Anexo I

Código utilzado para las simulaciones 1D:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>

#define rho 2.0

#define Potencial
//#define Exponencial

int size_vect,N,M,NSim;




/* ********************STRUCT***************** */
struct almacenaje_vecinos{
    int cum1, cum2, cum3, cum4;
}; typedef struct almacenaje_vecinos Almacenaje_vecinos;
/* ****************************************** */

/* ********************** FUNCIONES ****************** */
double random_01();
double probabilidades_ruptura(double cargas[N][M], double
   probabilidades[N][M]);
void calcula_elemento_a_romper(double probabilidades[N][M],
    int *fila, int *columna);
/********Reparto de cargas***********/
void reindexa_grietas(int matriz_indices[N][M], int *
   rotos_grieta, int colum, int fila, int *indice_nuevo);
   //Matriz_indices=grietas
void iguala_indices_2grietas(int matriz[N][M], int *
   rotos_grieta, int indice_asignado, int cambia); //Matriz
   =grietas
void iguala_indices_3grietas(int matriz[N][M], int *
   rotos_grieta, int indice_asignado, int cambia1, int
   cambia2);
```

```
29 void iguala_indices_4grietas(int matriz[N][M], int *
      rotos_grieta, int indice_asignado, int cambia1, int
      cambia2, int cambia3);
30 void encuentra_vecinos(int matriz_indices[N][M], int
      numero_vecinos[size_vect], Almacenaje_vecinos vecinos[N
      ][M]);
31 void calcula_cargas_red(double cargas_red[N][M], int
      matriz_etiquetas[N][M], Almacenaje_vecinos vecinos[N][M
      ],
32                               int rotos_grieta[size_vect],
                                    int numero_vecinos[
                                    size_vect]);
33 /***********************************/
34 void inicializa_vector(int *vector, int valor, int tamano);
35 void inicializa_cargas(double cargas[N][M]);
36 void inicializa_grietas(int grietas[N][M]);
37 void Histograma (double *,double *, int,int , double *,
      double *, double *);
38 /* ************************************************ */
39
40
41
42
43 int main ()
44 {
45     //Voy a poner aqui las modificaciones para haver TvsN
46     /*******************************/
47     int TamanoFinal,TamanoInicial;
48
49     TamanoInicial=50;
50     TamanoFinal=500;
51
52     /*******************************/
53
54     srand(time(NULL));
55
56     int i,j,Npasos,k,contador;
57
58     FILE*f;
59     char name3[128];
```

```
60      sprintf(name3, "TvsN1D(rho=5)V2.txt",NSim);
61      f=fopen(name3,"w");
62
63      for(TamanoInicial;TamanoInicial<=TamanoFinal;
           TamanoInicial=TamanoInicial*10)
64      {
65          N=TamanoInicial;
66          M=1;
67          size_vect=(int)N*M/2+1;
68          Npasos=N*M;
69          printf("\nN=M= %d\n",N);
70
71          if(TamanoInicial<5001)
72          {
73              NSim=400;
74          }
75          else
76          {
77              if(TamanoInicial<10001)
78              {
79                  NSim=150;
80              }
81              else
82              {
83                  if(TamanoInicial<50001)
84                  {
85                      NSim=30;
86                  }
87                  else
88                  {
89                      NSim=10;
90                  }
91
92              }
93
94          }
95
96
97
98
```

```
99
100         double cargas[N][M]; //Matriz de cargas
101         int grietas[N][M]; //Matriz que almazena las
               grietas
102         int rotos_grieta[size_vect];//Almacena el tama o
               de cada grieta
103         int perimetros[size_vect];//Almacena el numero de
               vecinos de cada grieta
104         double probabilidades[N][M]; //Matriz para las
               probabilidades de ruptura
105         int fila,columna; //Indices del elemento que se va
               a romper
106         int indice_nuevo; //Indice de la proxima grieta a
               indexarse
107         double delta,T; //Tiempo para cada paso y tiempo
               total
108         double v[NSim]; //Vector donde guardamos el valor
               de T para cada simulacion
109
110
111         Almacenaje_vecinos vecinos [N][M]; //Hoshen-
               Kopelman: para cada elemento asocio un vector
               con sus cuatro vecions
112
113
114         for(j=0;j<NSim;j++)
115         {
116             indice_nuevo=1;
117             inicializa_cargas(cargas);
118             inicializa_grietas(grietas);
119             for(i=0;i<size_vect;i++)
120                 rotos_grieta[i]=0;
121
122
123             T=0;
124             delta=0;
125             delta=probabilidades_ruptura(cargas,
                   probabilidades);
126             T+=delta;
127             for(i=0;i<Npasos-1;i++)//Debemos contar hasta N
```

```
                -1 porque sino estariamos asociando delta al
                 vector cuando se ha roto por completo
128             {
129
130                 calcula_elemento_a_romper(probabilidades,&
                       fila,&columna);
131
132                 //Rompo el elemento indicado
133                 cargas[fila][columna]=0;
134                 //Inicia el reparto de cargas
135                 reindexa_grietas(grietas,rotos_grieta,
                       columna,fila,&indice_nuevo);
136                 encuentra_vecinos(grietas,perimetros,
                       vecinos);
137                 calcula_cargas_red(cargas,grietas,vecinos,
                       rotos_grieta,perimetros);
138
139                 delta=0;
140                 delta=probabilidades_ruptura(cargas,
                       probabilidades);
141                 T+=delta;
142
143             }
144
145
146             /* Progreso por pantalla */
147             if(j*100%NSim==0)
148                 printf("Progreso = %d%%\n",j*100/NSim);
149             /* ********************  */
150
151             v[j]=T;
152         }
153
154
155         //Calculo el valor medio de T a s  como su error.
156
157         double media,var,err;
158         var=0;
159         media=0;
160
```

