

```

1  /*****!*****!
2  // Program to solve 2D convection-diffusion problem
3  // using DRP-SUPG finite element model
4  /*****!*****!
5  // Author : Neo Shih-Chao Kao
6  /*****!*****!
7
8  #include <iostream>
9  #include <fstream>
10 #include <iomanip>
11 #include <cmath>
12
13 using namespace std ;
14
15 #include "FEM_variables.h" // FEM2D header file
16 #include "FEM_DRP_SUPG.cpp" // DRP_SUPG model
17 #include "AXB_solver.cpp" // linear system solver
18 #include "FEM_tools.cpp" // including files
19
20 int main(){ // Finite element code to solve CD equation
21
22     /*****!*****!
23
24     // First open the file Idata to read some mesh parameter and diffusivity
25     ifstream Finput("Idata.dat") ;
26     Finput >> melem ;
27     Finput >> mpoin ;
28     Finput >> mtotv ;
29     Finput >> kappa ;
30     Finput.close() ;
31
32     /*****!*****!
33
34     // Allocate the meomry
35
36     nadfm = new int[mpoin] ;
37     nodfm = new int[mpoin] ;
38
39     lboud = new int[mpoin] ;
40     reabc = new int[mpoin] ;
41     bcnod = new int[melem*nnodp] ;
42     boudv = new double[mtotv] ;
43
44     cordx = new double[mpoin] ;
45     cordy = new double[mpoin] ;
46
47     uvely = new double[mpoin] ;
48     vvely = new double[mpoin] ;
49
50     eqphi = new double[mtotv] ;
51     egrhs = new double[mtotv] ;
52     matmx = new double[melem*msize] ;
53
54     lnods = new int[melem*nnodp] ;
55
56     shad1 = new double[melem*360] ;
57     shad2 = new double[melem*162] ;
58
59     // Setting all data for finite element calculation
60
61     Setting_data( melem , mpoin , mtotv , kappa , cordx ,
62                  cordy , uvely , vvely , lnods , bcnod ,
63                  reabc , nadfm , nodfm , shad1 , shad2 ,
64                  lboud , boudv ) ;
65
66     // Solve the convection diffusion equation
67
68     Solve( melem , mpoin , mtotv , cordx , cordy ,
69           uvely , vvely , kappa , lnods , nadfm ,
70           nodfm , bcnod , lboud , boudv , shad1 ,
71           shad2 , matmx , eqphi , egrhs ) ;
72
73     // Output the mesh file
74
75     Tec360_CDR( melem , mpoin , cordx , cordy , eqphi , lnods ) ;
76
77     // pause the program
78
79     system("pause");
80
81 }
82
83
84
85
86

```

```
1
2
3  /*
4  *   The header file to define the variables used in
5  *   the finite element code
6  */
7
8  #include <iostream>
9  #include <fstream>
10 #include <iomanip>
11 #include <cmath>
12
13 int i , j ;
14
15 int nnodp = 9 ;
16 int nnodl = 4 ;
17 int nevab = 9 ;
18 int msize = 81 ;
19
20 int melem ;
21 int mpoin ;
22 int mtotv ;
23
24 int *lnods ;
25 int *bcnod ;
26
27 // The DOF of node
28 int *nadm ;
29 int *nodfm ;
30
31 // Boundary condition
32
33 int *reabc ;
34 int *lboud ;
35 double *boudv ;
36
37 double kappa ;
38
39 double *cordx ; // x-directional coordinate
40 double *cordy ; // y-directional coordinate
41 double *uvel ; // x-directional velocity vector
42 double *vvel ; // y-directional velocity vector
43
44 double *eqphi ; // solution vector
45 double *eqrhs ; // right hand side vector
46
47 double *shad1 ; // store the interpolation and its 1st derivative function
48 double *shad2 ; // store the 2nd derivative interpolation function
49
50 double *matmx ;
51
52 int isize = sizeof(int ) ;
53 int dsize = sizeof(double) ;
54
55 //-----
56
57
```

```

1
2  /*
3  *   FEM_DRP_SUPG.cpp
4  */
5
6  double ABS(double x){
7      if( x >= 0.0 ) return x ;
8      if( x < 0.0 ) return -x ;
9  }
10
11 void Tau2D_SUPG( double locau , double locav , double eledx ,
12                 double eledy , double *artdf ){
13
14     //-----
15     // Routine to get the stabilized parameter tau in
16     // DRP-SUPG model
17     //-----
18     //
19     // 7-----6-----5
20     // |         |         |
21     // |         |         |
22     // 8-----9-----4
23     // |         |         |
24     // |         |         |
25     // 1-----2-----3
26     //
27     //-----
28
29     double alph , beta ;
30     double alphax , betax ;
31     double alphay , betay ;
32
33     double vmagn ; // The square of the magnitude of velocity
34
35     double pi = 3.14159265359 ;
36
37     vmagn = locau*locav+locav*locav ;
38
39     if ( ABS(locav) > 1e-15 ){
40         alph = (8.0-3.0*pi)/(-22.0+6.0*pi) ;
41         beta = 0.5 ;
42         //-----
43         alphax = (alph*locav*eledx)/(2.0*vmagn) ;
44         betax = (beta*locav*eledx)/(2.0*vmagn) ;
45     }
46
47     if ( ABS(locav) > 1e-15 ){
48         alph = (8.0-3.0*pi)/(-22.0+6.0*pi) ;
49         beta = 0.5 ;
50         //-----
51         alphay = (alph*locav*eledy)/(2.0*vmagn) ;
52         betay = (beta*locav*eledy)/(2.0*vmagn) ;
53     }
54
55     artdf[0] = alphax + alphay ; // 1
56     artdf[2] = alphax + alphay ; // 3
57     artdf[4] = alphax + alphay ; // 5
58     artdf[6] = alphax + alphay ; // 7
59
60     artdf[1] = betax + alphay ; // 2
61     artdf[5] = betax + alphay ; // 6
62
63     artdf[3] = alphax + betay ; // 4
64     artdf[7] = alphax + betay ; // 8
65
66     artdf[8] = betax + betay ; // 9
67
68 }

```

```

1 void Tec360_mesh( int      melem , int mpoin , double cordx[] ,
2                   double cordy[] , int lnods[] ){
3
4
5     int ipoin , jpoint , ielem ;
6     int nnodp = 9 ;
7
8     /*****
9     /* Set the file head title
10    /*****
11
12    ofstream Oufile("Mesh2D.plt");
13
14    Oufile.setf(ios::fixed) ;
15    Oufile.precision(15) ;
16
17    /*****
18    Oufile << " Variables = X , Y " << endl;
19    Oufile << " Zone N = " << mpoin << " E = " << 4 * melem << endl;
20    Oufile << " Datapacking=point, Zonetype=fequadrilateral" << endl;
21    /*****
22
23    for ( ipoin = 0 ; ipoin < mpoin ; ipoin++ ){
24        Oufile << cordx[ipoin] << " " << cordy[ipoin] << endl ;
25    }
26
27    for ( ielem = 0 ; ielem < melem ; ielem++ ){
28
29        Oufile << lnods[ielem*nnodp+0] << " " << lnods[ielem*nnodp+1] << " "
30        << lnods[ielem*nnodp+8] << " " << lnods[ielem*nnodp+7] << endl;
31
32        Oufile << lnods[ielem*nnodp+1] << " " << lnods[ielem*nnodp+2] << " "
33        << lnods[ielem*nnodp+3] << " " << lnods[ielem*nnodp+8] << endl;
34
35        Oufile << lnods[ielem*nnodp+8] << " " << lnods[ielem*nnodp+3] << " "
36        << lnods[ielem*nnodp+4] << " " << lnods[ielem*nnodp+5] << endl;
37
38        Oufile << lnods[ielem*nnodp+7] << " " << lnods[ielem*nnodp+8] << " "
39        << lnods[ielem*nnodp+5] << " " << lnods[ielem*nnodp+6] << endl;
40    }
41    Oufile.close();
42 }
43
44 void Tec360_CDR( int      melem , int mpoin, double cordx[] , double cordy[] ,
45                 double eqphi[] , int lnods[] ){
46
47     int ipoin , jpoint , ielem ;
48     int nnodp = 9 ;
49
50     /*****
51     /* Set the file head title
52     /*****
53
54     ofstream Oufile("CDR2D_FEM.plt");
55
56     Oufile.setf(ios::fixed) ;
57     Oufile.precision(8) ;
58
59     /*****
60     Oufile << " VARIABLES = X, Y, U " << endl;
61     Oufile << " ZONE N = " << mpoin << " E = " << 4 * melem << endl;
62     Oufile << " DATAPACKING=POINT, ZONETYPE=FEQUADRILATERAL" << endl;
63     /*****
64
65     for ( ipoin = 0 ; ipoin < mpoin ; ipoin++ ){
66         Oufile << cordx[ipoin] << " " << cordy[ipoin] << " "
67         << eqphi[ipoin] << endl ;
68     }
69
70     for ( ielem = 0 ; ielem < melem ; ielem++ ){
71
72         Oufile << lnods[ielem*nnodp+0] << " " << lnods[ielem*nnodp+1] << " "
73         << lnods[ielem*nnodp+8] << " " << lnods[ielem*nnodp+7] << endl;
74
75         Oufile << lnods[ielem*nnodp+1] << " " << lnods[ielem*nnodp+2] << " "
76         << lnods[ielem*nnodp+3] << " " << lnods[ielem*nnodp+8] << endl;
77
78         Oufile << lnods[ielem*nnodp+8] << " " << lnods[ielem*nnodp+3] << " "
79         << lnods[ielem*nnodp+4] << " " << lnods[ielem*nnodp+5] << endl;
80
81         Oufile << lnods[ielem*nnodp+7] << " " << lnods[ielem*nnodp+8] << " "
82         << lnods[ielem*nnodp+5] << " " << lnods[ielem*nnodp+6] << endl;
83     }
84     Oufile.close();
85 }

```

```

1  #include <iostream>
2  #include <fstream>
3  #include <iomanip>
4  #include <cmath>
5
6  using namespace std ;
7
8  int f2c( int f_index ){
9
10     //-----
11     // Routine to translate the index
12     //-----
13     // Note :
14     // In Fortran, the index starts from 1
15     // In C/C++ , the index starts from 0
16     //-----
17
18     int c_index ;
19     c_index = f_index-1 ;
20     return c_index ;
21 }
22
23
24 //+++++
25 //+++++
26
27 void Gauss( double *gpt , double *wet ){
28
29     //-----
30     // Routine to define the Gaussian point and
31     // weighted factors
32     //-----
33
34     gpt[0] = 0.7745966692 ;
35     gpt[1] = 0.0000000000 ;
36     gpt[2] = -0.7745966692 ;
37     //-----
38     wet[0] = 0.5555555556 ;
39     wet[1] = 0.8888888889 ;
40     wet[2] = 0.5555555556 ;
41 }
42
43 //+++++
44 //+++++
45
46 void shap9( double *phi , double *dndxi , double *dndyi ,
47            double xi , double eta ){
48
49     //-----
50     // Routine to define the biquadratic interpolation
51     // function, the order of interpolation is as follows
52     //-----
53     // phi : Interpolation function
54     // dndxi : dN/dxi
55     // dndyi : dN/deta
56     //-----
57     //
58     // 7-----6-----5
59     // |       |       |
60     // |       |       |
61     // 8-----9-----4
62     // |       |       |
63     // |       |       |
64     // 1-----2-----3
65     //
66     //-----
67
68     double x, y, xx, yy, xy, xxy, xyy, xxyy ;
69
70     x = xi ;
71     y = eta ;
72     xx = x * x ;
73     yy = y * y ;
74     xy = x * y ;
75     xxy = xx * y ;
76     xyy = x * yy ;
77     xxyy = xx * yy ;
78
79     // The biquadratic interpolation function
80
81     phi[0] = 0.25 * ( xy - xyy - xxy + xxyy ) ;
82     phi[1] = 0.50 * ( -y + yy + xxy - xxyy ) ;
83     phi[2] = 0.25 * ( -xy + xyy - xxy + xxyy ) ;
84     phi[3] = 0.50 * ( x + xx - xxy - xxyy ) ;
85     phi[4] = 0.25 * ( xy + xxy + xxy + xxyy ) ;
86     phi[5] = 0.50 * ( y + yy - xxy - xxyy ) ;
87     phi[6] = 0.25 * ( -xy - xyy + xxy + xxyy ) ;
88     phi[7] = 0.50 * ( -x + xx + xxy - xxyy ) ;
89     phi[8] = 1.00 -xx - yy + xxyy ;
90
91     // The biquadratic interpolation function w.r.t xi
92
93     dndxi[0] = (0.25*( y-yy)) + (0.5*(-xy+xyy)) ;
94     dndxi[1] = xy - xxy ;
95     dndxi[2] = (0.25*(-y+yy)) + (0.5*(-xy+xyy)) ;
96     dndxi[3] = (0.5*( 1.0-yy)) + x - xxy ;
97     dndxi[4] = (0.25*( y+yy)) + (0.5*( xy+xyy)) ;
98     dndxi[5] = -xy - xxy ;
99     dndxi[6] = (0.25*(-y-yy)) + (0.5*( xy+xyy)) ;
100    dndxi[7] = (0.5*(-1.0+yy)) + x - xxy ;
101    dndxi[8] = 2.0*(-x+xyy) ;
102
103     // The biquadratic interpolation function w.r.t eta
104
105     dndyi[0] = (0.25*( x-xx)) + (0.5*(-xy+xyy)) ;
106     dndyi[1] = (0.5*(-1.0+xx)) + y - xxy ;
107     dndyi[2] = (0.25*(-x-xx)) + (0.5*( xy+xyy)) ;
108     dndyi[3] = -xy - xxy ;
109     dndyi[4] = (0.25*( x+xx)) + (0.5*( xy+xyy)) ;
110     dndyi[5] = (0.5*( 1.0-xx)) + y - xxy ;
111     dndyi[6] = (0.25*(-x+xx)) + (0.5*(-xy+xyy)) ;
112     dndyi[7] = xy - xxy ;
113     dndyi[8] = 2.0*(-y+xyy) ;
114
115 } // end Shap9 subroutine

