

```
1 #include <sys/time.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <omp.h>
5 #include <math.h>
6 #include "../includes/libpoisson.h"
7
8 /* Função que retorna o tempo em função do relógio */
9 double walltime( double *t0 )
10 {
11
12     double mic, time;
13     double mega = 0.000001;
14     struct timeval tp;
15     struct timezone tzp;
16     static long base_sec = 0;
17     static long base_usec = 0;
18
19     (void) gettimeofday(&tp,&tzp);
20
21     if (base_sec == 0)
22     {
23         base_sec = tp.tv_sec;
24         base_usec = tp.tv_usec;
25     }
26
27     time = (double) (tp.tv_sec - base_sec);
28     mic = (double) (tp.tv_usec - base_usec);
29     time = (time + mic * mega) - *t0;
30     return(time);
31 }
32
33 void errorMsg(char error_text[])
34 /* Standard error handler */
35 {
36     fprintf(stderr,"Run-time error...\n");
37     fprintf(stderr,"%s\n",error_text);
38     fprintf(stderr,"...now exiting to system...\n");
39     exit(1);
40 }
41
42 /*Função que aloca as matrizes de nós, como arrays, de tamanho linXcol*/
43 nodeSides* criaVetorNode(int lin,int col)
44 {
45     nodeSides* v_aux;
46
47     v_aux = (nodeSides*) malloc(lin*col*sizeof(nodeSides));
48
49     if(v_aux==NULL)
50         errorMsg("allocation failure in vector");
51
52     return v_aux;
53 }
54
55 /*Função que aloca as matrizes de nós de tamanho linXcol*/
56 nodeSides** criaMatrizNode(int lin,int col, nodeSides* v_aux)
57 {
58     int i;
59     nodeSides** p_aux;
60
61     p_aux = (nodeSides**) malloc(lin*sizeof(nodeSides*));
62
63     if(p_aux==NULL)
64         errorMsg("allocation failure in vector");
65
66     for(i=0; i<lin; i++)
67         p_aux[i] = (nodeSides*) &v_aux[i*col];
68
69     return p_aux;
70 }
```

```
71
72 /*Função que aloca as matrizes de nós de tamanho linXcol*/
73 nodeSides** montaMatrizNode(int lin,int col)
74 {
75     int i;
76     nodeSides** p_aux;
77
78     p_aux = (nodeSides**) malloc(lin*sizeof(nodeSides*));
79
80     if(p_aux==NULL)
81         errorMsg("allocation failure in vector");
82
83     for(i=0; i<lin; i++)
84     {
85         p_aux[i] = (nodeSides*) malloc(col*sizeof(nodeSides));
86         if(p_aux[i]==NULL)
87             errorMsg("allocation failure in vector");
88     }
89
90     return p_aux;
91 }
92
93 /*Função que aloca as matrizes, como arrays, de tamanho linXcol*/
94 double* criaVetor(int lin,int col)
95 {
96     double* v_aux;
97
98     v_aux = (double*) malloc(lin*col*sizeof(double));
99
100    if(v_aux==NULL)
101        errorMsg("allocation failure in vector");
102
103    return v_aux;
104 }
105 }
106
107 /*Função que aloca as matrizes de tamanho linXcol*/
108 double** criaMatriz(int lin,int col, double* v_aux)
109 {
110     int i;
111     double** p_aux;
112
113     p_aux = (double**) malloc(lin*sizeof(double*));
114
115     if(p_aux==NULL)
116         errorMsg("allocation failure in vector");
117
118     for(i=0; i<lin; i++)
119         p_aux[i] = (double*) &v_aux[i*col];
120
121     return p_aux;
122 }
123
124 /*Função que aloca as matrizes de tamanho linXcol*/
125 double** montaMatriz(int lin,int col)
126 {
127     int i;
128     double** p_aux;
129
130     p_aux = (double**) malloc(lin*sizeof(double*));
131
132     if(p_aux==NULL)
133         errorMsg("allocation failure in vector");
134
135     for(i=0; i<lin; i++)
136     {
137         p_aux[i] = (double*) malloc(col*sizeof(double));
138         if(p_aux[i]==NULL)
139             errorMsg("allocation failure in vector");
140     }
```

```
141
142     return p_aux;
143 }
144
145 /*Função que aloca a matriz de parametros materiais, como