```
161
162         for(i=0;i<NSim;i++)
163         {
164             media+=v[i];
165             var+=v[i]*v[i];
166         }
167
168         media=media/(double)NSim;
169         var=var/(double)NSim;
170         var=var-media*media;
171         var=sqrt(var);
172         err=var/sqrt((double)NSim);
173
174         printf("\nN=M= %d\t<T>= %f\terr= %f\tNSim= %d\n",N,
               media,err,NSim);
175         fprintf(f," %d\t %f\t %f\n",N,media,err);
176
177         /*
178         if(TamanoInicial==100000)
179         {
180             TamanoInicial=500;
181         }
182         */
183     }
184     fclose(f);
185
186
187     /* ******************************** */
188
189
190 }
191 /* ****************REPARTO DE CARGAS*************** */
192 void inicializa_vector(int *vector, int valor, int tamano)
193 {
194     for(int i=0; i<tamano; i++){
195         vector[i]=valor;
196     }
197 }
198 void iguala_indices_2grietas(int matriz[N][M], int *
       rotos_grieta, int indice_asignado, int cambia)
```

```
199 {
200     if(cambia!=indice_asignado){    //Si es igual, no hay
           que hacer nada  No  se tendria que cumplir siempre
           que es distinto?
201         int ncambiados=0;
202         for(int i=0; i<N; i++){
203             for(int j=0; j<M; j++){
204                 if(matriz[i][j]==cambia){   // Cambiamos
                       los del  ndice  a cambiar por el  ndice
                        a asignar
205                     matriz[i][j]=indice_asignado;
206                     ncambiados++;
207                 }
208             }
209         }
210         rotos_grieta[indice_asignado]+=ncambiados;
211         rotos_grieta[cambia]=0;
212     }
213 }
214
215 void iguala_indices_3grietas(int matriz[N][M], int *
       rotos_grieta, int indice_asignado, int cambia1, int
       cambia2)
216 {
217     if(indice_asignado==cambia1){            // Si el
           primero ya es igual, no hace falta reindexarlo
218         iguala_indices_2grietas(matriz, rotos_grieta,
               indice_asignado, cambia2);
219     }else if(indice_asignado==cambia2){      // Si el
           segundo ya es igual, no hace falta reindexarlo
220         iguala_indices_2grietas(matriz, rotos_grieta,
               indice_asignado, cambia1);
221     }else{
222         int ncambiados=0;
223         for(int i=0; i<N; i++){
224             for(int j=0; j<M; j++){
225                 if(matriz[i][j]==cambia1 || matriz[i][j]==
                       cambia2){
226                     matriz[i][j]=indice_asignado;
227                     ncambiados++;
```

```
228                     }
229                 }
230             }
231             rotos_grieta[indice_asignado]+=ncambiados;
232             rotos_grieta[cambia1]=0;
233             rotos_grieta[cambia2]=0;
234         }
235     }
236     void iguala_indices_4grietas(int matriz[N][M], int *
        rotos_grieta, int indice_asignado, int cambia1, int
        cambia2, int cambia3)
237     {
238         if(indice_asignado==cambia1){           // Si el
            tercero ya es el asignado no hace falta cambiarlo
239             iguala_indices_3grietas(matriz, rotos_grieta,
                indice_asignado, cambia2, cambia3);
240         }else if(indice_asignado==cambia2){     // Si el
            segundo ya es el asignado no hace falta cambiarlo
241             iguala_indices_3grietas(matriz, rotos_grieta,
                indice_asignado, cambia1, cambia3);
242         }else if(indice_asignado==cambia3){     // Si el
            tercero ya es el asignado no hace falta cambiarlo
243                 iguala_indices_3grietas(matriz, rotos_grieta,
                    indice_asignado, cambia1, cambia2);
244         }else{
245             int ncambiados=0;
246             for(int i=0; i<N; i++){
247                 for(int j=0; j<M; j++){
248                     if(matriz[i][j]==cambia1 || matriz[i][j]==
                        cambia2 || matriz[i][j]==cambia3){
249                         matriz[i][j]=indice_asignado;
250                         ncambiados++;
251                     }
252                 }
253             }
254             rotos_grieta[indice_asignado]+=ncambiados;
255             rotos_grieta[cambia1]=0;
256             rotos_grieta[cambia2]=0;
257             rotos_grieta[cambia3]=0;
258         }
```