```

```

116
117 //+++++
118 //+++++
119
120 void shap4( double *phi , double *dndxi, double *dndyi,
121            double xi , double eta ){
122
123     //-----
124     // Routine to define the bilinear interpolation
125     // function, the order of interpolation is as follows
126     //-----
127     // phi : Interpolation function
128     // dndxi : dN/dxi
129     // dndyi : dN/deta
130     //-----
131     //
132     // 4-----+-----3
133     // |         |         |
134     // |         |         |
135     // +-----+-----+
136     // |         |         |
137     // |         |         |
138     // 1-----+-----2
139     //
140     //-----
141     double x, y, xy ;
142
143     x = xi ;
144     y = eta ;
145     xy = x * y ;
146
147     /***** shape function value *****/
148
149     phi[0] = 0.25 * ( 1.0 - x - y + xy ) ;
150     phi[1] = 0.25 * ( 1.0 + x - y - xy ) ;
151     phi[2] = 0.25 * ( 1.0 + x + y + xy ) ;
152     phi[3] = 0.25 * ( 1.0 - x + y - xy ) ;
153
154     /***** Derivative shape function w.r.t xi *****/
155
156     dndxi[0] = 0.25 * ( -1.0 + y ) ;
157     dndxi[1] = 0.25 * ( 1.0 - y ) ;
158     dndxi[2] = 0.25 * ( 1.0 + y ) ;
159     dndxi[3] = 0.25 * ( -1.0 - y ) ;
160
161     /***** Derivative shape function w.r.t eta *****/
162
163     dndyi[0] = 0.25 * ( -1.0 + x ) ;
164     dndyi[1] = 0.25 * ( -1.0 - x ) ;
165     dndyi[2] = 0.25 * ( 1.0 + x ) ;
166     dndyi[3] = 0.25 * ( 1.0 - x ) ;
167
168 }
169
170 //+++++
171 //+++++
172
173 double Jacob( double *dndxi , double *dndyi , double *elexx ,
174              double *eleyy , double *Jmat , double *Jinv ){
175
176     //-----
177     // Routine to compute
178     // 1. Jacobian matrix
179     // 2. The determinant of Jacobian matrix
180     // 3. The inverse of the Jacobian matrix
181     //-----
182
183     int ndime = 2 ;
184     int nnodp = 9 ;
185
186     int i , j ;
187
188     double Jdet ;
189
190     // Initialize the Jacobian and its inverse matrix
191
192     for ( i = 0 ; i < ndime*ndime ; i++){ // start i-loop
193         Jmat[i] = 0.0 ;
194         Jinv[i] = 0.0 ;
195     } // end i-loop
196
197     // Compute the Jacobian matrix
198
199     //-----
200     //      +-          -+
201     //      | Jmat(0),Jmat(1) |
202     // J =  |                 |
203     //      | Jmat(2),Jmat(3) |
204     //      +-          -+
205     //-----
206
207     for ( i = 0 ; i < nnodp ; i++){ // start i-loop
208         Jmat[0] += elexx[i] * dndxi[i] ;
209         Jmat[1] += eleyy[i] * dndxi[i] ;
210         Jmat[2] += elexx[i] * dndyi[i] ;
211         Jmat[3] += eleyy[i] * dndyi[i] ;
212     } // end i-loop
213
214     Jdet = Jmat[0] * Jmat[3] - Jmat[1] * Jmat[2] ;
215
216     // Compute the inverse of Jacobian matrix
217
218     Jinv[0] = Jmat[3] / Jdet ;
219     Jinv[3] = Jmat[0] / Jdet ;
220     Jinv[1] = -Jmat[1] / Jdet ;
221     Jinv[2] = -Jmat[2] / Jdet ;
222
223 } // end Jacob subroutine
224
225 //+++++
226 //+++++
227
228 double GeoTensor2( double *dndx2 , double *dndy2 , double *dndxy ,
229                   double *elexx , double *eleyy , double *Jmat2 ){
230

```

```

231     int i ;
232     int nnodp = 9 ;
233
234     // Compute the Jacobian matrix
235
236     //-----
237     // Jmat2[0] : d^2x/dxi^2
238     // Jmat2[1] : d^2x/det^2
239     // Jmat2[2] : d^2x/dxideta
240     // Jmat2[3] : d^2y/dxi^2
241     // Jmat2[4] : d^2y/deta^2
242     // Jmat2[5] : d^2y/dxideta
243     //-----
244
245     for ( i = 0 ; i < 6 ; i++ ) Jmat2[i] = 0.0 ;
246
247     for ( i = 0 ; i < nnodp ; i++ ){ // start i-loop
248         Jmat2[0] += elxx[i] * dndx2[i] ;
249         Jmat2[1] += elxx[i] * dndy2[i] ;
250         Jmat2[2] += elxx[i] * dndxy[i] ;
251         Jmat2[3] += eleyy[i] * dndx2[i] ;
252         Jmat2[4] += eleyy[i] * dndy2[i] ;
253         Jmat2[5] += eleyy[i] * dndxy[i] ;
254     } // end i-loop
255
256 } // end Jacob subroutine
257
258 //+++++
259 //+++++
260
261 void Find_FEM_nadfm_nodfm( int melem , int mpoin , int *nadfm , int *nodfm ){
262
263     int ipoin ;
264
265     for ( ipoin = 0 ; ipoin < mpoin ; ipoin++ ){
266         nodfm[ipoin] = 1 ;
267     }
268
269     nadfm[0] = 1 ;
270
271     for ( ipoin = 1 ; ipoin < mpoin ; ipoin++ ){
272         nadfm[ipoin] = nadfm[ipoin-1] + nodfm[ipoin-1] ;
273     }
274
275 } // end Find_FEM_nodfm_nadfm subroutine
276
277 //+++++
278 //+++++
279
280 void Find_Reabc( int melem , int mpoin , int *bcnod , int *reabc ){
281
282     //-----
283     // Routine to find the degree of repeated boundary node
284     //-----
285
286     int nnodp = 9 ;
287     int ipoin , jpin ;
288     int ielem ;
289     int temp1 , temp2 ;
290     int n ;
291
292     //----- initialization -----
293
294     for ( ipoin = 0 ; ipoin < mpoin ; ipoin++ ){
295         reabc[ipoin] = 0 ;
296     } // end inodp loop
297
298     for ( ipoin = 1 ; ipoin <= mpoin ; ipoin++ ){ // start ipoin-loop
299
300         temp1 = ipoin ;
301
302         n = 0 ;
303
304         // Get the number of boundary node
305
306         for ( ielem = 0 ; ielem < melem ; ielem++ ){
307             for ( jpin = 0 ; jpin < nnodp ; jpin++ ){
308                 temp2 = bcnod[(ielem*nnodp)+jpin] ;
309                 if ( temp1 == temp2 ) n += 1 ;
310             } // end jpin-loop
311         } // end ielem-loop
312
313         reabc[f2c(ipoin)] = n ;
314     } // end ipoin-loop
315 } // end Find_reabc subroutine
316
317 //+++++
318 //+++++
319
320 int Find_local( int ielem , int bcdof , int *lnods ){
321
322     int nnodp = 9 ;
323     int inodp ;
324     int jnodp ;
325     int locat ;
326
327
328     for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
329         if ( bcdof == lnods[ielem*nnodp+inodp] ) {
330             locat = inodp+1 ;
331         } // end if-statement
332     } // end inodp-loop
333     return locat ;
334 } // end Find_local subroutine
335
336 //+++++
337 //+++++
338
339 void LU_solver( double **A , double *b , double *x , int mpoin ){
340
341     //-----
342     // Routine to solve the linear sysem using LU solver
343     //-----
344
345     int i , j , k , l ;

```

```

346 double M ;
347
348 //-----
349 // eliminate procedure
350 //-----
351
352 for ( i = 0 ; i < mpoin-1 ; ++i ){ // start for loop 1
353     for ( j = i+1 ; j < mpoin ; ++j ){ // start for loop 2
354         M = A[j][i]/A[i][i] ;
355         A[j][i] = 0 ;
356         for ( l = i+1 ; l < mpoin ; ++l ){ // start for loop 3
357             A[j][l] = A[j][l] - M * A[i][l];
358         } // end for loop 3
359         b[j]=b[j]-M*b[i];
360     } // end for loop 2
361 } // end for loop 1
362
363 //-----
364 // backward substitution
365 //-----
366
367 x[mpoin-1] = b[mpoin-1]/A[mpoin-1][mpoin-1];
368 for ( j = mpoin-2 ; j >= 0 ; --j ){ // start for loop 4
369     x[j] = b[j] ;
370     for ( l = mpoin-1 ; l >= j+1 ; --l ){ // start for loop 5
371         x[j] = x[j]-x[l]*A[j][l];
372     } // end for loop 4
373     x[j] = x[j]/A[j][j];
374 } // end for loop 5
375 } // end subroutine
376
377 //+++++
378 //+++++
379
380 void Drivelst( int melem , int mpoin , double *cordx , double *cordy ,
381               int *lnods , double *shadl ){
382
383     //-----
384     // Routine to compute the total value of interpolation
385     // function, derivative functions and the integration
386     // weighted factors under all Gaussian point
387     //-----
388
389     /*****/
390     int ngaus = 3 ; // The number of Gaussian quadrature
391     int nnodp = 9 ; // The number of biquadratic interpolation function
392     int nnodl = 4 ; // The number of bilinear interpolation function
393     int ndime = 2 ;
394     int igauss , jgauss , lgauss ;
395     int inodp , jnodp , ielem ;
396     int inodl , jnodl , id ;
397     int indl , ind2 , gap1 , gap2 ;
398     int idnum[9] ;
399     int idime ;
400     int kgapa , kgali ;
401     int ngapa , ngali , igapa , igali ;
402     /*****/
403     double xeqiv ; // The x-direction Gaussian quadrature point
404     double yeqiv ; // The y-direction Gaussian quadrature point
405     double gpt[3] ; // The Gaussian quadrature points
406     double wet[3] ; // The Gaussian quadrature weighted factors
407     double elxx[9] ; // The x-coordinate of an element
408     double elyy[9] ; // The y-coordinate of an element
409     double Jmat[4] ; // The Jacobian matrix
410     double Jinv[4] ; // The inverse of Jacobian matrix
411     double Jdet ; // The determinant of Jacobian matrix
412     double temp1 ;
413     double temp2 ;
414     /*****/
415     double phil[9] , dndx1[9] , dndy1[9] ; // biquadratic
416     double phi2[4] , dndx2[4] , dndy2[4] ; // bilinear
417     /*****/
418     double cartp[2][9] ;
419     double cartl[2][4] ;
420     double shap[81] , carpg[2][81] ;
421     double shalg[36] , carlg[2][36] ;
422     double areaw[9] ;
423     /*****/
424
425     // Get the Gaussian point and weighed factor
426
427     Gauss( gpt , wet ) ;
428
429     // Compute the value of interpolation and derivative
430     // functions for each element
431
432     for ( ielem = 0 ; ielem < melem ; ielem++ ){ // start ielem-loop
433         for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
434
435             //-----
436             // Get the datas of "ielem"-th element
437             //-----
438             // Note : Since the mesh generator code is written using Fortran
439             // The index of array in Fortran starts from 1, but the
440             // index of array C++ start from 0. It is why the following
441             // index minus 1
442
443             elxx[inodp] = cordx[lnods[(ielem*nnodp)+inodp]-1] ;
444             elyy[inodp] = cordy[lnods[(ielem*nnodp)+inodp]-1] ;
445
446         } // end inodp loop
447
448         // Gaussian quadrature integration
449
450         lgauss = 0 ;
451
452         // Start Gaussian integration index
453
454         for ( igauss = 0 ; igauss < ngauss ; igauss++ ){ // x-dir Gaussian quad.
455             for ( jgauss = 0 ; jgauss < ngauss ; jgauss++ ){ // y-dir Gaussian quad.
456
457                 /* Gaussian quadrature time + 1 */
458
459                 lgauss += 1 ;
460