146 *arrays, de tamanho linXcol
147 */
148 nodeMaterial* criaVetorMaterial(int lin,int col)
149 {
150     nodeMaterial* v_aux;
151
152     v_aux = (nodeMaterial*) malloc(lin*col*sizeof(nodeMaterial));
153
154     if(v_aux==NULL)
155         errorMsg("allocation failure in dvector()");
156
157     return v_aux;
158 }
159
160 /*Função que aloca a matriz de parametros materiais
161 *de tamanho linXcol
162 */
163 nodeMaterial** criaMatrizMaterial(int lin,int col, nodeMaterial* v_aux)
164 {
165     int i;
166     nodeMaterial** p_aux;
167
168     p_aux = (nodeMaterial**) malloc(lin*sizeof(nodeMaterial*));
169
170     if(p_aux==NULL)
171         errorMsg("allocation failure in dvector()");
172
173     for(i=0; i<lin; i++)
174         p_aux[i] = (nodeMaterial*) &v_aux[i*col];
175
176     return p_aux;
177 }
178
179 /*Função que aloca a matriz de parametros materiais
180 *de tamanho linXcol
181 */
182 nodeMaterial** montaMatrizMaterial(int lin,int col)
183 {
184     int i;
185     nodeMaterial** p_aux;
186
187     p_aux = (nodeMaterial**) malloc(lin*sizeof(nodeMaterial*));
188
189     if(p_aux==NULL)
190         errorMsg("allocation failure in dvector()");
191
192     for(i=0; i<lin; i++)
193     {
194         p_aux[i] = (nodeMaterial*) malloc(col*sizeof(nodeMaterial));
195         if(p_aux[i]==NULL)
196             errorMsg("allocation failure in vector");
197     }
198
199     return p_aux;
200 }
201
202 /* Função para o canto inferior esquerdo*/
203 void canto_d_l(const int i, const int j,
204               nodeMaterial **pMat,
205               nodeSides **beta,
206               nodeSides **q,
207               nodeSides **q_old,
208               nodeSides **l_old,
209               double **p)
210
```

```

211 {
212     register double shi, AuxU,AuxR,DU,DR;          /* Auxiliares para cada lado das células
213     */
214     AuxU = pMat[i][j].shi/(1+beta[i][j].up*pMat[i][j].shi);
215     AuxR = pMat[i][j].shi/(1+beta[i][j].rh*pMat[i][j].shi);
216     DU = AuxU*(beta[i][j].up*q_old[i][j+1].dn+l_old[i][j+1].dn);
217     DR = AuxR*(beta[i][j].rh*q_old[i+1][j].lf+l_old[i+1][j].lf);
218     shi = (pMat[i][j].f + DU + DR)/(AuxU + AuxR);
219     q[i][j].up = AuxU*shi - DU;
220     q[i][j].rh = AuxR*shi - DR;
221     p[i][j] = shi;
222 }
223
224 /* Função para o canto inferior esquerdo*/
225 void canto_d_lArray(const int i, const int j, const int N,
226                     nodeMaterial *pMat,
227                     nodeSides *beta,
228                     nodeSides *q,
229                     nodeSides *q_old,
230                     nodeSides *l_old,
231                     double *p)
232
233 {
234     register double shi, AuxU,AuxR,DU,DR;          /* Auxiliares para cada lado das células
235     */
236     register int k = i*N + j;
237
238     shi = pMat[k].shi;
239     AuxU = shi/(1+beta[k].up*shi);
240     AuxR = shi/(1+beta[k].rh*shi);
241     DU = AuxU*(beta[k].up*q_old[k+1].dn+l_old[k+1].dn);
242     DR = AuxR*(beta[k].rh*q_old[k+N].lf+l_old[k+N].lf);
243     p[k] = shi = (pMat[k].f + DU + DR)/(AuxU + AuxR);
244     q[k].up = AuxU*shi - DU;
245     q[k].rh = AuxR*shi - DR;
246 }
247 /* Função para o canto superior esquerdo*/
248 void canto_u_l(const int i, const int j, nodeMaterial **pMat,
249               nodeSides **beta,
250               nodeSides **q,
251               nodeSides **q_old,
252               nodeSides **l_old,
253               double **p)
254 {
255     register double shi,AuxD, AuxR,DD, DR;          /* Auxiliares para cada lado das células
256     */
257     AuxD = pMat[i][j].shi/(1+beta[i][j].dn*pMat[i][j].shi);