```
259 }
260 //Nota: Matriz indices=grietas
261 void reindexa_grietas(int matriz_indices[N][M], int *
       rotos_grieta, int colum, int fila, int *indice_nuevo)
262 {
263     /*---Calculamos las coordenadas de los elementos
          adyacentes, respetando las condiciones peri dicas
          ---*/
264     int fila_up, fila_down, colum_left, colum_right;
265
266     colum_left=(colum+2*M-1)%M;     // Si colum es 0,
          colum_left=M-1; en caso contrario, colum_left=colum
          -1
267     fila_up=(fila+2*N-1)%N;         // Si fila es 0,
          fila_up=M-1; en caso contrario, fila_up=fila-1
268     colum_right=(colum+M+1)%M;      // Si colum es M-1,
          colum_right=0; en caso contrario, colum_right=colum
          +1
269     fila_down=(fila+N+1)%N;         // Si fila es N-1,
          fila_down=0; en caso contrario, fila_down=fila+1
270
271     /*---Calculamos los  ndices  de los elementos
          adyacentes---*/
272     int up, down, left, right, indice_asignado;     //
           ndice_asignado : indice de la grieta que va a
          perdurar, y el que se asocia al nuevo roto
273     int cambia;                                     //
           ndice  a sustituir por  ndice_asignado  en la
          reindexaci n
274     up=matriz_indices[fila_up][colum];
275     down=matriz_indices[fila_down][colum];
276     left=matriz_indices[fila][colum_left];
277     right=matriz_indices[fila][colum_right];
278
279     /*---Distinguimos casos  seg n  el  n mero de vecinos
          rotos---*/
280     switch (!!up + !!left + !!down + !!right) {     // '!!a
          ' devuelve 1 si a!=0, 0 si a=0. Esto da el # de
          vecinos que no son 0
281       case 0:   // Nueva grieta
```

```
282           indice_asignado=*indice_nuevo;
283           (*indice_nuevo)++; // La siguiente grieta nueva
                 estar asociada al siguiente ndice nuevo
284         break;
285         case 1:   // El elemento roto es vecino de una s la
               grieta. Se le asocia el ndice no nulo.
286           if(up!=0){
287               indice_asignado=up;
288           }else if(left!=0){
289               indice_asignado=left;
290           }else if(right!=0){
291               indice_asignado=right;
292           }else{
293               indice_asignado=down;
294           }
295         break;
296         case 2:   // El elemento roto es vecino de 2 grietas.
               Se le asocia el ndice !=0, respetando prioridad
              (up, left, right, down)
297           if(up!=0){
298               indice_asignado=up;
299               cambia=down+left+right;     // Alguno de estos
                     3 es no nulo, el resto son nulos
300           }else if(left!=0){
301               indice_asignado=left;
302               cambia=down+right;          // Uno es nulo y el
                     otro no
303           }else{
304               indice_asignado=right;
305               cambia=down;
306           }
307           iguala_indices_2grietas(matriz_indices,
                 rotos_grieta, indice_asignado, cambia); // Se
                 igualan los ndices de ambas grietas
308         break;
309         case 3:   // El elemento roto es vecino de 3 grietas.
               Se le asocia el ndice !=0, respetando prioridad
              (up, left, right, down)
310           if(up==0){
311               indice_asignado=left;
```

```
312                 iguala_indices_3grietas(matriz_indices,
                       rotos_grieta, indice_asignado, right, down);
                           // Se igualan los  ndices  de las 3
                       grietas
313             }else{
314                 indice_asignado=up;
315                 if(left==0){
316                     iguala_indices_3grietas(matriz_indices,
                           rotos_grieta, indice_asignado, right,
                           down);
317                 }else{
318                     if(right==0){
319                         iguala_indices_3grietas(matriz_indices,
                               rotos_grieta, indice_asignado, left
                              , down);
320                     }else{
321                         iguala_indices_3grietas(matriz_indices,
                               rotos_grieta, indice_asignado,
                              right, left);
322                     }
323                 }
324             }
325           break;
326           case 4:   // El elemento roto es vecino de 4 c mulos
                .Se le asocia el  ndice !=0, respetando prioridad
                (up, left, right, down)
327             indice_asignado=up;
328             iguala_indices_4grietas(matriz_indices,
                   rotos_grieta, indice_asignado, down, left, right
                   );   // Se igualan los  ndices  de las 4 grietas
329         }
330         matriz_indices[fila][colum]=indice_asignado;    // Se
               asigna el  ndice  adecuado al nuevo elemento roto
331         rotos_grieta[indice_asignado]++;                //
               Nuevo elemento para la grieta con la que e s t  en
               contacto
332     }
333     void encuentra_vecinos(int matriz_indices[N][M], int
        numero_vecinos[size_vect], Almacenaje_vecinos vecinos[N
        ][M])
```