```

```

461      /* Get the Gaussian quadrature point */
462
463      xeqiv = gpt[igaus] ; // x-directional
464      yeqiv = gpt[jgaus] ; // y-directional
465
466      //=====
467      // [1] Biquadratic element
468      //=====
469
470      // Substitute the Gaussian point into shape functions
471
472      shap9( phil , dndx1 , dndy1 , xeqiv , yeqiv ) ;
473
474      // Compute the Jacobian matrix, inverse
475
476      Jacob( dndx1 , dndy1 , elexx , eleyy , Jmat , Jinv ) ;
477
478      // Compute the determinant
479
480      Jdet = Jmat[0] * Jmat[3] - Jmat[1] * Jmat[2] ;
481
482      /*****
483      /* Compute the coordinate transform
484      *****/
485      /*      dN1      dN9      dN1      dN9
486      /*      ----- & -----
487      /*      dx      dx      dy      dy
488      *****/
489
490      for ( idime = 0 ; idime < ndime ; idime++ ){ // start idime-loop
491
492          // x-direction
493          if ( idime == 0 ){
494              temp1 = Jinv[0] ;
495              temp2 = Jinv[1] ;
496          }
497
498          // y-direction
499          if ( idime == 1 ){
500              temp1 = Jinv[2] ;
501              temp2 = Jinv[3] ;
502          }
503
504          for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
505              cartp[idime][inodp]=temp1*dndx1[inodp]+temp2*dndy1[inodp] ;
506          } // end inodp loop
507      } // end idime loop
508
509      // Compute dxdy = Jdet * dxi * deta
510
511      areaw[lgaus-1] = Jdet * wet[igaus] * wet[jgaus] ;
512
513      /*****
514      /* Store the shape function and its derivative function values */
515      *****/
516
517      for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
518
519          //-----
520          // Lgaus = 1 => kgapa = 0 ~ 8
521          // Lgaus = 2 => kgapa = 9 ~ 17
522          // |
523          // Lgaus = 9 => kgapa = 71 ~ 80
524          //-----
525
526          kgapa = (lgaus-1)*nnodp+inodp ;
527
528          // store shape functions
529
530          shapg[kgapa] = phil[inodp] ;
531
532          // store Derivative shape functions
533
534          carpg[0][kgapa] = cartp[0][inodp] ;
535          carpg[1][kgapa] = cartp[1][inodp] ;
536
537      } // end inodp-loop
538
539      //=====
540      // [2] Bilinear element
541      //=====
542
543      // Substitute the Gaussian point into shape functions
544
545      shap4( phi2 , dndx2 , dndy2 , xeqiv , yeqiv ) ;
546
547      /*****
548      /* Compute the coordinate transform
549      *****/
550      /*      dN1      dN4      dN1      dN4
551      /*      ----- & -----
552      /*      dx      dx      dy      dy
553      *****/
554
555      for ( idime = 0 ; idime < ndime ; idime++ ){ // start idime-loop
556
557          if ( idime == 0 ){
558              temp1 = Jinv[0] ;
559              temp2 = Jinv[1] ;
560          }
561
562          if ( idime == 1 ){
563              temp1 = Jinv[2] ;
564              temp2 = Jinv[3] ;
565          }
566
567          for ( inodl = 0 ; inodl < nnodl ; inodl++ ){ // start inodl-loop
568              cartl[idime][inodl] = temp1*dndx2[inodl]+temp2*dndy2[inodl] ;
569          } // end inodl-loop
570      } // end idime-loop
571
572      //-----
573      // Store the bilinear interpolation function and its
574      // derivative interpolation shape functions
575      //-----

```

```

576
577     for ( inod1 = 0 ; inod1 < nnod1 ; inod1++ ){ // start inod1-loop
578
579         //-----
580         // Lgaus = 1 => kgali = 0 ~ 3
581         // Lgaus = 2 => kgali = 4 ~ 7
582         //      |
583         // Lgaus = 9 => kgali = 32 ~ 35
584         //-----
585
586         kgali = (lgaus-1)*nnod1+inod1 ;
587
588         // Linear interpolation function
589
590         shalg[kgali] = phi2[inod1] ;
591
592         // Derivative interpolation functions
593
594         carlg[0][kgali] = cartl[0][inod1] ;
595         carlg[1][kgali] = cartl[1][inod1] ;
596
597     } // end inod1 loop
598 } // end jgaus loop
599 } // end igaus loop
600
601 // write the all data to array
602
603 ngapa = 81 ;
604 ngali = 36 ;
605 gap1 = 243 ;
606 gap2 = 108 ;
607
608 /*****
609 /* [1] Store the biguadratic interpolation function and
610 /* derivative interpolation function after substituting */
611 /* the Gaussian quadrature points
612 /*****/
613
614 ind1 = 0 ;
615
616 for ( igapa = 0 ; igapa < ngapa ; igapa++ ){
617     ind2 = ind1 + igapa * 3 ;
618     shadl[(ielem*360)+ind2+0] = carpg[0][igapa] ;
619     shadl[(ielem*360)+ind2+1] = carpg[1][igapa] ;
620     shadl[(ielem*360)+ind2+2] = shapg[igapa] ;
621 } // end igapa loop
622
623 /*****
624 /* [2] Store the bilinear interpolation function and derivative */
625 /* interpolation function after substituting the Gaussian */
626 /* quadrature points
627 /*****/
628
629 ind1 = gap1 ;
630
631 for ( igali = 0 ; igali < ngali ; igali++ ){
632     ind2 = ind1 + igali * 3 ;
633     shadl[(ielem*360)+ind2+0] = carlg[0][igali] ;
634     shadl[(ielem*360)+ind2+1] = carlg[1][igali] ;
635     shadl[(ielem*360)+ind2+2] = shalg[igali] ;
636 }
637
638 /*****
639 /* [3] Store the inegration weighted factors
640 /*****/
641
642 ind1 = gap1 + gap2 ;
643
644 for ( igauss = 0 ; igauss < ngauss*ngauss ; igauss++ ){
645     ind2 = ind1 + igauss ;
646     shadl[(ielem*360)+ind2] = areaw[igauss] ;
647 } // end igauss loop
648 } // end ielem loop
649 } // end Drivelst subroutine
650
651 //+++++
652 //+++++
653
654 void mixq9( double *dndxx , double *dndyy , double *dndxy ,
655            double xi , double eta ) {
656
657     double x , y ;
658
659     x = xi ;
660     y = eta ;
661
662     //-----
663     // Second derivative w.r.t xi
664     //-----
665     dndxx[0] = -0.5 * y * ( 1.0 - y ) ;
666     dndxx[1] = y * ( 1.0 - y ) ;
667     dndxx[2] = -0.5 * y * ( 1.0 - y ) ;
668     dndxx[3] = 1.0 - y*y ;
669     dndxx[4] = 0.5 * y * ( 1.0 + y ) ;
670     dndxx[5] = -1.0 * y * ( 1.0 + y ) ;
671     dndxx[6] = 0.5 * y * ( 1.0 + y ) ;
672     dndxx[7] = 1.0 - y*y ;
673     dndxx[8] = -2.0 + 2.0 * y*y ;
674     //-----
675     // Second derivative w.r.t eta
676     //-----
677     dndyy[0] = -0.5 * x * ( 1.0 - x ) ;
678     dndyy[1] = 1.0 - x*x ;
679     dndyy[2] = 0.5 * x * ( 1.0 + x ) ;
680     dndyy[3] = -1.0 * x * ( 1.0 + x ) ;
681     dndyy[4] = 0.5 * x * ( 1.0 + x ) ;
682     dndyy[5] = 1.0 - x*x ;
683     dndyy[6] = -0.5 * x * ( 1.0 - x ) ;
684     dndyy[7] = x * ( 1.0 - x ) ;
685     dndyy[8] = -2.0 + 2.0 * x*x ;
686     //-----
687     // Mixed second derivative w.r.t xi and eta
688     //-----
689     dndxy[0] = 0.25 - 0.5 * x - 0.5 * y + x*y ;
690     dndxy[1] = x - 2.0 * x * y ;