258     AuxR = pMat[i][j].shi/(1+beta[i][j].rh*pMat[i][j].shi);
259     DD = AuxD*(beta[i][j].dn*q_old[i][j-1].up+l_old[i][j-1].up);
260     DR = AuxR*(beta[i][j].rh*q_old[i+1][j].lf+l_old[i+1][j].lf);
261     shi = (pMat[i][j].f + DD + DR)/(AuxD + AuxR);
262     q[i][j].dn = AuxD*shi - DD;
263     q[i][j].rh = AuxR*shi - DR;
264     p[i][j] = shi;
265 }
266
267 /* Função para o canto superior esquerdo*/
268 void canto_u_lArray(const int i, const int j, const int N,
269                     nodeMaterial *pMat,
270                     nodeSides *beta,
271                     nodeSides *q,
272                     nodeSides *q_old,
273                     nodeSides *l_old,
274                     double *p)
275 {
276     register double shi,AuxD, AuxR,DD, DR;          /* Auxiliares para cada lado das células
277     */

```

```

277     register int k = i*N + j;
278
279     shi = pMat[k].shi;
280     AuxD = shi/(1+beta[k].dn*shi);
281     AuxR = shi/(1+beta[k].rh*shi);
282     DD = AuxD*(beta[k].dn*q_old[k-1].up+l_old[k-1].up);
283     DR = AuxR*(beta[k].rh*q_old[k+N].lf+l_old[k+N].lf);
284     p[k] = shi = (pMat[k].f + DD + DR)/(AuxD + AuxR);
285     q[k].dn = AuxD*shi - DD;
286     q[k].rh = AuxR*shi - DR;
287 }
288
289 /* Função para o canto inferior direito*/
290 void canto_d_r(const int i, const int j,      nodeMaterial **pMat,
291               nodeSides **beta,
292               nodeSides **q,
293               nodeSides **q_old,
294               nodeSides **l_old,
295               double **p)
296 {
297
298     register double shi, AuxU, AuxL,DU,DL;      /* Auxiliares para cada lado das células
299 */
300     AuxU = pMat[i][j].shi/(1+beta[i][j].up*pMat[i][j].shi);
301     AuxL = pMat[i][j].shi/(1+beta[i][j].lf*pMat[i][j].shi);
302     DU = AuxU*(beta[i][j].up*q_old[i][j+1].dn+l_old[i][j+1].dn);
303     DL = AuxL*(beta[i][j].lf*q_old[i-1][j].rh+l_old[i-1][j].rh);
304     shi = (pMat[i][j].f + DU + DL)/(AuxU + AuxL);
305     q[i][j].up = AuxU*shi - DU;
306     q[i][j].lf = AuxL*shi - DL;
307     p[i][j] = shi;
308 }
309
310 /* Função para o canto inferior direito*/
311 void canto_d_rArray(const int i, const int j, const int N,
312                   nodeMaterial *pMat,
313                   nodeSides *beta,
314                   nodeSides *q,
315                   nodeSides *q_old,
316                   nodeSides *l_old,
317                   double *p)
318 {
319
320     register double shi, AuxU, AuxL,DU,DL;      /* Auxiliares para cada lado das células
321 */
322     register int k = i*N + j;
323
324     shi = pMat[k].shi;
325     AuxU = shi/(1+beta[k].up*shi);
326     AuxL = shi/(1+beta[k].lf*shi);
327     DU = AuxU*(beta[k].up*q_old[k+1].dn+l_old[k+1].dn);
328     DL = AuxL*(beta[k].lf*q_old[k-N].rh+l_old[k-N].rh);
329     p[k] = shi = (pMat[k].f + DU + DL)/(AuxU + AuxL);
330     q[k].up = AuxU*shi - DU;
331     q[k].lf = AuxL*shi - DL;
332 }
333 /* Funcao para o canto superior dereito*/
334 void canto_u_r(const int i, const int j,      nodeMaterial **pMat,
335               nodeSides **beta,
336               nodeSides **q,
337               nodeSides **q_old,
338               nodeSides **l_old,
339               double **p)
340 {
341
342     register double shi, AuxD,AuxL,DD,DL;      /* Auxiliares para cada lado das células
343 */

```

```

344     AuxD = pMat[i][j].shi/(1+beta[i][j].dn*pMat[i][j].shi);
345     AuxL = pMat[i][j].shi/(1+beta[i][j].lf*pMat[i][j].shi);
346     DD = AuxD*(beta[i][j].dn*q_old[i][j-1].up+l_old[i][j-1].up);
347     DL = AuxL*(beta[i][j].lf*q_old[i-1][j].rh+l_old[i-1][j].rh);