```
334 {
335     int col_left, col_right, fila_up, fila_down;
336     int indice_up, indice_left, indice_right, indice_down;
337     inicializa_vector(numero_vecinos, 0, size_vect);
          // El n mero de vecinos se calcula cada vez que se
          ejecuta esta funci n
338
339     /*---Leemos la matriz:---*/
340     for(int i=0; i<N; i++){
341         fila_up=(i-1+2*N)%N;
342         fila_down=(i+1+N)%N;
343         for(int j=0; j<M; j++){
344             col_left=(j-1+2*M)%M;
345             col_right=(j+1+M)%M;
346             if(matriz_indices[i][j]!=0){    // Los
                  elementos ya rotos, como no tienen carga, no
                   los utilizaremos
347                 vecinos[i][j].cum1=0;
348                 vecinos[i][j].cum2=0;
349                 vecinos[i][j].cum3=0;
350                 vecinos[i][j].cum4=0;
351             }else{                          // Si no es 0,
                  no es un vecino, sino un elemento roto
352                 indice_up=matriz_indices[fila_up][j];
353                 indice_left=matriz_indices[i][col_left];
354                 indice_right=matriz_indices[i][col_right];
355                 indice_down=matriz_indices[fila_down][j];
356
357                 /* Guardamos los  ndices  de los c mulos
                      contiguos al elemento i, j.
358                  * Si alguna es 0, se guarda un 0, que al
                       no corresponder a ning n  c mulo,
                       significa
359                  * que en realidad no tiene vecino en esa
                       direcci n.
360                  *
361                  * Incrementamos el n mero de vecinos de
                       la(s) grieta(s) dada(s) por el(los)
                        ndice (s)
362                  * del (de los) elemento(s) contiguo(s).
```

```
363                      * Si alguno es 0, se incrementa el numero
                            de vecinos del "c mulo 0", que es el
                            c mulo
364                      * de los elementos no rotos y no se
                            utiliza para calcular las cargas.
365                      */
366
367                     vecinos[i][j].cum1=indice_up;
368                     numero_vecinos[indice_up]++;
369                     if(indice_left!=indice_up)
370                     {
371                         vecinos[i][j].cum2=indice_left;
372                         numero_vecinos[indice_left]++;
373                         if(indice_right!=indice_up &&
                               indice_right!=indice_left)
374                         {
375                             vecinos[i][j].cum3=indice_right;
376                             numero_vecinos[indice_right]++;
377                             if(indice_down!=indice_up &&
                                   indice_down!=indice_left &&
                                   indice_down!=indice_right) //Los
                                    4 son distintos
378                             {
379                                 vecinos[i][j].cum4=indice_down;
380                                 numero_vecinos[indice_down]++;
381                             }else             //3 son distintos
382                             {
383                                 vecinos[i][j].cum4=0;       //
                                       Si es doblemente vecino de
                                       alguno, solo lo guardamos 1
                                       vez
384                             }
385                         }else
386                         {
387                             if(indice_down!=indice_up &&
                                   indice_down!= indice_left) //3
                                   son distintos
388                             {
389                                 vecinos[i][j].cum3=indice_down;
390                                 numero_vecinos[indice_down]++;
```

```
391                             }else  //2 son distintos
392                             {
393                                 vecinos[i][j].cum3=0;
394                             }
395                             vecinos[i][j].cum4=0;
396                         }
397                     }else
398                     {
399                         if(indice_right!=indice_up){
400                             vecinos[i][j].cum2=indice_right;
401                             numero_vecinos[indice_right]++;
402                             if(indice_down!=indice_up &&
                                   indice_down!=indice_right){
403                                 vecinos[i][j].cum3=indice_down;
404                                 numero_vecinos[indice_down]++;
405                             }else
406                             {
407                                 vecinos[i][j].cum3=0;
408                             }
409                         }else
410                         {
411                             if(indice_down!=indice_up)
412                             {
413                                 vecinos[i][j].cum2=indice_down;
414                                 numero_vecinos[indice_down]++;
415                             }else
416                             {
417                                 vecinos[i][j].cum2=0;
418                             }
419                             vecinos[i][j].cum3=0;
420                         }
421                         vecinos[i][j].cum4=0;
422                     }
423                 }
424             }
425         }
426     }
427 void calcula_cargas_red(double cargas_red[N][M], int
       matriz_etiquetas[N][M], Almacenaje_vecinos vecinos[N][M
       ],
```

```
428                             int rotos_grieta[size_vect], int
                                   numero_vecinos[size_vect])
429 {
430     int etiqueta;
431     for(int i=0; i<N; i++){
432         for(int j=0; j<M; j++){
433             etiqueta=matriz_etiquetas[i][j];
434             if(etiqueta!=0){      // Elemento ya roto, no
                   soporta carga
435                 cargas_red[i][j]=0;
436             }else{                // Elemento sin romper
437                 /* Incrementamos las cargas, como la suma (
                       restringida a los vecinos no nulos),
                       mediante: r_k/s_k
438                  * siendo r el n mero de elementos rotos
                        de la grieta y s el n mero de vecinos
                        de la grieta.
439                  */
440                 cargas_red[i][j]=1;
441                 if((vecinos[i][j].cum1)!=0){
442                     cargas_red[i][j]+=((double)rotos_grieta
                           [vecinos[i][j].cum1]/(numero_vecinos
                           [vecinos[i][j].cum1]));
443                 }
444                 if((vecinos[i][j].cum2)!=0){
445                     cargas_red[i][j]+=((double)rotos_grieta
                           [vecinos[i][j].cum2]/(numero_vecinos
                           [vecinos[i][j].cum2]));
446                 }
447                 if((vecinos[i][j].cum3)!=0){
448                     cargas_red[i][j]+=((double)rotos_grieta
                           [vecinos[i][j].cum3]/(numero_vecinos
                           [vecinos[i][j].cum3]));
449                 }
450                 if((vecinos[i][j].cum4)!=0){
451                     cargas_red[i][j]+=((double)rotos_grieta
                           [vecinos[i][j].cum4]/(numero_vecinos
                           [vecinos[i][j].cum4]));
452                 }
453             }
```