```

```

691     dndxy[2] = -0.25 - 0.5 * x + 0.5 * y + x*y ;
692     dndxy[3] = -y - 2.0 * x * y ;
693     dndxy[4] = 0.25 + 0.5 * x + 0.5 * y + x*y ;
694     dndxy[5] = -x - 2.0 * x * y ;
695     dndxy[6] = -0.25 + 0.5 * x - 0.5 * y + x*y ;
696     dndxy[7] = y - 2.0 * x * y ;
697     dndxy[8] = 4.0 * x * y ;
698     //-----
699 }
700
701 //+++++
702 //+++++
703
704 void Drive2nd( int melem , int mpoin , double *cordx , double *cordy ,
705               int *lnods , double *shad2 ){
706
707     //-----
708     // Routine to compute the total value of interpolation
709     // function, derivative functions and the integration
710     // weighted factors under all Gaussian point
711     //-----
712
713     /*****/
714     int ssize = 3 ;
715     int ngaus = 3 ; // The number of Gaussian quadrature
716     int nnodp = 9 ; // The number of biquadratic interpolation function
717     int nnodl = 4 ; // The number of bilinear interpolation function
718     int ndime = 2 ;
719     int igauss , jgauss , lgauss ;
720     int inodp , jnodp , ielem ;
721     int inodl , jnodl , id ;
722     int indl , ind2 , gap1 , gap2 ;
723     int idime ;
724     int kgapa , kgali ;
725     int ngapa , ngali , igapa , igali ;
726     /*****/
727     double xeqiv ; // The x-direction Gaussian quadrature point
728     double yeqiv ; // The y-direction Gaussian quadrature point
729     double gpt[3] ; // The Gaussian quadrature points
730     double wet[3] ; // The Gaussian quadrature weighted factors
731     double elexx[9] ; // The x-coordinate of an element
732     double eleyy[9] ; // The y-coordinate of an element
733     double Jmat[4] ; // The Jacobian matrix
734     double Jinv[4] ; // The inverse of Jacobian matrix
735     double Jdet ; // The determinant of Jacobian matrix
736     double temp1 ;
737     double temp2 ;
738     /*****/
739     double phil[9] , dndx1[9] , dndy1[9] ; // biquadratic
740     double dndxx[9] , dndyy[9] , dndxy[9] ;
741
742     double dxdxi , dxdet , dydxi , dydet ;
743
744     double dxdxi2 , dydxi2 , dxdet2 , dydet2 , dxdxe , dydxe ;
745
746     double dndx , dndy ;
747
748     /*****/
749
750     double cartpl[2][9] ;
751     double carpgl[2][81] ;
752
753     double cartp2[3][9] ;
754     double carpg2[3][81] ;
755
756     /*****/
757
758     //double A[3][3] ;
759
760     double **A = (double**)malloc(sizeof(double*)*3);
761     for ( i = 0 ; i < 3 ; i++ ){
762         A[i] = (double*)malloc(sizeof(double)*3);
763     }
764
765     double x[3] ;
766     double b[3] ;
767
768     /*****/
769
770     // Get the Gaussian point and weighed factor
771
772     Gauss( gpt , wet ) ;
773
774     // Compute the value of interpolation and derivative
775     // functions for each element
776
777     for ( ielem = 0 ; ielem < melem ; ielem++ ){ // start ielem-loop
778         for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
779
780             //-----
781             // Get the datas of "ielem"-th element
782             //-----
783             // Note : Since the mesh generator code is written using Fortran
784             // The index of array in Fortran starts from 1, but the
785             // index of array C++ start from 0. It is why the following
786             // index minus 1
787
788             elexx[inodp] = cordx[lnods[(ielem*nnodp)+inodp]-1] ;
789             eleyy[inodp] = cordy[lnods[(ielem*nnodp)+inodp]-1] ;
790
791         } // end inodp loop
792
793         // Gaussian quadrature integration
794
795         lgauss = 0 ;
796
797         // Start Gaussian integration index
798
799         for ( igauss = 0 ; igauss < ngauss ; igauss++ ){ // x-dir Gaussian quad.
800             for ( jgauss = 0 ; jgauss < ngauss ; jgauss++ ){ // y-dir Gaussian quad.
801
802                 /* Gaussian quadrature time + 1 */
803
804                 lgauss += 1 ;
805

```

```

806      /* Get the Gaussian quadrature point */
807
808      xeqiv = gpt[igauss] ; // x-directional
809      yeqiv = gpt[jgauss] ; // y-directional
810
811      //=====
812      // [1] Biquadratic element
813      //=====
814
815      // Substirute the Gaussian point into mixed derivative interpolation
816      // function
817
818      mixq9( dndxx , dndyy , dndxy , xeqiv , yeqiv ) ;
819
820      // Substitute the Gaussian point into shape functions
821
822      shap9( phil , dndx1 , dndy1 , xeqiv , yeqiv ) ;
823
824      // Compute the Jacobian matrix, inverse
825
826      Jacob( dndx1 , dndy1 , elexx , eleyy , Jmat , Jinv ) ;
827
828      //-----
829
830      dxddxi = Jmat[0] ;
831      dyddxi = Jmat[1] ;
832      dxdet = Jmat[2] ;
833      dydet = Jmat[3] ;
834
835      dxddxi2 = 0.0 ;
836      dyddxi2 = 0.0 ;
837      dxdet2 = 0.0 ;
838      dydet2 = 0.0 ;
839      dxddxe = 0.0 ;
840      dyddxe = 0.0 ;
841
842      for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
843          dxddxi += elexx[inodp] * dndxx[inodp] ; // d^2x/dxi^2
844          dyddxi += eleyy[inodp] * dndxx[inodp] ; // d^2y/dxi^2
845          dxdet2 += elexx[inodp] * dndyy[inodp] ; // d^2x/deta^2
846          dydet2 += eleyy[inodp] * dndyy[inodp] ; // d^2y/deta^2
847          dxddxe += elexx[inodp] * dndxy[inodp] ; // d^2x/dxideta
848          dyddxe += eleyy[inodp] * dndxy[inodp] ; // d^2y/dxideta
849      } // end inodp-loop
850
851      /*****
852      /* Compute the coordinate transform
853      /*****
854      /*      dN1      dN9      dN1      dN9
855      /*      ----- & -----
856      /*      dx      dx      dy      dy
857      /*****
858
859      for ( idime = 0 ; idime < ndime ; idime++ ){ // start idime-loop
860          // x-direction
861          if ( idime == 0 ){
862              temp1 = Jinv[0] ;
863              temp2 = Jinv[1] ;
864          }
865          // y-direction
866          if ( idime == 1 ){
867              temp1 = Jinv[2] ;
868              temp2 = Jinv[3] ;
869          }
870          for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
871              cartpl[idime][inodp]=temp1*dndx1[inodp]+temp2*dndy1[inodp] ;
872          } // end inodp loop
873      } // end idime loop
874
875      //-----
876      // Compute the 2nd derivative function
877      //-----
878
879      A[0][0] =      dxddxi * dxddxi ;
880      A[0][1] =      dyddxi * dyddxi ;
881      A[0][2] = 2.0 * dxddxi * dyddxi ;
882      A[1][0] =      dxdet * dxdet ;
883      A[1][1] =      dydet * dydet ;
884      A[1][2] = 2.0 * dxdet * dydet ;
885      A[2][0] = dxddxi * dxdet ;
886      A[2][1] = dyddxi * dydet ;
887      A[2][2] = dxddxi * dydet + dxdet * dyddxi ;
888
889      for ( inodp = 0 ; inodp < nnodp ; inodp++ ){
890
891          b[0]=dndxx[inodp]-cartpl[0][inodp]*dxddxi2-cartpl[1][inodp]*dyddxi2 ;
892          b[1]=dndyy[inodp]-cartpl[0][inodp]*dxdet2-cartpl[1][inodp]*dydet2 ;
893          b[2]=dndxy[inodp]-cartpl[0][inodp]*dxddxe-cartpl[1][inodp]*dyddxe ;
894
895          // Solve the small linear system
896
897          LU_solver( A , b , x , ssize ) ;
898
899          //-----
900
901          cartp2[0][inodp]=x[0] ; // dN/dx2
902          cartp2[1][inodp]=x[1] ; // dN/dy2
903          cartp2[2][inodp]=x[2] ; // dN/dxy
904
905      } // end inodp-loop
906
907      /*****
908      /* Store the shape function and its derivative function values */
909      /*****
910
911      for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
912
913          //-----
914          // Lgauss = 1 => kgapa = 0 ~ 8
915          // Lgauss = 2 => kgapa = 9 ~ 17
916          //      |
917          // Lgauss = 9 => kgapa = 71 ~ 80
918          //-----
919
920          kgapa = (lgauss-1)*nnodp+inodp ;

```

```

921      // store Derivative shape functions
922
923      carpg2[0][kgapa] = cartp2[0][inodp] ;
924      carpg2[1][kgapa] = cartp2[1][inodp] ;
925      carpg2[2][kgapa] = cartp2[2][inodp] ;
926
927  } // end inodp-loop
928
929  } // end jgaus loop
930  } // end igaus loop
931
932  // write the all data to array
933
934  ngapa = 81 ;
935  gap1 = 243 ;
936
937  /* Store the 2nd derivative biquadratic interpolation
938  /* function under the all Gaussian quadrature points
939  /*
940  /*
941
942  ind1 = 0 ;
943
944  for ( igapa = 0 ; igapa < ngapa ; igapa++ ){
945      ind2 = ind1 + igapa * 3 ;
946      shad2[(ielem*243)+ind2+0] = carpg2[0][igapa] ;
947      shad2[(ielem*243)+ind2+1] = carpg2[1][igapa] ;
948      shad2[(ielem*243)+ind2+2] = carpg2[2][igapa] ;
949  } // end igapa loop
950
951  } // end ielem loop
952  } // end Drive2nd subroutine
953
954  //+++++
955  //+++++
956
957  void Drive2nd_02( int melem, int mpoin, double *cordx, double *cordy,
958                  int *lnods, double *shad2 ){
959
960      //-----
961      // Routine to compute the total value of interpolation
962      // function, derivative functions and the integration
963      // weighted factors under all Gaussian point
964      //-----
965
966      /*
967      int ssize = 3 ;
968      int ngaus = 3 ; // The number of Gaussian quadrature
969      int nnodp = 9 ; // The number of biquadratic interpolation function
970      int nnodl = 4 ; // The number of bilinear interpolation function
971      int ndime = 2 ;
972      int igaus, jgaus, lgaus ;
973      int inodp, jnodp, ielem ;
974      int inodl, jnodl, id ;
975      int ind1, ind2, gap1, gap2 ;
976      int idime ;
977      int kgapa, kgali ;
978      int ngapa, ngali, igapa, igali ;
979      /*
980      double xeqiv ; // The x-direction Gaussian quadrature point
981      double yeqiv ; // The y-direction Gaussian quadrature point
982      double gpt[3] ; // The Gaussian quadrature points
983      double wet[3] ; // The Gaussian quadrature weighted factors
984      double elxx[9] ; // The x-coordinate of an element
985      double elyy[9] ; // The y-coordinate of an element
986      double Jmat[4] ; // The Jacobian matrix
987      double Jmat2[6] ;
988      double Jinv[4] ; // The inverse of Jacobian matrix
989      double Jdet ; // The determinant of Jacobian matrix
990      double temp1 ;
991      double temp2 ;
992      /*
993      double phil[9] ;
994      double dndx1[9] ; // dN/dxi
995      double dndel[9] ; // dN/deta
996
997      double dndx2[9] ; // d^2N/dxi^2
998      double dnde2[9] ; // d^2N/deta^2
999      double dndxe[9] ; // d^2N/dxideta
1000
1001      double dxdx1 ; // dx/dxi
1002      double dxdel ; // dx/deta
1003      double dydx1 ; // dy/dxi
1004      double dydel ; // dy/deta
1005
1006      double dxdx2 ; // d^2x/dxi^2
1007      double dxde2 ; // d^2x/deta^2
1008      double dxdxe ; // d^2x/dxideta
1009
1010      double dydx2 ; // d^2y/dxi^2
1011      double dyde2 ; // d^2y/deta^2
1012      double dydxe ; // d^2y/dxideta
1013
1014      /*
1015      double cartp1[2][9] ;
1016      double carpg1[2][81] ;
1017
1018      double cartp2[2][9] ;
1019      double carpg2[2][81] ;
1020
1021      /*
1022      // Get the Gaussian point and weighed factor
1023      Gauss( gpt, wet ) ;
1024
1025      // Compute the value of interpolation and derivative
1026      // functions for each element
1027
1028      for ( ielem = 0 ; ielem < melem ; ielem++ ){ // start ielem-loop
1029          for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
1030              //-----
1031