348     shi = (pMat[i][j].f + DD + DL)/(AuxD + AuxL);
349     q[i][j].dn = AuxD*shi - DD;
350     q[i][j].lf = AuxL*shi - DL;
351     p[i][j] = shi;
352 }
353
354 /* Funcao para o canto superior direito*/
355 void canto_u_rArray(const int i, const int j, const int N,
356                    nodeMaterial *pMat,
357                    nodeSides *beta,
358                    nodeSides *q,
359                    nodeSides *q_old,
360                    nodeSides *l_old,
361                    double *p)
362 {
363
364     register double shi, AuxD,AuxL,DD,DL;          /* Auxiliares para cada lado das células
365     /*
366     register int k = i*N + j;
367
368     shi = pMat[k].shi;
369     AuxD = shi/(1+beta[k].dn*shi);
370     AuxL = shi/(1+beta[k].lf*shi);
371     DD = AuxD*(beta[k].dn*q_old[k-1].up+l_old[k-1].up);
372     DL = AuxL*(beta[k].lf*q_old[k-N].rh+l_old[k-N].rh);
373     p[k] = shi = (pMat[k].f + DD + DL)/(AuxD + AuxL);
374     q[k].dn = AuxD*shi - DD;
375     q[k].lf = AuxL*shi - DL;
376 }
377
378 /* Função para a fronteira superior U */
379 void fronteira_u(const int n, const int j,
380                  nodeMaterial **pMat,
381                  nodeSides **beta,
382                  nodeSides **q,
383                  nodeSides **q_old,
384                  nodeSides **l_old,
385                  double **p)
386 {
387
388     register double shi;
389     register int i;
390     register double AuxD, AuxR, AuxL, DD, DR, DL;          /* Auxiliares para cada lado das
391     /* células */
392     for (i = 2; i<n; i++)
393     {
394         shi = pMat[i][j].shi;
395         AuxL = shi/(1+beta[i][j].lf*shi);
396         AuxR = shi/(1+beta[i][j].rh*shi);
397         AuxD = shi/(1+beta[i][j].dn*shi);
398         DL = AuxL*(beta[i][j].lf*q_old[i-1][j].rh+l_old[i-1][j].rh);
399         DR = AuxR*(beta[i][j].rh*q_old[i+1][j].lf+l_old[i+1][j].lf);
400         DD = AuxD*(beta[i][j].dn*q_old[i][j-1].up+l_old[i][j-1].up);
401         shi = (pMat[i][j].f + DD + DL + DR)/(AuxD + AuxL+AuxR);
402         q[i][j].lf = AuxL*shi - DL;
403         q[i][j].rh = AuxR*shi - DR;
404         q[i][j].dn = AuxD*shi - DD;
405         p[i][j] = shi;
406     }
407 }
408
409 /* Função para a fronteira superior U */
410 void fronteira_uArray(const int N, const int j,
411                       nodeMaterial *pMat,

```

```

412         nodeSides *beta,
413         nodeSides *q,
414         nodeSides *q_old,
415         nodeSides *l_old,
416         double *p)
417 {
418
419     register double shi;
420     register int i, n = N*(N-2);
421     register double AuxD, AuxR, AuxL, DD, DR, DL; /* Auxiliares para cada lado das
422     /*
423     celulas */
424     for (i=2*N+j; i < n; i+=N)
425     {
426         shi = pMat[i].shi;
427         AuxL = shi/(1+beta[i].lf*shi);
428         AuxR = shi/(1+beta[i].rh*shi);
429         AuxD = shi/(1+beta[i].dn*shi);
430         DL = AuxL*(beta[i].lf*q_old[i-N].rh+l_old[i-N].rh);
431         DR = AuxR*(beta[i].rh*q_old[i+N].lf+l_old[i+N].lf);
432         DD = AuxD*(beta[i].dn*q_old[i-1].up+l_old[i-1].up);
433         p[i] = shi = (pMat[i].f + DD + DL + DR)/(AuxD + AuxL+AuxR);
434         q[i].lf = AuxL*shi - DL;
435         q[i].rh = AuxR*shi - DR;
436         q[i].dn = AuxD*shi - DD;
437     }
438 }
439 /* Função para a fronteira inferior D */
440 void fronteira_d(const int n, const int j,
441                 nodeMaterial **pMat,
442                 nodeSides **beta,
443                 nodeSides **q,
444                 nodeSides **q_old,
445                 nodeSides **l_old,
446                 double **p)
447 {
448     register double shi;