```
454             }
455         }
456     }
457     /*
        ************************************************************
         */
458
459     void inicializa_cargas(double cargas[N][M])
460     {
461         int i,j;
462         for (i=0;i<N;i++)
463         {
464             for(j=0;j<M;j++)
465             {
466                 cargas[i][j]=1;
467             }
468         }
469     }
470     void inicializa_grietas(int grietas[N][M])
471     {
472         int i,j;
473         for (i=0;i<N;i++)
474         {
475             for(j=0;j<M;j++)
476             {
477                 grietas[i][j]=0;
478             }
479         }
480     }
481     #ifdef Potencial
482
483     double probabilidades_ruptura(double cargas[N][M], double
        probabilidades[N][M])
484     {
485         double gamma_total=0;       // Tiempo de vida de la
            configuracion actual
486         double gamma[N][M];         // Matriz para el calculo de
             las probabilidades
487
488         for(int i=0; i<N; i++){    // Calculo de las gammas
```

```
489             for(int j=0; j<M; j++){
490                 gamma[i][j]=pow((cargas[i][j]),rho);
491                 gamma_total+=gamma[i][j];
492             }
493         }
494         for(int i=0; i<N; i++){   // Calculo de las
                probabilidades
495             for(int j=0; j<M; j++){
496                 probabilidades[i][j]=(gamma[i][j])/(gamma_total
                    );
497             }
498         }
499 
500         return 1/gamma_total;     // Se devuelve el valor de
                delta
501     }
502 
503     #endif // Potencial
504 
505 
506     #ifdef Exponencial
507     double probabilidades_ruptura(double cargas[N][M], double
            probabilidades[N][M])
508     {
509         double gamma_total=0;       // Tiempo de vida de la
                configuracion actual
510         double gamma[N][M];         // Matriz para el calculo de
                 las probabilidades
511 
512         //Inicializo gamma
513         for(int i=0; i<N; i++){
514             for(int j=0; j<M; j++){
515                 gamma[i][j]=0;
516             }
517         }
518 
519 
520         for(int i=0; i<N; i++)//Calculo las gammas
521         {
522             for(int j=0; j<M; j++)
```

```
523             {
524                 if(cargas[i][j]!=0)
525                 {
526                     gamma[i][j]=exp(cargas[i][j]);
527                     gamma_total+=gamma[i][j];
528                 }
529             }
530         }
531 
532         for(int i=0; i<N; i++){   // Calculo de las
              probabilidades
533             for(int j=0; j<M; j++){
534                 probabilidades[i][j]=(gamma[i][j])/(gamma_total
                      );
535             }
536         }
537 
538         return 1/gamma_total;     // Se devuelve el valor de
              delta
539     }
540 
541     #endif // Exponencial
542 
543     void calcula_elemento_a_romper(double probabilidades[N][M],
          int *fila, int *columna)
544     {
545         double numero=random_01();
546         double acum_prob=0;
547         int j, i;
548 
549         /* El intervalo [0,1) se divide en segmentos, cada uno
              de una longitud
550          * igual a la probabilidad de ruptura del elemento. El
               numero entre 0 y 1
551          * determina el segmento en el que cae, al cual se le
               asocian las coordenadas,
552          * que son los parametros de salida
553          */
554         for(i=0; i<N; i++){
555             for(j=0; j<M; j++){
```

```
556                 acum_prob+=probabilidades[i][j];
557                 if(acum_prob>numero){   // El indice se
                       devuelve cuando el numero esta entre dos
                       separaciones de segmentos
558                     *fila=i;
559                     *columna=j;
560                     return;   // Para salir del bucle
561                 }
562             }
563         }
564     }
565
566     void Histograma (double *data,double *Hist, int N_data,int
           N_intervalos, double *d, double *m, double *Max)
567     {
568         /*
569         *data -> input, Datos sobre los que se genera el
               histograma
570         *Hist -> output Histograma calculado
571         N_data -> input Numero de datos
572         N_intervalos -> input, Numero de intervalos del
               histograma
573         *d -> output Medida de cada intervalo del histograma
574         *m -> output Valor minimo de los datos
575         *M -> output Valor maximo de los datos
576         */
577         int i,Indice;// Importante que el indice sea int ya que
               va a redondear al entero
578         double Norm,delta,minimo,maximo;
579         for (i=0;i<N_intervalos;i++)
580         {
581             Hist[i]=0;
582         }
583         minimo=10000000;
584         maximo=-10000000;
585         for (i=0;i<N_data;i++) //Calculo el minimo y maximo
               valor
586         {
587             if(data[i]>maximo)maximo=data[i];
588             if(data[i]<minimo)minimo=data[i];
```