```

```

1036 // Get the datas of "ielem"-th element
1037 //-----
1038 // Note : Since the mesh generator code is written using Fortran
1039 // The index of array in Fortran starts from 1, but the
1040 // index of array C++ start from 0. It is why the following
1041 // index minus 1
1042
1043 elcex[inodp] = cordx[lndos[(ielem*nnodp)+inodp]-1] ;
1044 elcyy[inodp] = cordy[lndos[(ielem*nnodp)+inodp]-1] ;
1045
1046 } // end inodp loop
1047
1048 // Gaussian quadrature integration
1049
1050 lgaus = 0 ;
1051
1052 // Start Gaussian integration index
1053
1054 for ( igaos = 0 ; igaos < ngaos ; igaos++ ){ // x-dir Gaussian quad.
1055     for ( jgaos = 0 ; jgaos < ngaos ; jgaos++ ){ // y-dir Gaussian quad.
1056
1057         /* Gaussian quadrature time + 1 */
1058
1059         lgaus += 1 ;
1060
1061         /* Get the Gaussian quadrature point */
1062
1063         xeqiv = gpt[igaos] ; // x-directional
1064         yeqiv = gpt[jgaos] ; // y-directional
1065
1066         //=====
1067         // [1] Biquadratic element
1068         //=====
1069
1070         // Substirute the Gaussian point into mixed derivative interpolation
1071         // function
1072
1073         mixq9( dndx2 , dnde2 , dndxe , xeqiv , yeqiv ) ;
1074
1075         // Substitute the Gaussian point into shape functions
1076
1077         shap9( phil , dndx1 , dndel , xeqiv , yeqiv ) ;
1078
1079         // Compute the Jacobian matrix, inverse
1080
1081         Jacob( dndx1 , dndel , elcex , elcyy , Jmat , Jinv ) ;
1082
1083         // Compute the 2nd derivative geometric tensor
1084
1085         GeoTensor2( dndx2 , dnde2 , dndxe , elcex , elcyy , Jmat2 ) ;
1086
1087         // Compute the determinant
1088
1089         Jdet = Jmat[0] * Jmat[3] - Jmat[1] * Jmat[2] ;
1090
1091         //-----
1092
1093         dxdx1 = Jmat[0] ;
1094         dydx1 = Jmat[1] ;
1095         dxdel = Jmat[2] ;
1096         dydel = Jmat[3] ;
1097
1098         dxdx2 = Jmat2[0] ; // d^2x/dxi^2
1099         dxde2 = Jmat2[1] ; // d^2x/deta^2
1100         dxdxe = Jmat2[2] ; // d^2x/dxideta
1101
1102         dydx2 = Jmat2[3] ; // d^2y/dxi^2
1103         dyde2 = Jmat2[4] ; // d^2y/deta^2
1104         dydxe = Jmat2[5] ; // d^2y/dxideta
1105
1106         /*-----
1107         /* Compute the coordinate transform
1108         /*-----
1109         /* d^2N1 d^2N9 d^2N1 d^2N9
1110         /* ----- & ----- ~ -----
1111         /* dx^2 dx^2 dy^2 dy^2
1112         /*-----
1113
1114         // Compute the 1st derivative function first
1115
1116         for ( idime = 0 ; idime < ndime ; idime++ ){ // start idime-loop
1117             // x-direction
1118             if ( idime == 0 ){
1119                 temp1 = Jinv[0] ;
1120                 temp2 = Jinv[1] ;
1121             }
1122             // y-direction
1123             if ( idime == 1 ){
1124                 temp1 = Jinv[2] ;
1125                 temp2 = Jinv[3] ;
1126             }
1127             for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
1128                 cartpl[idime][inodp] = temp1 * dndx1[inodp] +
1129                                     temp2 * dndel[inodp] ;
1130             } // end inodp loop
1131         } // end idime loop
1132
1133         // Compute the 2nd derivative function
1134
1135         for ( idime = 0 ; idime < ndime ; idime++ ){ // start idime-loop
1136             for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
1137                 // idime = 0 --> d^2N/dx^2
1138                 // idime = 1 --> d^2N/dy^2
1139                 if ( idime == 0 )
1140                 {
1141                     cartp2[idime][inodp] = ( dndx2[inodp]*dydel*dydel +
1142                                                dnde2[inodp]*dydx1*dydx1 +
1143                                                dndx1[inodp]*(dydel*dydxe-dydx1*dyde2)+
1144                                                dndel[inodp]*(dydx1*dydxe-dydel*dydx2)-
1145                                                2.0*dndxe[inodp]*dydx1*dydel ) / Jdet ;
1146                 }
1147                 else
1148                 {
1149                     cartp2[idime][inodp] = ( dndx2[inodp]*dxdel*dxdel +
1150                                                dnde2[inodp]*dxdx1*dxdx1 +

```

```

1151         dndx1[inodp]*(dxdel1*dxdx1-dxdel1*dxdx2)+
1152         dndel[inodp]*(dxdx1*dxdx1-dxdel1*dxdx2)-
1153         2.0*dndxe[inodp]*dxdx1*dxdel1 ) / Jdet ;
1154     }
1155 } // end inodp loop
1156 } // end idime loop
1157
1158
1159 /******
1160 /* Store the shape function and its derivative function values */
1161 /******
1162
1163 for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
1164
1165     //-----
1166     // Lgaus = 1 => kgapa = 0 ~ 8
1167     // Lgaus = 2 => kgapa = 9 ~ 17
1168     // |
1169     // Lgaus = 9 => kgapa = 71 ~ 80
1170     //-----
1171
1172     kgapa = (lgaus-1)*nnodp+inodp ;
1173
1174     // store Derivative shape functions
1175
1176     carpg2[0][kgapa] = cartp2[0][inodp] ;
1177     carpg2[1][kgapa] = cartp2[1][inodp] ;
1178
1179 } // end inodp-loop
1180
1181 } // end jgaus loop
1182 } // end igaus loop
1183
1184 // write the all data to array
1185
1186 ngapa = 81 ;
1187
1188 /******
1189 /* Store the 2nd derivative biquadratic interpolation */
1190 /* function under the all Gaussian quadrature points */
1191 /******
1192
1193 ind1 = 0 ;
1194
1195 for ( igapa = 0 ; igapa < ngapa ; igapa++ ){
1196     ind2 = ind1+igapa*2 ;
1197     shad2[(ielem*162)+ind2+0] = carpg2[0][igapa] ;
1198     shad2[(ielem*162)+ind2+1] = carpg2[1][igapa] ;
1199 } // end igapa loop
1200
1201 } // end ielem loop
1202 } // end Drive2nd_02 subroutine
1203
1204 //+++++
1205 //+++++
1206
1207 void deg_u( int *usequ ){
1208     usequ[0] = 0 ;
1209     usequ[1] = 1 ;
1210     usequ[2] = 2 ;
1211     usequ[3] = 3 ;
1212     usequ[4] = 4 ;
1213     usequ[5] = 5 ;
1214     usequ[6] = 6 ;
1215     usequ[7] = 7 ;
1216     usequ[8] = 8 ;
1217 }
1218
1219 //+++++
1220 //+++++
1221
1222 void Matrix_Galerkin( int ielem , double *cordx , double *cordy , double *elexx ,
1223                     double *eleyy , double *elecex , double *elecyc , double kappa ,
1224                     int *nadm , int *nodfm , int *lboud , double *shad1 ,
1225                     double *matmx , double *lorhs ){
1226
1227     //-----
1228     // Routine to compute the element matrix using DRP-SUPG model
1229     //-----
1230     // Variable :
1231     // - cordx : The global x-component coordinate
1232     // - cordy : The global y-component coordinate
1233     // - elexx : The x-component coordinate in an element
1234     // - eleyy : The y-component coordinate in an element
1235     // - ielem : The element index
1236     // - kappa : The diffusivity
1237     // - elecex : The x-component velocity in an element
1238     // - elecyc : The y-component velocity in an element
1239     // - lboud : The boundary indicator
1240     // - boudv : The value of boundary condition
1241     // - matmx : The global fluid matrix
1242     // - lorhs : The local right hand side
1243     //-----
1244
1245     int nnodp = 9 ;
1246     int nnodl = 4 ;
1247     int nevab = 9 ;
1248     int lgaus = 9 ;
1249     int msize = 81 ;
1250
1251     int i , j , k , idime , jdime ;
1252     int igaus , inodp , jnodp ;
1253     int kgapa , kgali , kpoin ;
1254     int ievab , jevab , bcdof ;
1255     int ind1 , ind2 , gap1 , gap2 ;
1256     int igali , igapa , ngapa , ngali ;
1257     int irow , jcol , usequ[9] ;
1258     int templ , temp2 ;
1259     int itotu , redof ;
1260     //-----
1261     double shapi ; // Ni
1262     double shapj ; // Nj
1263     double carxi ; // dNidx
1264     double carxj ; // dNjdx
1265     double caryi ; // dNidy

```

```

1266 double caryj ; // dNjdy
1267 double darea ; // |J|*dxi*deta
1268 //-----
1269 double shalg[36], carlg[2][36], shapg[81], carpg[2][81];
1270 double shapp[9], cartp[2][9], shapl[4], cartl[2][4];
1271 double areaw[9];
1272 double flumx[neva][neva];
1273 //-----
1274 double diff ; // diffusion term
1275 double conv ; // convection term
1276 double stabc ; // stabilized term (conv)
1277 double stabd ; // stabilized term (diff)
1278 double visco ; // Viscosity
1279 double eledx ; // x-directional element length
1280 double eledy ; // y-directional element length
1281 double sourx ; // The source term
1282 //-----
1283 double locax ;
1284 double locay ;
1285 double locau ;
1286 double locav ;
1287 double weigi ;
1288 //-----
1289
1290 // Get the number of variables
1291
1292 deg_u( usequ );
1293
1294 //-----
1295 // 針對每個元素,一組一組取出其形狀函數與導函數
1296 //-----
1297
1298 ngapa = 81 ;
1299 ngali = 36 ;
1300 gap1 = 243 ;
1301 gap2 = 108 ;
1302
1303 //===== [1] 取出拋物線形狀函數與導函數 =====
1304
1305 ind1 = 0 ;
1306
1307 for ( igapa = 0 ; igapa < ngapa ; igapa++ ){ // start igapa-loop
1308     ind2=ind1+igapa*3 ;
1309     //-----
1310     carpg[0][igapa] = shadl[(ielem*360)+ind2+0];
1311     carpg[1][igapa] = shadl[(ielem*360)+ind2+1];
1312     shapg[igapa] = shadl[(ielem*360)+ind2+2];
1313 } // end igapa-loop
1314
1315 //===== [2] 取出線性形狀函數與導函數 =====
1316
1317 ind1 = gap1 ; // 243
1318
1319 for ( igali = 0 ; igali < ngali ; igali++ ){
1320     ind2=ind1+igali*3 ;
1321     //-----
1322     carlg[0][igali] = shadl[(ielem*360)+ind2+0];
1323     carlg[1][igali] = shadl[(ielem*360)+ind2+1];
1324     shalg[igali] = shadl[(ielem*360)+ind2+2];
1325 } // end igali loop
1326
1327 //===== [3] 取出積分權重 =====
1328
1329 ind1 = gap1+gap2 ; // 351
1330
1331 for ( igauss = 0 ; igauss < lgauss ; igauss++ ){
1332     ind2=ind1+igauss ;
1333     areaw[igauss] = shadl[(ielem*360)+ind2+0] ;
1334 } // end igauss loop
1335
1336 //*****
1337 for ( ievab = 0 ; ievab < nevab ; ievab++ ){
1338     for ( jevab = 0 ; jevab < nevab ; jevab++ ){
1339         flumx[ievab][jevab] = 0.0 ;
1340     } // end jevab loop
1341 } // end ievab loop
1342 //*****
1343
1344 //-----
1345 // Start Gaussian integration loop
1346 //-----
1347
1348 for ( igauss = 1 ; igauss <= lgauss ; igauss++ ){
1349     /**** Get the integration weight factor *****/
1350
1351     darea = areaw[igauss-1] ;
1352
1353     /**** 依序抓取一組一組積分點所對應的形狀函數與導函數 *****/
1354
1355     for ( inodp = 0 ; inodp < nnodp ; inodp++ ){
1356         //-----
1357         kgapa = (igauss-1)*nnodp+inodp ;
1358         //-----
1359         cartp[0][inodp] = carpg[0][kgapa] ;
1360         cartp[1][inodp] = carpg[1][kgapa] ;
1361         shapp[inodp] = shapg[kgapa] ;
1362         //-----
1363     } // end inodp loop
1364
1365     /**** 轉換物理量到計算空間上 *****/
1366
1367     locau = 0.0 ;
1368     locav = 0.0 ;
1369     locax = 0.0 ;
1370     locay = 0.0 ;
1371
1372     //-----
1373     for ( inodp = 0 ; inodp < nnodp ; inodp++ ){
1374         locau += shapp[inodp]*elec[0][inodp] ;
1375         locav += shapp[inodp]*elec[1][inodp] ;
1376         locax += shapp[inodp]*ellex[inodp] ;
1377         locay += shapp[inodp]*eleyy[inodp] ;
1378     } // end inodp loop
1379     //-----
1380