449     register int i;
450     register double AuxU,AuxR, AuxL,DU,DR, DL; /* Auxiliares para cada lado das celulas
451     /*
452     for (i=2; i<n; i++)
453     {
454         shi = pMat[i][j].shi;
455         AuxL = shi/(1+beta[i][j].lf*shi);
456         AuxR = shi/(1+beta[i][j].rh*shi);
457         AuxU = shi/(1+beta[i][j].up*shi);
458         DL = AuxL*(beta[i][j].lf*q_old[i-1][j].rh+l_old[i-1][j].rh);
459         DR = AuxR*(beta[i][j].rh*q_old[i+1][j].lf+l_old[i+1][j].lf);
460         DU = AuxU*(beta[i][j].up*q_old[i][j+1].dn+l_old[i][j+1].dn);
461         shi = (pMat[i][j].f + DU + DL + DR)/(AuxU + AuxL+AuxR);
462         q[i][j].lf = AuxL*shi - DL;
463         q[i][j].rh = AuxR*shi - DR;
464         q[i][j].up = AuxU*shi - DU;
465         p[i][j] = shi;
466     }
467 }
468
469 /* Função para a fronteira inferior D */
470 void fronteira_dArray(const int N, const int j,
471                      nodeMaterial *pMat,
472                      nodeSides *beta,
473                      nodeSides *q,
474                      nodeSides *q_old,
475                      nodeSides *l_old,
476                      double *p)
477 {
478     register double shi;
479     register int i, n = N*(N-2);

```

```

480     register double AuxU,AuxR, AuxL,DU,DR, DL; /* Auxiliares para cada lado das celulas
481     */
482     for (i=2*N+j; i < n; i+=N)
483     {
484         shi = pMat[i].shi;
485         AuxL = shi/(1+beta[i].lf*shi);
486         AuxR = shi/(1+beta[i].rh*shi);
487         AuxU = shi/(1+beta[i].up*shi);
488         DL = AuxL*(beta[i].lf*q_old[i-N].rh+l_old[i-N].rh);
489         DR = AuxR*(beta[i].rh*q_old[i+N].lf+l_old[i+N].lf);
490         DU = AuxU*(beta[i].up*q_old[i+1].dn+l_old[i+1].dn);
491         p[i] = shi = (pMat[i].f + DU + DL + DR)/(AuxU + AuxL+AuxR);
492         q[i].lf = AuxL*shi - DL;
493         q[i].rh = AuxR*shi - DR;
494         q[i].up = AuxU*shi - DU;
495     }
496 }
497
498 /* Função para a fronteira direita R */
499 void fronteira_r(const int i, const int n,
500                 nodeMaterial **pMat,
501                 nodeSides **beta,
502                 nodeSides **q,
503                 nodeSides **q_old,
504                 nodeSides **l_old,
505                 double **p)
506 {
507
508     register double shi;
509     register int j;
510     register double AuxU,AuxD, AuxL,DU, DD, DL; /* Auxiliares para cada lado das celulas
511     */
512     for (j=2; j<n; j++)
513     {
514         shi = pMat[i][j].shi;
515         AuxU = shi/(1+beta[i][j].up*shi);
516         AuxD = shi/(1+beta[i][j].dn*shi);
517         AuxL = shi/(1+beta[i][j].lf*shi);
518         DU = AuxU*(beta[i][j].up*q_old[i][j+1].dn+l_old[i][j+1].dn);
519         DD = AuxD*(beta[i][j].dn*q_old[i][j-1].up+l_old[i][j-1].up);
520         DL = AuxL*(beta[i][j].lf*q_old[i-1][j].rh+l_old[i-1][j].rh);
521         p[i][j] = shi = (pMat[i][j].f + DU + DL + DD)/(AuxU + AuxL+AuxD);
522         q[i][j].up = AuxU*shi - DU;
523         q[i][j].dn = AuxD*shi - DD;
524         q[i][j].lf = AuxL*shi - DL;
525     }
526 }
527
528 /* Função para a fronteira direita R */
529 void fronteira_rArray(const int i, const int N,
530                      nodeMaterial *pMat,
531                      nodeSides *beta,
532                      nodeSides *q,
533                      nodeSides *q_old,
534                      nodeSides *l_old,
535                      double *p)
536 {
537
538     register double shi;
539     register int j, n = (i+1)*N - 2 ;;
540     register double AuxU,AuxD, AuxL,DU, DD, DL; /* Auxiliares para cada lado das celulas
541     */
542     for (j=(i*N)+2; j < n; j++)