```
589      }
590      delta=(maximo-minimo)/N_intervalos;
591      if(delta==0)
592      {
593          printf("No se pueden calcular los intervalos");
594          exit(1);
595      }
596      //Nucleo del programa
597      for(i=0;i<N_data;i++)
598      {
599      Indice=(data[i]-minimo)/delta;
600      Hist[Indice]=Hist[Indice]+1;
601      }
602      *d=delta;
603      *m=minimo;
604      *Max=maximo;
605      //Ahora normalizo
606
607      Norm=1.0/(N_data*delta);
608      for(i=0;i<N_intervalos;i++)
609      {
610          Hist[i]=Hist[i]*Norm;
611      }
612  }
613
614
615
616
617  /***************************** PARISI RAPUANO
         ***************/
618  #define NormParisi (2.3283063671E-10F)  //Para normalizar
         el valor generado en la rueda de Parisi-Rapuano
619
620  double random_01()
621  {
622
623      int i;
624      unsigned int rueda[256], aleatorio;
625      unsigned char indice_ran, indice1, indice2, indice3;
626      //Inicializar rueda
```

```
627     for(i = 0; i < 256; i++)
628         rueda[i] = (rand()<<16) + rand();
629     //Inicializar indices
630     indice_ran = 0; indice1 = 0; indice2 = 0; indice3 = 0;
631     //Modificamos los indices
632     indice1 = indice_ran - 24;
633     indice2 = indice_ran - 55;
634     indice3 = indice_ran - 61;
635     //Modificamos la rueda
636     rueda[indice_ran] = rueda[indice1] + rueda[indice2];
637     //Generamos un numero aleatorio entre 0 y 2^32-1
638     aleatorio = (rueda[indice_ran]^rueda[indice3]);
639     //Cambiamos la posicion base para el siguiente numero
            aleatorio
640     indice_ran++;
641     //Devolvemos el numero aleatorio normalizado, entre 0 y
            1
642     return aleatorio * NormParisi;
643 }
644 /******************************** FIN PARISI RAPUANO
        ***********/
```

Código utilizado para las simulaciones 2D (entorno de Von Neumann):

```
1
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <time.h>
5  #include <math.h>
6
7  int size_vect,N,M,NSim;
8
9  /* ******************STRUCT***************** */
10 struct almacenaje_vecinos{
11     int cum1, cum2, cum3, cum4;
12 }; typedef struct almacenaje_vecinos Almacenaje_vecinos;
13 /* ****************************************** */
14
15 /* ********************* FUNCIONES ***************** */
16 double random_01();
```

```
17 double probabilidades_ruptura(double cargas[N][M], double
      probabilidades[N][M]);
18 void calcula_elemento_a_romper(double probabilidades[N][M],
       int *fila, int *columna);
19 /********Reparto de cargas***********/
20 void reindexa_grietas(int matriz_indices[N][M], int *
      rotos_grieta, int colum, int fila, int *indice_nuevo);
21 void iguala_indices_2grietas(int matriz[N][M], int *
      rotos_grieta, int indice_asignado, int cambia);
22 void iguala_indices_3grietas(int matriz[N][M], int *
      rotos_grieta, int indice_asignado, int cambia1, int
      cambia2);
23 void iguala_indices_4grietas(int matriz[N][M], int *
      rotos_grieta, int indice_asignado, int cambia1, int
      cambia2, int cambia3);
24 void encuentra_vecinos(int matriz_indices[N][M], int
      numero_vecinos[size_vect], Almacenaje_vecinos vecinos[N
      ][M]);
25 void calcula_cargas_red(double cargas_red[N][M], int
      matriz_etiquetas[N][M], Almacenaje_vecinos vecinos[N][M
      ],
26              int rotos_grieta[size_vect], int numero_vecinos
                   [size_vect]);
27 /*********************************/
28 void inicializa_vector(int *vector, int valor, int tamano);
29 void inicializa_cargas(double cargas[N][M]);
30 void inicializa_grietas(int grietas[N][M]);
31 void Histograma (double *,double *, int,int , double *,
      double *, double *);
32 /* ********************************************** */
33
34 int main ()
35 {
36     /*******************************/
37     int TamanoFinal,TamanoInicial;
38
39     TamanoInicial=20;
40     TamanoFinal=200;
41
42     /*******************************/
```

```
43
44      srand(time(NULL));
45
46      int i,j,Npasos,k;
47
48      FILE*f;
49      char name3[128];
50      sprintf(name3, "TvsN2D(Neumann).txt",NSim);
51      f=fopen(name3,"w");
52
53      for(TamanoInicial;TamanoInicial<=TamanoFinal;
           TamanoInicial+=10)
54      {
55          NSim=2000;
56          N=TamanoInicial;
57          M=TamanoInicial;
58          size_vect=(int)N*M/2+1;
59          Npasos=N*M;
60          printf("\nN=M= %d\n",N);
61
62          if(TamanoInicial>=50)
63          {
64                  if(TamanoInicial>=100)
65                  {
66                      NSim=(int)NSim/100;
67                  }else
68                  {
69                      NSim=(int)NSim/10;
70                  }
71          }
72
73          double cargas[N][M]; //Matriz de cargas
74          int grietas[N][M]; //Matriz que almazena las
               grietas
75          int rotos_grieta[size_vect];//Almacena el tama o
               de cada grieta
76          int perimetros[size_vect];//Almacena el numero de
               vecinos de cada grieta
77          double probabilidades[N][M]; //Matriz para las
               probabilidades de ruptura
```