```

```

1381 // diffusivity coefficient
1382
1383 visco = kappa ;
1384
1385 for ( inodp = 0 ; inodp < nnodp ; inodp++ ){
1386
1387     /*****
1388     shapi = shapp[inodp] ; // Ni
1389     carxi = cartp[0][inodp] ; // dNidx
1390     caryi = cartp[1][inodp] ; // dNidy
1391     *****/
1392
1393     // weighted function
1394
1395     weigi = shapi ;
1396
1397     // local row index
1398
1399     irow = usequ[inodp] ;
1400
1401     for ( jnodp = 0 ; jnodp < nnodp ; jnodp++ ){
1402
1403         // local column index
1404
1405         jcol = usequ[jnodp] ;
1406
1407         /*****
1408         shapj = shapp[jnodp] ; // Nj
1409         carxj = cartp[0][jnodp] ; // dNjdx
1410         caryj = cartp[1][jnodp] ; // dNjdy
1411         *****/
1412
1413         // Compute the diffusion term
1414
1415         diff = kappa * ( carxi * carxj + caryi * caryj ) * darea ;
1416
1417         // Compute the convection term
1418
1419         conv = shapi * ( locau * carxj + locav * caryj ) * darea ;
1420
1421         // Add the convection and diffusion term to the matrix
1422
1423         flumx[irow][jcol] = flumx[irow][jcol] + diff + conv ;
1424
1425     } // end jnodp loop
1426 } // end inodp loop
1427 } // end igauss loop
1428
1429 //+++++
1430 // Modify the matrix if necessary
1431 //+++++
1432
1433 for ( inodp = 0 ; inodp < nnodp ; inodp++ ){
1434
1435     // Take the number label of boundary
1436
1437     bcdof = bcnod[ielem*nnodp+inodp] ;
1438
1439     // Boundary number label = 0 --> Interior node
1440     // --> It does not need to fix matrix
1441
1442     if ( bcdof == -1 ) continue ;
1443
1444     // Take the degree of repeation of node
1445
1446     redof = reabc[bcdof-1] ;
1447
1448     // Translate the global node number to local node number and use it
1449     // to find the corresponding row location of elementary matrix
1450
1451     temp1 = Find_local( ielem , bcdof , lnods ) ;
1452
1453     // Take the dof
1454
1455     itotu = nadfm[bcdof-1] ;
1456
1457     //-----
1458     // Modify the matrix
1459     //-----
1460
1461     if ( lboud[itotu-1] == 1 ) {
1462
1463         temp2 = usequ[temp1-1] ;
1464
1465         for ( ievab = 0 ; ievab < nevab ; ievab++ ){ // start ievab-loop
1466             flumx[temp2][ievab] = 0.0 ;
1467         } // end ievab-loop
1468
1469         flumx[temp2][temp2] = 1.0 / redof ;
1470     }
1471 } // end inodp-loop
1472
1473 // 二維矩陣 flumx 壓縮為一維儲存在 matmx 中
1474
1475 k = 0 ;
1476
1477 //-----
1478 for ( ievab = 0 ; ievab < nevab ; ievab++ ){
1479     for ( jevab = 0 ; jevab < nevab ; jevab++ ){
1480         matmx[ielem*msize+k] = flumx[ievab][jevab] ;
1481         k += 1 ;
1482     }
1483 }
1484 //-----
1485
1486 } // end Matrix subroutine
1487
1488 //+++++
1489 //+++++
1490
1491 void Matrix_DRP_SUPG( int ielem , double *cordx , double *cordy , double *elexx ,
1492                      double *eleyy , double *elec , double *elec , double *elecy , double *elecx ,
1493                      int *nodfm , int *lboud , double *shad1 ,
1494                      double *shad2 , double *matmx , double *lorhs ){
1495

```

```

1496 //-----
1497 // Routine to compute the element matrix using DRP-SUPG model
1498 //-----
1499 // Variable :
1500 // - cordx : The global x-component coordinate
1501 // - cordy : The global y-component coordinate
1502 // - elcex : The x-component coordinate in an element
1503 // - elcex : The y-component coordinate in an element
1504 // - ielem : The element index
1505 // - kappa : The diffusivity
1506 // - elecex : The x-component velocity in an element
1507 // - elcex : The y-component velocity in an element
1508 // - lboud : The boundary indicator
1509 // - boudv : The value of boundary condition
1510 // - matmx : The global fluid matrix
1511 // - lorhs : The local right hand side
1512 //-----
1513
1514 int nnodp = 9 ;
1515 int nnodl = 4 ;
1516 int nevab = 9 ;
1517 int lgaus = 9 ;
1518 int msize = 81 ;
1519
1520 int i , j , k , idime , jdime ;
1521 int igauss , inodp , jnodp ;
1522 int kgapa , kgali , kpoin ;
1523 int ievab , jevab , bcdof ;
1524 int ind1 , ind2 , gap1 , gap2 ;
1525 int igali , igapa , ngapa , ngali ;
1526 int irow , jcol , usequ[9] ;
1527 int templ , temp2 ;
1528 int itotu , redof ;
1529 //-----
1530 double shapi ; // Ni
1531 double shapj ; // Nj
1532 double dnidx ; // dNi/dx
1533 double dnjdx ; // dNj/dx
1534 double dnidy ; // dNi/dy
1535 double dnjdy ; // dNj/dy
1536 double darea ; // |J|*dxi*deta
1537 double dnjdx2 ; // d^2Nj/dx^2
1538 double dnjdy2 ; // d^2Nj/dy^2
1539 double dnjdxxy ; // d^2Nj/dxdy
1540 //-----
1541 double shalg[36] , shapg[81] , cartl[2][4] , carlg[2][36] ;
1542 double shapl[4] , shapp[9] ;
1543
1544 double carpg1[2][81] ;
1545 double cartp1[2][9] ;
1546 double carpg2[2][81] ;
1547 double cartp2[2][9] ;
1548
1549 double areaw[9] ;
1550 double artdf[9] ;
1551
1552 double flumx[nevab][nevab] ;
1553
1554 //-----
1555
1556 double diff ; // diffusion term
1557 double conv ; // convection term
1558 double stabc ; // stabilized term (conv)
1559 double stabd ; // stabilized term (diff)
1560 double visco ; // Viscosity
1561 double eledx ; // x-directional element length
1562 double eledy ; // y-directional element length
1563 double sourx ; // The source term
1564
1565 double x4 , y4 , x8 , y8 ;
1566 double x2 , y2 , x6 , y6 ;
1567
1568 //-----
1569 double locax ;
1570 double locay ;
1571 double locau ;
1572 double locav ;
1573 double tau ;
1574 //-----
1575
1576 // Get the number of variables
1577 deg_u( usequ ) ;
1578
1579 //-----
1580
1581 // 針對每個元素,一組一組取出其形狀函數與導函數
1582 //-----
1583
1584 ngapa = 81 ;
1585 ngali = 36 ;
1586 gap1 = 243 ;
1587 gap2 = 108 ;
1588
1589 //===== [1] 取出拋物線形狀函數與導函數 =====
1590
1591 ind1 = 0 ;
1592
1593 for ( igapa = 0 ; igapa < ngapa ; igapa++ ){ // start igapa-loop
1594 ind2=ind1+igapa*3 ;
1595 //-----
1596 carpg1[0][igapa] = shad1[(ielem*360)+ind2+0]; // dN/dx
1597 carpg1[1][igapa] = shad1[(ielem*360)+ind2+1]; // dN/dy
1598 shapg[igapa] = shad1[(ielem*360)+ind2+2]; // N
1599 } // end igapa-loop
1600
1601 for ( igapa = 0 ; igapa < ngapa ; igapa++ ){ // start igapa-loop
1602 ind2=ind1+igapa*2 ;
1603 //-----
1604 carpg2[0][igapa] = shad2[(ielem*162)+ind2+0]; // d^2N/dx^2
1605 carpg2[1][igapa] = shad2[(ielem*162)+ind2+1]; // d^2N/dy^2
1606 } // end igapa-loop
1607
1608 //===== [2] 取出線性形狀函數與導函數 =====
1609
1610

```

```

1611 ind1 = gap1 ; // 243
1612
1613 for ( igali = 0 ; igali < ngali ; igali++ ){
1614     ind2=ind1+igali*3 ;
1615     //-----
1616     carlg[0][igali] = shad1[(ielem*360)+ind2+0];
1617     carlg[1][igali] = shad1[(ielem*360)+ind2+1];
1618     shalg[igali] = shad1[(ielem*360)+ind2+2];
1619 } // end igali loop
1620
1621 //===== [3] 取出積分權重 =====
1622
1623 ind1 = gap1+gap2 ; // 351
1624
1625 for ( igauss = 0 ; igauss < lgauss ; igauss++ ){
1626     ind2=ind1+igauss ;
1627     areaw[igauss] = shad1[(ielem*360)+ind2+0] ;
1628 } // end igauss loop
1629
1630 //*****
1631 for ( ievab = 0 ; ievab < nevab ; ievab++ ){
1632     for ( jevab = 0 ; jevab < nevab ; jevab++ ){
1633         flumx[ievab][jevab] = 0.0 ;
1634     } // end jevab loop
1635 } // end ievab loop
1636 //*****
1637
1638 // Compute the characteristic element length
1639
1640 x2 = cordx[lnodes[ielem*nnodp+1]-1] ;
1641 x4 = cordx[lnodes[ielem*nnodp+3]-1] ;
1642 x6 = cordx[lnodes[ielem*nnodp+5]-1] ;
1643 x8 = cordx[lnodes[ielem*nnodp+7]-1] ;
1644
1645 y2 = cordy[lnodes[ielem*nnodp+1]-1] ;
1646 y4 = cordy[lnodes[ielem*nnodp+3]-1] ;
1647 y6 = cordy[lnodes[ielem*nnodp+5]-1] ;
1648 y8 = cordy[lnodes[ielem*nnodp+7]-1] ;
1649
1650 eledx = sqrt((x4-x8)*(x4-x8)+(y4-y8)*(y4-y8));
1651 eledy = sqrt((x2-x6)*(x2-x6)+(y2-y6)*(y2-y6));
1652
1653 //-----
1654 //          Start Gaussian integration loop
1655 //-----
1656
1657 for ( igauss = 1 ; igauss <= lgauss ; igauss++ ){
1658
1659     /**** Get the integration weight factor ****/
1660
1661     darea = areaw[igauss-1] ;
1662
1663     /**** 依序抓取一組一組積分點所對應的形狀函數與導函數 *****/
1664
1665     for ( inodp = 0 ; inodp < nnodp ; inodp++ ){
1666         //-----
1667         kgapa = (igauss-1)*nnodp+inodp ;
1668         //-----
1669         cartpl[0][inodp] = carpg1[0][kgapa] ;
1670         cartpl[1][inodp] = carpg1[1][kgapa] ;
1671         cartp2[0][inodp] = carpg2[0][kgapa] ;
1672         cartp2[1][inodp] = carpg2[1][kgapa] ;
1673         shapp[inodp] = shapg[kgapa] ;
1674         //-----
1675     } // end inodp loop
1676
1677     /**** 轉換物理量到計算空間上 *****/
1678
1679     locau = 0.0 ;
1680     locav = 0.0 ;
1681     locax = 0.0 ;
1682     locay = 0.0 ;
1683
1684     //-----
1685     for ( inodp = 0 ; inodp < nnodp ; inodp++ ){
1686         locau += shapp[inodp]*elec[inodp] ;
1687         locav += shapp[inodp]*elec[inodp] ;
1688         locax += shapp[inodp]*elex[inodp] ;
1689         locay += shapp[inodp]*eley[inodp] ;
1690     } // end inodp loop
1691     //-----
1692
1693     // Get the stabilized parameter
1694
1695     Tau2D_SUPG( locau , locav , eledx , eledy , artdf ) ;
1696
1697     // diffusivity coefficient
1698
1699     visco = kappa ;
1700
1701     for ( inodp = 0 ; inodp < nnodp ; inodp++ ){
1702
1703         tau = artdf[inodp] ;
1704
1705         /*****
1706         shapi = shapp[inodp] ; // Ni
1707         dnidx = cartpl[0][inodp] ; // dNi/dx
1708         dnidy = cartpl[1][inodp] ; // dNi/dy
1709         *****/
1710
1711         // local row index
1712
1713         irow = usequ[inodp] ;
1714
1715         for ( jnodp = 0 ; jnodp < nnodp ; jnodp++ ){
1716
1717             // local column index
1718
1719             jcol = usequ[jnodp] ;
1720
1721             /*****
1722             shapj = shapp[jnodp] ; // Nj
1723             dnjdx = cartpl[0][jnodp] ; // dNj/dx
1724             dnjdy = cartpl[1][jnodp] ; // dNj/dy
1725             dnjdx2 = cartp2[0][jnodp] ; // d^2Nj/dx^2