543     {
544         shi = pMat[j].shi;
545         AuxU = shi/(1+beta[j].up*shi);
546         AuxD = shi/(1+beta[j].dn*shi);
547         AuxL = shi/(1+beta[j].lf*shi);

```



```

547     DU = AuxU*(beta[j].up*q_old[j+1].dn+l_old[j+1].dn);
548     DD = AuxD*(beta[j].dn*q_old[j-1].up+l_old[j-1].up);
549     DL = AuxL*(beta[j].lf*q_old[j-N].rh+l_old[j-N].rh);
550     p[j] = shi = (pMat[j].f + DU + DL + DD)/(AuxU + AuxL+AuxD);
551     q[j].up = AuxU*shi - DU;
552     q[j].dn = AuxD*shi - DD;
553     q[j].lf = AuxL*shi - DL;
554 }
555
556 }
557
558 /* Função para a fronteira esquerda L */
559 void fronteira_l(const int i, const int n,
560                 nodeMaterial **pMat,
561                 nodeSides **beta,
562                 nodeSides **q,
563                 nodeSides **q_old,
564                 nodeSides **l_old,
565                 double **p)
566 {
567     register double shi;
568     register int j;
569     register double AuxU, AuxD, AuxR, DU, DD, DR; /* Auxiliares para cada lado das células
570 */
571     for (j=2; j<n; j++)
572     {
573         shi = pMat[i][j].shi;
574         AuxU = shi/(1+beta[i][j].up*shi);
575         AuxD = shi/(1+beta[i][j].dn*shi);
576         AuxR = shi/(1+beta[i][j].rh*shi);
577         DU = AuxU*(beta[i][j].up*q_old[i][j+1].dn+l_old[i][j+1].dn);
578         DD = AuxD*(beta[i][j].dn*q_old[i][j-1].up+l_old[i][j-1].up);
579         DR = AuxR*(beta[i][j].rh*q_old[i+1][j].lf+l_old[i+1][j].lf);
580         shi = (pMat[i][j].f + DU + DR + DD)/(AuxU + AuxR+AuxD);
581         q[i][j].up = AuxU*shi - DU;
582         q[i][j].dn = AuxD*shi - DD;
583         q[i][j].rh = AuxR*shi - DR;
584         p[i][j] = shi;
585     }
586 }
587
588 /* Função para a fronteira esquerda L */
589 void fronteira_lArray(const int i, const int N,
590                      nodeMaterial *pMat,
591                      nodeSides *beta,
592                      nodeSides *q,
593                      nodeSides *q_old,
594                      nodeSides *l_old,
595                      double *p)
596 {
597     register double shi;
598     register int j, n = (i+1)*N - 2 ;
599     register double AuxU, AuxD, AuxR, DU, DD, DR; /* Auxiliares para cada lado das células
600 */
601     for (j=(i*N)+2; j < n; j++)
602     {
603         shi = pMat[j].shi;
604         AuxU = shi/(1+beta[j].up*shi);
605         AuxD = shi/(1+beta[j].dn*shi);
606         AuxR = shi/(1+beta[j].rh*shi);
607         DU = AuxU*(beta[j].up*q_old[j+1].dn+l_old[j+1].dn);
608         DD = AuxD*(beta[j].dn*q_old[j-1].up+l_old[j-1].up);
609         DR = AuxR*(beta[j].rh*q_old[j+N].lf+l_old[j+N].lf);
610         p[j] = shi = (pMat[j].f + DU + DR + DD)/(AuxU + AuxR+AuxD);
611         q[j].up = AuxU*shi - DU;
612         q[j].dn = AuxD*shi - DD;
613         q[j].rh = AuxR*shi - DR;
614     }

```

```

615 }
616
617 /* Função para os nós internos */
618 void internos(const int n,
619             nodeMaterial **pMat,
620             nodeSides **beta,
621             nodeSides **q,
622             nodeSides **q_old,
623             nodeSides **l_old,
624             double **p)
625 {
626     register double shi;
627     register int i,j;
628     register double AuxU, AuxD, AuxR, AuxL, DU, DD, DR, DL; /* Auxiliares para cada lado
629     das células */
630
631     for (i=2; i<n; i++)
632         for (j=2; j<n; j++)
633         {
634             shi = pMat[i][j].shi;
635             AuxU = shi/(1+beta[i][j].up*shi);
636             AuxR = shi/(1+beta[i][j].rh*shi);
637             AuxD = shi/(1+beta[i][j].dn*shi);
638             AuxL = shi/(1+beta[i][j].lf*shi);
639