```
78          int fila,columna; //Indices del elemento que se va
               a romper
79          int indice_nuevo; //Indice de la proxima grieta a
               indexarse
80          double delta,T; //Tiempo para cada paso y tiempo
               total
81          double v[NSim]; //Vector donde guardamos el valor
               de T para cada simulacion
82
83
84          Almacenaje_vecinos vecinos [N][M]; //Hoshen-
               Kopelman: para cada elemento asocio un vector
               con sus cuatro vecions
85
86
87          for(j=0;j<NSim;j++)
88          {
89              indice_nuevo=1;
90              inicializa_cargas(cargas);
91              inicializa_grietas(grietas);
92              for(i=0;i<size_vect;i++)
93                  rotos_grieta[i]=0;
94
95
96              T=0;
97              delta=0;
98              delta=probabilidades_ruptura(cargas,
                   probabilidades);
99              T+=delta;
100             for(i=0;i<Npasos-1;i++)//Debemos contar hasta N
                   -1 porque sino estariamos asociando delta al
                    vector cuando se ha roto por completo
101             {
102
103                 calcula_elemento_a_romper(probabilidades,&
                       fila,&columna);
104
105                 //Rompo el elemento indicado
106                 cargas[fila][columna]=0;
107                 //Inicia el reparto de cargas
```

```
108                 reindexa_grietas(grietas,rotos_grieta,
                        columna,fila,&indice_nuevo);
109                 encuentra_vecinos(grietas,perimetros,
                        vecinos);
110                 calcula_cargas_red(cargas,grietas,vecinos,
                        rotos_grieta,perimetros);
111
112                 delta=0;
113                 delta=probabilidades_ruptura(cargas,
                        probabilidades);
114                 T+=delta;
115
116             }
117
118
119             /* Progreso por pantalla */
120             if(j*100%NSim==0)
121                 printf("Progreso = %d%%\n",j*100/NSim);
122             /* ********************  */
123
124             v[j]=T;
125         }
126
127
128         //Calculo el valor medio de T a s  como su error.
129
130         double media,var,err;
131         var=0;
132         media=0;
133
134
135         for(i=0;i<NSim;i++)
136         {
137             media+=v[i];
138             var+=v[i]*v[i];
139         }
140
141         media=media/(double)NSim;
142         var=var/(double)NSim;
143         var=var-media*media;
```

```
144             var=sqrt(var);
145             err=var/sqrt((double)NSim);
146 
147             printf("\nN=M= %d\t<T>= %f\terr= %f\tNSim= %d\n",N,
                    media,err,NSim);
148             fprintf(f," %d\t %d\t %f\t %f\n",N,(int)N*N,media,err);
149 
150 
151         }
152         fclose(f);
153 
154 
155         /* ********************************* */
156 
157 
158     }
159     /* *****************REPARTO DE CARGAS**************** */
160     void inicializa_vector(int *vector, int valor, int tamano)
161     {
162         for(int i=0; i<tamano; i++){
163             vector[i]=valor;
164         }
165     }
166     void iguala_indices_2grietas(int matriz[N][M], int *
            rotos_grieta, int indice_asignado, int cambia)
167     {
168         if(cambia!=indice_asignado){    //Si es igual, no hay
                que hacer nada  No  se tendria que cumplir siempre
                que es distinto?
169             int ncambiados=0;
170             for(int i=0; i<N; i++){
171                 for(int j=0; j<M; j++){
172                     if(matriz[i][j]==cambia){   // Cambiamos
                            los del  ndice  a cambiar por el  ndice
                             a asignar
173                         matriz[i][j]=indice_asignado;
174                         ncambiados++;
175                     }
176                 }
177             }
```

```
178             rotos_grieta[indice_asignado]+=ncambiados;
179             rotos_grieta[cambia]=0;
180         }
181     }
182
183     void iguala_indices_3grietas(int matriz[N][M], int *
        rotos_grieta, int indice_asignado, int cambia1, int
        cambia2)
184     {
185         if(indice_asignado==cambia1){            // Si el
            primero ya es igual, no hace falta reindexarlo
186             iguala_indices_2grietas(matriz, rotos_grieta,
                indice_asignado, cambia2);
187         }else if(indice_asignado==cambia2){      // Si el
            segundo ya es igual, no hace falta reindexarlo
188             iguala_indices_2grietas(matriz, rotos_grieta,
                indice_asignado, cambia1);
189         }else{
190             int ncambiados=0;
191             for(int i=0; i<N; i++){
192                 for(int j=0; j<M; j++){
193                     if(matriz[i][j]==cambia1 || matriz[i][j]==
                        cambia2){
194                         matriz[i][j]=indice_asignado;
195                         ncambiados++;
196                     }
197                 }
198             }
199             rotos_grieta[indice_asignado]+=ncambiados;
200             rotos_grieta[cambia1]=0;
201             rotos_grieta[cambia2]=0;
202         }
203     }
204     void iguala_indices_4grietas(int matriz[N][M], int *
        rotos_grieta, int indice_asignado, int cambia1, int
        cambia2, int cambia3)
205     {
206         if(indice_asignado==cambia1){           // Si el
            tercero ya es el asignado no hace falta cambiarlo
207             iguala_indices_3grietas(matriz, rotos_grieta,
```