```

```

1726     dnjdy2 = cartp2[1][jnodp] ; // d^2Nj/dy^2
1727     /*****/
1728
1729     // Compute the diffusion term
1730
1731     diff = visco * ( dnidx * dnjdx + dnidy * dnjdy ) * darea ;
1732
1733     // Compute the convection term
1734
1735     conv = shapi * ( locau * dnjdx + locav * dnjdy ) * darea ;
1736
1737     // Compute the stabilized term (convection)
1738
1739     stabc = tau * ( locau * dnidx + locav * dnidy ) *
1740             ( locau * dnjdx + locav * dnjdy ) * darea ;
1741
1742     // Compute the stabilized term (diffusion)
1743
1744     stabd = tau * ( locau * dnidx + locav * dnidy ) *
1745             -visco * ( dnjdx2 + dnjdy2 ) * darea ;
1746
1747     // Add the convection and diffusion term to the matrix
1748
1749     flumx[irow][jcol]=flumx[irow][jcol]+diff+conv+stabc+stabd ;
1750
1751 } // end jnodp loop
1752 } // end inodp loop
1753 } // end igaus loop
1754
1755 //+++++
1756 // Modify the matrix if necessary
1757 //+++++
1758
1759 for ( inodp = 0 ; inodp < nnodp ; inodp++ ){
1760
1761     // Take the number label of boundary
1762
1763     bcdof = bcnod[ielem*nnodp+inodp] ;
1764
1765     // Boundary number label = 0 --> Interior node
1766     // --> It does not need to fix matrix
1767
1768     if ( bcdof == -1 ) continue ;
1769
1770     // Take the degree of repetition of node
1771
1772     redof = reabc[bcdof-1] ;
1773
1774     // Translate the global node number to local node number and use it
1775     // to find the corresponding row location of elementary matrix
1776
1777     temp1 = Find_local( ielem , bcdof , lnods ) ;
1778
1779     // Take the dof
1780
1781     itotu = nadfm[bcdof-1] ;
1782
1783     //-----
1784     // Modify the matrix
1785     //-----
1786
1787     if ( lboud[itotu-1] == 1 ) {
1788
1789         temp2 = usequ[temp1-1] ;
1790
1791         for ( ievab = 0 ; ievab < nevab ; ievab++ ){ // start ievab-loop
1792             flumx[temp2][ievab] = 0.0 ;
1793         } // end ievab-loop
1794
1795         flumx[temp2][temp2] = 1.0 / redof ;
1796     }
1797 } // end inodp-loop
1798
1799 // 二維矩陣 flumx 壓縮為一維儲存在 matmx 中
1800
1801 k = 0 ;
1802
1803 //-----
1804 for ( ievab = 0 ; ievab < nevab ; ievab++ ){
1805     for ( jevab = 0 ; jevab < nevab ; jevab++ ){
1806         matmx[ielem*msize+k] = flumx[ievab][jevab] ;
1807         k += 1 ;
1808     }
1809 }
1810 //-----
1811
1812 } // end Matrix subroutine
1813
1814 //+++++
1815 //+++++
1816
1817 void Read_mesh( int     melem , int     mpoin , int     mtotv ,
1818                double  kappa , double *cordx , double *cordy ,
1819                int     *lnods , int     *bcnod , int     *lboud ,
1820                double  *boudv ){
1821
1822     //-----
1823     // Routine to read the mesh file
1824     //-----
1825
1826     int nnodp=9 ;
1827     int ipoin ;
1828     int ielem ;
1829     int itotv ;
1830     int temp ;
1831
1832     // open the mesh file to read it
1833
1834     ifstream Infile("Input_mesh.mesh");
1835
1836     Infile >> melem >> mpoin >> mtotv >> kappa ;
1837
1838     // [1] Read the mesh coordinates
1839
1840     for( ipoin = 0 ; ipoin < mpoin ; ipoin++ ){

```

```

1841     Infile >> cordx[ipoin] >> cordy[ipoin] ;
1842 }
1843
1844 // [2] Read the mesh connectivity
1845
1846 for ( ielem = 0 ; ielem < melem ; ielem++ ){
1847     Infile >> temp ;
1848     Infile >> lnodes[(ielem*nnodp)+0]>>lnodes[(ielem*nnodp)+1]>>lnodes[(ielem*nnodp)+2]
1849     >>lnodes[(ielem*nnodp)+3]>>lnodes[(ielem*nnodp)+4]>>lnodes[(ielem*nnodp)+5]
1850     >>lnodes[(ielem*nnodp)+6]>>lnodes[(ielem*nnodp)+7]>>lnodes[(ielem*nnodp)+8];
1851 }
1852
1853 // [3] Read the element with boundary condition
1854
1855 for ( ielem = 0 ; ielem < melem ; ielem++ ){
1856     Infile >> bcnod[(ielem*nnodp)+0]>>bcnod[(ielem*nnodp)+1]>>bcnod[(ielem*nnodp)+2]
1857     >>bcnod[(ielem*nnodp)+3]>>bcnod[(ielem*nnodp)+4]>>bcnod[(ielem*nnodp)+5]
1858     >>bcnod[(ielem*nnodp)+6]>>bcnod[(ielem*nnodp)+7]>>bcnod[(ielem*nnodp)+8];
1859 }
1860
1861 // [4] Read the boundary condition label and value
1862
1863 for ( itotv = 0 ; itotv < mtotv ; itotv++ ){
1864     Infile >> lboud[itotv] >> boudv[itotv] ;
1865 }
1866
1867 // close the mesh file
1868 Infile.close() ;
1869
1870 }
1871
1872 void Tec360_mesh( int melem , int mpoin , double cordx[] ,
1873                 double cordy[] , int lnodes[] ){
1874
1875     int ipoin , jpoin , ielem ;
1876     int nnodp = 9 ;
1877
1878     /******
1879     /* Set the file head title
1880     /******
1881
1882     ofstream Oufile("Mesh2D.plt");
1883
1884     Oufile.setf(ios::fixed) ;
1885     Oufile.precision(15) ;
1886
1887     /******
1888     Oufile << " Variables = X , Y " << endl;
1889     Oufile << " Zone N = " << mpoin << " E = " << 4 * melem << endl;
1890     Oufile << " Datapacking=point, Zonetype=fequadrilateral" << endl;
1891     /******
1892
1893     for ( ipoin = 0 ; ipoin < mpoin ; ipoin++ ){
1894         Oufile << cordx[ipoin] << " " << cordy[ipoin] << endl ;
1895     }
1896
1897     for ( ielem = 0 ; ielem < melem ; ielem++ ){
1898
1899         Oufile << lnodes[ielem*nnodp+0] << " " << lnodes[ielem*nnodp+1] << " "
1900         << lnodes[ielem*nnodp+8] << " " << lnodes[ielem*nnodp+7] << endl;
1901
1902         Oufile << lnodes[ielem*nnodp+1] << " " << lnodes[ielem*nnodp+2] << " "
1903         << lnodes[ielem*nnodp+3] << " " << lnodes[ielem*nnodp+8] << endl;
1904
1905         Oufile << lnodes[ielem*nnodp+8] << " " << lnodes[ielem*nnodp+3] << " "
1906         << lnodes[ielem*nnodp+4] << " " << lnodes[ielem*nnodp+5] << endl;
1907
1908         Oufile << lnodes[ielem*nnodp+7] << " " << lnodes[ielem*nnodp+8] << " "
1909         << lnodes[ielem*nnodp+5] << " " << lnodes[ielem*nnodp+6] << endl;
1910     }
1911     Oufile.close();
1912 }
1913
1914 void Tec360_CDR( int melem , int mpoin, double cordx[], double cordy[],
1915                 double eqphi[], int lnodes[] ){
1916
1917     int ipoin, jpoin, ielem ;
1918     int nnodp = 9 ;
1919
1920     /******
1921     /* Set the file head title
1922     /******
1923
1924     ofstream Oufile("CDR2D_FEM.plt");
1925
1926     Oufile.setf(ios::fixed) ;
1927     Oufile.precision(8) ;
1928
1929     /******
1930     Oufile << " VARIABLES = X, Y, U " << endl;
1931     Oufile << " ZONE N = " << mpoin << " E = " << 4 * melem << endl;
1932     Oufile << " DATAPACKING=POINT, ZONETYPE=FEQUADRILATERAL" << endl;
1933     /******
1934
1935     for ( ipoin = 0 ; ipoin < mpoin ; ipoin++ ){
1936         Oufile << cordx[ipoin] << " " << cordy[ipoin] << " "
1937         << eqphi[ipoin] << endl ;
1938     }
1939
1940     for ( ielem = 0 ; ielem < melem ; ielem++ ){
1941
1942         Oufile << lnodes[ielem*nnodp+0] << " " << lnodes[ielem*nnodp+1] << " "
1943         << lnodes[ielem*nnodp+8] << " " << lnodes[ielem*nnodp+7] << endl;
1944
1945         Oufile << lnodes[ielem*nnodp+1] << " " << lnodes[ielem*nnodp+2] << " "
1946         << lnodes[ielem*nnodp+3] << " " << lnodes[ielem*nnodp+8] << endl;
1947
1948         Oufile << lnodes[ielem*nnodp+8] << " " << lnodes[ielem*nnodp+3] << " "
1949         << lnodes[ielem*nnodp+4] << " " << lnodes[ielem*nnodp+5] << endl;
1950
1951         Oufile << lnodes[ielem*nnodp+7] << " " << lnodes[ielem*nnodp+8] << " "
1952         << lnodes[ielem*nnodp+5] << " " << lnodes[ielem*nnodp+6] << endl;
1953     }
1954     Oufile.close();
1955 }