640             DL = AuxL*(beta[i][j].lf*q_old[i-1][j].rh+l_old[i-1][j].rh);
641             DD = AuxD*(beta[i][j].dn*q_old[i][j-1].up+l_old[i][j-1].up);
642             DR = AuxR*(beta[i][j].rh*q_old[i+1][j].lf+l_old[i+1][j].lf);
643             DU = AuxU*(beta[i][j].up*q_old[i][j+1].dn+l_old[i][j+1].dn);
644
645             shi = (pMat[i][j].f + DU + DD + DL + DR)/(AuxU + AuxD + AuxL+AuxR);
646
647             q[i][j].up = AuxU*shi - DU;
648             q[i][j].rh = AuxR*shi - DR;
649             q[i][j].dn = AuxD*shi - DD;
650             q[i][j].lf = AuxL*shi - DL;
651             p[i][j] = shi;
652         }
653 }
654 }
655
656 /* Função para os nós internos */
657 void internosArray(const int N,
658                 nodeMaterial *pMat,
659                 nodeSides *beta,
660                 nodeSides *q,
661                 nodeSides *q_old,
662                 nodeSides *l_old,
663                 double *p)
664 {
665     register double shi;
666     register int i, j,n = N-4;
667     register double AuxU, AuxD, AuxR, AuxL, DU, DD, DR, DL; /* Auxiliares para cada lado
668     das células */
669     nodeSides *q_ant, *q_pos, *q_atu, *l_ant, *l_pos, *l_atu;
670     nodeSides *beta_, *q_;
671     double *p_;
672     nodeMaterial *pMat_;
673
674     q_ant = &q_old[1];
675     q_atu = q_ant + N;
676     q_pos = q_atu + N;
677
678     l_ant = &l_old[1];
679     l_atu = l_ant + N;
680     l_pos = l_atu + N;
681
682     pMat_ = &pMat[N+1];
683     beta_ = &beta[N+1];

```

```

683     q_ = &q[N+1];
684     p_ = &p[N+1];
685
686     for (i=1; i <= n; i++)
687     {
688         q_ant += N;
689         q_atu += N;
690         q_pos += N;
691
692         l_ant += N;
693         l_atu += N;
694         l_pos += N;
695
696         pMat_ += N;
697         beta_ += N;
698         q_ += N;
699         p_ += N;
700         for (j=1; j <= n; j++)
701         {
702             shi = pMat_[j].shi;
703             AuxU = shi/(1+beta_[j].up*shi);
704             AuxR = shi/(1+beta_[j].rh*shi);
705             AuxD = shi/(1+beta_[j].dn*shi);
706             AuxL = shi/(1+beta_[j].lf*shi);
707
708             DL = AuxL*(beta_[j].lf*q_ant[j].rh+l_ant[j].rh);
709             DD = AuxD*(beta_[j].dn*q_atu[j-1].up+l_atu[j-1].up);
710             DR = AuxR*(beta_[j].rh*q_pos[j].lf+l_pos[j].lf);
711             DU = AuxU*(beta_[j].up*q_atu[j+1].dn+l_atu[j+1].dn);
712
713             p_[j] = shi = (pMat_[j].f + DU + DD + DL + DR)/(AuxU + AuxD + AuxL+AuxR);
714
715             q_[j].up = AuxU*shi - DU;
716             q_[j].rh = AuxR*shi - DR;
717             q_[j].dn = AuxD*shi - DD;
718             q_[j].lf = AuxL*shi - DL;
719         }
720     }
721 }
722 }
723
724 /* Atualização dos multiplicadores de lagrange */
725 double lagrangeUpdate(const int n,
726                       nodeSides **beta,
727                       nodeSides **q,
728                       nodeSides **q_old,
729                       nodeSides **l,
730                       nodeSides **l_old,
731                       double **p)
732 {
733     register double Media = 0.0;
734     register int i,j;
735
736     for (i=1; i<=n; i++)
737         for (j=1; j<=n; j++)
738         {
739             l[i][j].up = beta[i][j].up*(q[i][j].up + q_old[i][j+1].dn) + l_old[i][j+1].dn;
740             l[i][j].dn = beta[i][j].dn*(q[i][j].dn + q_old[i][j-1].up) + l_old[i][j-1].up;
741             l[i][j].rh = beta[i][j].rh*(q[i][j].rh + q_old[i+1][j].lf) + l_old[i+1][j].lf;
742             l[i][j].lf = beta[i][j].lf*(q[i][j].lf + q_old[i-1][j].rh) + l_old[i-1][j].rh;
743             Media += p[i][j];
