 = inpoel[ielem][ lpofa[ifael][ipofa] ];
  inpofa[ipofa] = ipoi0;
  lpoin[ipoin0] = 1;
}
```

5.3.3 Step3

ipoin0 の周りの要素要素 jelem0 の面 jfael0 上の点が全て要素 ielem の面 ifael に含まれるか *lpoin* を使って調べる。全て含まれていれば *elsuel* [ielem][ifael] に jelem0 を代入して終了、一致していなければ新しい要素の面について調べる。

```
ipoin0 = inpofa[0];
bool iflg0 = false;
for(int ielsup=elsup_ind[ipoin0];ielsup<elsup_ind[ipoin0+1];ielsup++){
  int jelem0 = elsup[ielsup];
  if( ielem == jelem0 ) continue;
  for(int jfael=0;jfael<nfael;jfael++){
     iflg0 = true;
     for(int ipofa=0;ipofa<npofa;ipofa++){
```

```
        int jpoin0 = inpoel[jelem0][ lpofa[jfael][jpofa] ];
        if( lpoin[jpoin0] == 0 ){
           iflg0 = false;
           break;
        }
     }
     if( iflg0 ){
        elsuel[jelem0][jfael] = ielem;
        break;
     }
  }
  if( iflag0 ) break;
}
```

5.3.4 Step4

lpoin を元に戻す

```
for(int ipofa=0;ipofa<npofa;ipofa++){
   lpoin[ inpofa[ipofa] ] = 0;
}
```

6 参考にしたもの

[1]

7 参考文献

参考文献

[1] Löhner, P. R.: *Applied Computational Fluid Dynamics Techniques: An IntroductionBased on Finite Element Methods*, Wiley, 2 edition (2008).