```

```

1956
1957 //+++++
1958 //+++++
1959
1960 void Setting_data( int      melem , int      mpoin , int      mtotv , double kappa ,
1961                   double *cordx , double *cordy , double *uvely , double *vvely ,
1962                   int      *lnods , int      *bcnod , int      *reabc , int      *nadm ,
1963                   int      *nodfm , double *shad1 , double *shad2 , int      *lboud ,
1964                   double *boudv ){
1965
1966 //-----
1967 // Routine to
1968 //-----
1969
1970 // Read the mesh data
1971
1972 Read_mesh( melem , mpoin , mtotv , kappa , cordx ,
1973            cordy , lnods , bcnod , lboud , boudv ) ;
1974
1975 // Set the degree of freedom
1976
1977 Find_FEM_nadm_nodfm( melem , mpoin , nadm , nodfm ) ;
1978
1979 // Set the all data for finite element calculation
1980
1981 Drive1st( melem , mpoin , cordx , cordy , lnods , shad1 ) ;
1982
1983 //Drive2nd( melem , mpoin , cordx , cordy , lnods , shad2 ) ;
1984
1985 Drive2nd_02( melem , mpoin , cordx , cordy , lnods , shad2 ) ;
1986
1987 // Find the degree of repeated node
1988
1989 Find_Reabc( melem , mpoin , bcnod , reabc ) ;
1990
1991 }
1992
1993 //+++++
1994 //+++++
1995
1996 void Solve( int      melem , int      mpoin , int      mtotv , double *cordx ,
1997            double *cordy , double *uvely , double *vvely , double kappa ,
1998            int      *lnods , int      *nadm , int      *nodfm , int      *bcnod ,
1999            int      *lboud , double *boudv , double *shad1 , double *shad2 ,
2000            double *matmx , double *eqphi , double *eqrhs ) {
2001
2002 //-----
2003 // Routine to solve 2D convection-diffusion equation
2004 // using finite element method
2005 //-----
2006
2007 int nnodp = 9 ;
2008
2009 int inodp ;
2010 int ipoin ;
2011 int itotv ;
2012 int ielem ;
2013
2014 double pi = 3.141592653589793238462 ;
2015
2016 double elexx[nnodp] ;
2017 double eleyy[nnodp] ;
2018 double elecex[nnodp] ;
2019 double elecyc[nnodp] ;
2020 double lorhs[nnodp] ;
2021
2022 // Compute the convection field
2023
2024 for ( ipoin = 0 ; ipoin < mpoin ; ipoin++ ){
2025     uvely[ipoin] = cos(pi/6.0) ;
2026     vvely[ipoin] = sin(pi/6.0) ;
2027 }
2028
2029 // Compute the all elementry matrices
2030
2031 for ( ielem = 0 ; ielem < melem ; ielem++ ) { // start ielem-loop
2032
2033     for ( inodp = 0 ; inodp < nnodp ; inodp++ ){ // start inodp-loop
2034         elexx[inodp] = cordx[lnods[ielem*nnodp+inodp]-1] ;
2035         eleyy[inodp] = cordy[lnods[ielem*nnodp+inodp]-1] ;
2036         elecex[inodp] = uvely[lnods[ielem*nnodp+inodp]-1] ;
2037         elecyc[inodp] = vvely[lnods[ielem*nnodp+inodp]-1] ;
2038     } // end inodp-loop
2039
2040     //
2041     // Using classical Galerkin model
2042     //
2043
2044     /*
2045     Matrix_Galerkin( ielem , cordx , cordy , elexx , eleyy ,
2046                     elecex , elecyc , kappa , nadm , nodfm ,
2047                     lboud , shad1 , matmx , lorhs ) ;
2048
2049     */
2050
2051     //
2052     // Using DRP-SUPG model
2053     //
2054
2055     Matrix_DRP_SUPG( ielem , cordx , cordy , elexx , eleyy ,
2056                     elecex , elecyc , kappa , nadm , nodfm ,
2057                     lboud , shad1 , shad2 , matmx , lorhs ) ;
2058
2059 } // end ielem-loop
2060
2061 // Invoke the boundary condition
2062 for ( itotv = 0 ; itotv < mtotv ; itotv++ ){ // start itotv-loop
2063     if ( lboud[itotv] == 1 ) {
2064         eqrhs[itotv] = boudv[itotv] ;
2065     }
2066 } // end itotv-loop
2067
2068 // Solve the linear system using GPBI-CG solver
2069
2070 gpbicg( melem , mpoin , mtotv , matmx , eqphi ,
2071         eqrhs , nadm , nodfm , lnods ) ;

```

2071
2072 }
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088

```

1
2 //-----
3 // Program to solve the linear system
4 //-----
5
6 double dot( int mpoin , double *veca , double *vecb ){
7
8     //-----
9     // Routine to compute the inner-product
10    // <veca,vecb>
11    //-----
12
13    int ipoin ;
14    double inner ;
15
16    inner = 0.0 ;
17
18    for ( ipoin = 0 ; ipoin < mpoin ; ipoin++ ){
19        inner += veca[ipoin] * vecb[ipoin] ;
20    }
21    return inner ;
22 }
23
24 //+++++
25 //+++++
26
27 void spmv_ebe( int melem , double *v , double *av , int *lnods,
28               double *matmx , int *nadfm , int *nodfm ){
29
30    //-----
31    // Routine to compute matrix-vector product
32    // using EBE technique
33    //-----
34
35    int nnodp = 9 ;
36    int nevab = 9 ;
37    int locel[9] ;
38    int msize = 81 ;
39
40    int ibufr , kelem , kevab , inodp , jnodp , kpoin ,
41        iadfm , lodfm , iodfm , irow , krow , jcol ,
42        kcol , ievab , jevab , k ;
43
44    double flumx[nevab][nevab] ;
45
46    ibufr = 0 ;
47
48    for ( kelem = 0 ; kelem < melem ; kelem++ ){
49
50        // Get the kelem-th element matrix data
51
52        k = 0 ;
53
54        for ( ievab = 0 ; ievab < nevab ; ievab++ ){
55            for ( jevab = 0 ; jevab < nevab ; jevab++ ){
56                flumx[ievab][jevab] = matmx[kelem*msize+k] ;
57                k += 1 ;
58            }
59        }
60
61        for ( k = 0 ; k < nnodp ; k++ ){
62            locel[k] = 0 ;
63        }
64
65        // Creat global DOF array for each local element DOF
66
67        kevab = 0 ;
68
69        for ( inodp = 0 ; inodp < nnodp ; inodp++ ){
70            kpoin = lnods[kelem*nnodp+inodp] ;
71            iadfm = nadfm[kpoin-1] ;
72            lodfm = nodfm[kpoin-1] ;
73            //-----
74            for ( iodfm = 0 ; iodfm < lodfm ; iodfm++ ){
75                locel[kevab] = iadfm + iodfm ;
76                kevab += 1 ;
77            } // end iodfm loop
78            //-----
79        } // end inodp loop
80
81        // Element-by-Element procedure
82
83        for ( irow = 0 ; irow < nevab ; irow++ ){
84            krow = locel[irow]-1 ;
85            for ( jcol = 0 ; jcol < kevab ; jcol++ ){
86                kcol = locel[jcol]-1 ;
87                av[krow] = av[krow] + flumx[irow][jcol] * v[kcol] ;
88            }
89        } // end irow loop
90    } // end kelem loop
91 }
92
93 //+++++
94 //+++++
95
96 void gpbicg( int melem , int mpoin , int mtotv , double *matmx ,
97             double *egphi , double *eqrhs , int *nadfm , int *nodfm ,
98             int *lnods ){
99
100    //-----
101    // Routine to solve the Ax=b using GPBICG solver
102    // Ref :
103    // S.L.Zhang, GPBi-CG: Generalized product-type
104    // methods based on Bi-CG for solving nonsymmetric
105    // linear system, SIAM J. Sci. Comput. Vol.18, No.2
106    // pp.537-551, 1997.
107    //-----
108
109    int i , j , k , n ;
110
111    int iiter ;
112
113    double x0[mtotv] ;
114    double rs[mtotv] ;
115    double rk[mtotv] ;

```

```

116 double tk[mtotv] ;
117 double yk[mtotv] ;
118 double uk[mtotv] ;
119 double zk[mtotv] ;
120 double wk[mtotv] ;
121 double pk[mtotv] ;
122
123 double rk_old[mtotv] ;
124 double tk_old[mtotv] ;
125
126 double Apk[mtotv] ;
127 double ATK[mtotv] ;
128 double Ax0[mtotv] ;
129
130 //-----
131
132 double alpha, beta, eta, zeta ;
133
134 double norm2 ;
135 double bnorm ;
136 double toler = 1e-12 ;
137
138 n = mtotv ;
139
140 for ( i = 0 ; i < mtotv ; i++ ) x0[i] = 1e-16 ;
141 for ( i = 0 ; i < mtotv ; i++ ) Ax0[i] = 0.0 ;
142
143 spmv_ebe( melem , x0 , Ax0 , lnods , matmx , nadfm , nodfm ) ;
144
145 for ( i = 0 ; i < mtotv ; i++ ) {
146     rk[i] = egrhs[i] - Ax0[i] ;
147     rs[i] = rk[i] ;
148     wk[i] = 0.0 ;
149     tk[i] = 0.0 ;
150     pk[i] = 0.0 ;
151     uk[i] = 0.0 ;
152     yk[i] = 0.0 ;
153     zk[i] = 0.0 ;
154 }
155
156 beta = 0.0 ;
157 bnorm = 0.0 ;
158
159 for ( i = 0 ; i < mtotv ; i++ ){
160     bnorm += egrhs[i] * egrhs[i] ;
161 }
162
163 bnorm = sqrt(bnorm/mtotv) ;
164
165 /*
166  * GPBI-CG iteration loop
167  */
168
169 iiter = 0 ;
170
171 while(1) {
172
173     iiter += 1 ;
174
175     for ( i = 0 ; i < mtotv ; i++ ){
176         pk[i] = rk[i] + beta * ( pk[i] - uk[i] ) ;
177     }
178
179     // Compute Apk
180
181     for ( i = 0 ; i < mtotv ; i++ ) Apk[i] = 0.0 ;
182
183     spmv_ebe( melem , pk , Apk , lnods , matmx , nadfm , nodfm ) ;
184
185     // Compute alpha
186
187     alpha = dot(n,rs,rk)/dot(n,rs,Apk);
188
189     // Compute yk
190
191     for ( i = 0 ; i < mtotv ; i++ ){
192         yk[i] = tk[i] - rk[i] - alpha * wk[i] + alpha * Apk[i] ;
193     }
194
195     // Compute tk
196
197     for ( i = 0 ; i < mtotv ; i++ ) tk_old[i] = tk[i] ;
198
199     for ( i = 0 ; i < mtotv ; i++ ){
200         tk[i] = rk[i] - alpha * Apk[i];
201     }
202
203     // Compute ATK
204
205     for ( i = 0 ; i < mtotv ; i++ ) ATK[i] = 0.0 ;
206
207     spmv_ebe( melem , tk , ATK , lnods , matmx , nadfm , nodfm ) ;
208
209     //-----
210
211     if ( iiter == 1 ) {
212
213         // Compute zeta
214         zeta = dot(n,ATK,tk)/dot(n,ATK,ATK) ;
215
216         // Compute eta
217         eta = 0.0 ;
218     }
219     else
220     {
221         // Compute zeta
222         zeta = (dot(n,yk , yk)*dot(n,ATK,tk)-dot(n,yk, tk)*dot(n,ATK,yk))/
223             (dot(n,ATK,ATK)*dot(n, yk,yk)-dot(n,yk,ATK)*dot(n,ATK,yk)*1.0);
224
225         // Compute eta
226         eta = (dot(n,ATK,ATK)*dot(n,yk,tk)-dot(n,yk,ATK)*dot(n,ATK,tk))/
227             (dot(n,ATK,ATK)*dot(n,yk,yk)-dot(n,yk,ATK)*dot(n,ATK,yk)*1.0);
228     }
229
230     // Compute uk

```

```

231 for ( i = 0 ; i < mtotv ; i++ ){
232     uk[i] = zeta * Apk[i] + eta * ( tk_old[i] - rk[i] + beta * uk[i] ) ;
233 }
234
235 // Compute zk
236
237 for ( i = 0 ; i < mtotv ; i++ ){
238     zk[i] = zeta * rk[i] + eta * zk[i] - alpha * uk[i] ;
239 }
240
241 // Update the solution
242
243 for ( i = 0 ; i < mtotv ; i++ ){
244     eqphi[i] = eqphi[i] + alpha * pk[i] + zk[i] ;
245 }
246
247 // Compute the residual rk
248
249 for ( i = 0 ; i < mtotv ; i++ ) rk_old[i] = rk[i] ;
250
251 for ( i = 0 ; i < mtotv ; i++ ){
252     rk[i] = tk[i] - eta * yk[i] - zeta * Atk[i] ;
253 }
254
255 // Compute beta
256
257 beta = (alpha/zeta)*(dot(n,rs,rk)/dot(n,rs,rk_old)) ;
258
259 // Compute wk
260
261 for ( i = 0 ; i < mtotv ; i++ ) wk[i] = Atk[i] + beta * Apk[i] ;
262
263 // Compute the L2 notm and check the convergence !!
264
265 norm2 = 0.0 ;
266
267 for ( i = 0 ; i < mtotv ; i++ ){
268     norm2 += rk[i] * rk[i] ;
269 }
270
271 norm2 = sqrt(norm2/mtotv)/bnorm ;
272
273 //-----
274 if ( iiter % 10 == 0 ) {
275     cout << " ITERS = " << iiter << ", ||rk||_L2 = " << norm2 << endl;
276 }
277 //-----
278 if ( norm2 < toler ) {
279     cout << endl ;
280     cout << " ----- " << endl ;
281     cout << "         GP-BICG converges !!! " << endl;
282     cout << " ----- " << endl ;
283     cout << endl ;
284     break ;
285 }
286
287 } // end GPBICG-solver
288
289
290 }
291
292

```