744         }
745
746     Media /= (n*n);
747     return Media;
748 }
749
750 /* Atualização dos multiplicadores de lagrange */
751 double lagrangeUpdateArray(const int N,
752                           nodeSides *beta,

```

```

753         nodeSides *q,
754         nodeSides *q_old,
755         nodeSides *l,
756         nodeSides *l_old,
757         double *p)
758 {
759     register int i,j, n = N-2;
760     register double Media = 0.0;
761     nodeSides *q_ant, *q_pos, *q_atu, *l_ant, *l_pos, *l_atu;
762     nodeSides *beta_, *q_, *l_;
763     double *p_;
764
765     q_atu = &q_old[0];
766     q_ant = q_atu - N;
767     q_pos = q_atu + N;
768
769     l_atu = &l_old[0];
770     l_ant = l_atu - N;
771     l_pos = l_atu + N;
772
773     beta_ = &beta[0];
774     q_ = &q[0];
775     p_ = &p[0];
776     l_ = &l[0];
777
778     for (i=1; i <= n; i++)
779     {
780         q_ant += N;
781         q_atu += N;
782         q_pos += N;
783
784         l_ant += N;
785         l_atu += N;
786         l_pos += N;
787
788         beta_ += N;
789         q_ += N;
790         p_ += N;
791         l_ += N;
792         for (j=1; j<=n; j++)
793         {
794             l_[j].up = beta_[j].up*(q_[j].up + q_atu[j+1].dn) + l_atu[j+1].dn;
795             l_[j].dn = beta_[j].dn*(q_[j].dn + q_atu[j-1].up) + l_atu[j-1].up;
796             l_[j].rh = beta_[j].rh*(q_[j].rh + q_pos[j].lf) + l_pos[j].lf;
797             l_[j].lf = beta_[j].lf*(q_[j].lf + q_ant[j].rh) + l_ant[j].rh;
798             Media += p_[j];
799         }
800     }
801
802     Media /= (n*n);
803     return Media;
804 }
805
806 /* Impondo a média zero na distriubição de pressões
807  * e cálculo de verificação de convergência */
808
809 double mediaZero(const int n,
810                 double Media,
811                 nodeSides **l,
812                 double **p,
813                 double **p_old)
814 {
815     register double sum1, sum2, aux;
816     register int i,j;
817     sum1 = sum2 = 0.0;
818
819     for (i=1; i<=n; i++)
820         for (j=1; j<=n; j++)
821         {
822             p[i][j] -= Media;

```

```

823         l[i][j].up -= Media;
824         l[i][j].dn -= Media;
825         l[i][j].rh -= Media;
826         l[i][j].lf -= Media;
827
828         aux = p[i][j] - p_old[i][j];
829         sum1 += aux*aux;
830         sum2 += p[i][j] * p[i][j];
831     }
832
833     /*Erro relativo entre a pressão atual e anterior*/
834     return sqrt(sum1/sum2);
835 }
836
837 /* Impondo a média zero na distriuição de pressões
838  * e cálculo de verificação de convergência */
839
840 double mediaZeroArray(const int N,
841                      double Media,
842                      nodeSides *l,
843                      double *p,
844                      double *p_old)
845 {
846     register int i,j, n = N-2;
847     register double pj, sum1, sum2, aux;
848     sum1 = sum2 = 0.0;
849     nodeSides *l_;
850     double *p_, *p__;
851
852     p_ = &p[0];
853     p__ = &p_old[0];
854     l_ = &l[0];
855
856     pj = Media;
857
858     for (i=1; i <= n; i++)
859     {
860         p_ += N;
861         p__ += N;
862         l_ += N;
863         for (j=1; j<=n; j++)
864         {
865             l_[j].up -= pj;
866             l_[j].dn -= pj;
867             l_[j].rh -= pj;
868             l_[j].lf -= pj;
869             pj = p_[j] -= pj;
870
871             aux = pj - p__[j];
872             sum1 += (aux*aux);
873             sum2 += (pj*pj);
874             pj = Media;
875         }
876     }
877
878     /*Erro relativo entre a pressão atual e anterior*/
879     return sqrt(sum1/sum2);
880 }
881

```