```
            indice_asignado, cambia2, cambia3);
208     }else if(indice_asignado==cambia2){     // Si el
           segundo ya es el asignado no hace falta cambiarlo
209         iguala_indices_3grietas(matriz, rotos_grieta,
               indice_asignado, cambia1, cambia3);
210     }else if(indice_asignado==cambia3){     // Si el
           tercero ya es el asignado no hace falta cambiarlo
211             iguala_indices_3grietas(matriz, rotos_grieta,
                   indice_asignado, cambia1, cambia2);
212     }else{
213         int ncambiados=0;
214         for(int i=0; i<N; i++){
215             for(int j=0; j<M; j++){
216                 if(matriz[i][j]==cambia1 || matriz[i][j]==
                       cambia2 || matriz[i][j]==cambia3){
217                     matriz[i][j]=indice_asignado;
218                     ncambiados++;
219                 }
220             }
221         }
222         rotos_grieta[indice_asignado]+=ncambiados;
223         rotos_grieta[cambia1]=0;
224         rotos_grieta[cambia2]=0;
225         rotos_grieta[cambia3]=0;
226     }
227 }
228 //Nota: Matriz indices=grietas
229 void reindexa_grietas(int matriz_indices[N][M], int *
       rotos_grieta, int colum, int fila, int *indice_nuevo)
230 {
231     /*---Calculamos las coordenadas de los elementos
           adyacentes, respetando las condiciones peri dicas
           ---*/
232     int fila_up, fila_down, colum_left, colum_right;
233
234     colum_left=(colum+2*M-1)%M;     // Si colum es 0,
           colum_left=M-1; en caso contrario, colum_left=colum
           -1
235     fila_up=(fila+2*N-1)%N;         // Si fila es 0,
           fila_up=M-1; en caso contrario, fila_up=fila-1
```

```
236       colum_right=(colum+M+1)%M;      // Si colum es M-1,
             colum_right=0; en caso contrario, colum_right=colum
             +1
237       fila_down=(fila+N+1)%N;         // Si fila es N-1,
             fila_down=0; en caso contrario, fila_down=fila+1
238
239       /*---Calculamos los  ndices  de los elementos
             adyacentes---*/
240       int up, down, left, right, indice_asignado;     //
              ndice_asignado : indice de la grieta que va a
             perdurar, y el que se asocia al nuevo roto
241       int cambia;                                     //
              ndice  a sustituir por  ndice_asignado  en la
             reindexaci n
242       up=matriz_indices[fila_up][colum];
243       down=matriz_indices[fila_down][colum];
244       left=matriz_indices[fila][colum_left];
245       right=matriz_indices[fila][colum_right];
246
247       /*---Distinguimos casos  seg n  el  n mero de vecinos
             rotos---*/
248       switch (!!up + !!left + !!down + !!right) {     // '!!a
             ' devuelve 1 si a!=0, 0 si a=0. Esto da el # de
             vecinos que no son 0
249         case 0:   // Nueva grieta
250           indice_asignado=*indice_nuevo;
251           (*indice_nuevo)++; // La siguiente grieta nueva
                 estar  asociada al siguiente  ndice  nuevo
252         break;
253         case 1:   // El elemento roto es vecino de una  s la
              grieta. Se le asocia el  ndice  no nulo.
254           if(up!=0){
255               indice_asignado=up;
256           }else if(left!=0){
257               indice_asignado=left;
258           }else if(right!=0){
259               indice_asignado=right;
260           }else{
261               indice_asignado=down;
262           }
```

```
263         break;
264         case 2:   // El elemento roto es vecino de 2 grietas.
                Se le asocia el  ndice !=0, respetando prioridad
               (up, left, right, down)
265           if(up!=0){
266               indice_asignado=up;
267               cambia=down+left+right;     // Alguno de estos
                     3 es no nulo, el resto son nulos
268           }else if(left!=0){
269               indice_asignado=left;
270               cambia=down+right;          // Uno es nulo y el
                      otro no
271           }else{
272               indice_asignado=right;
273               cambia=down;
274           }
275           iguala_indices_2grietas(matriz_indices,
                 rotos_grieta, indice_asignado, cambia); // Se
                 igualan los  ndices  de ambas grietas
276         break;
277         case 3:   // El elemento roto es vecino de 3 grietas.
                Se le asocia el  ndice !=0, respetando prioridad
               (up, left, right, down)
278           if(up==0){
279               indice_asignado=left;
280               iguala_indices_3grietas(matriz_indices,
                     rotos_grieta, indice_asignado, right, down);
                         // Se igualan los  ndices  de las 3
                     grietas
281           }else{
282               indice_asignado=up;
283               if(left==0){
284                   iguala_indices_3grietas(matriz_indices,
                         rotos_grieta, indice_asignado, right,
                         down);
285               }else{
286                   if(right==0){
287                       iguala_indices_3grietas(matriz_indices,
                             rotos_grieta, indice_asignado, left
                            , down);
```

```
288                     }else{
289                         iguala_indices_3grietas(matriz_indices,
                                rotos_grieta, indice_asignado,
                               right, left);
290                     }
291                 }
292             }
293           break;
294           case 4:   // El elemento roto es vecino de 4 c mulos
                .Se le asocia el  ndice !=0, respetando prioridad
                (up, left, right, down)
295             indice_asignado=up;
296             iguala_indices_4grietas(matriz_indices,
                   rotos_grieta, indice_asignado, down, left, right
                   );   // Se igualan los  ndices  de las 4 grietas
297         }
298         matriz_indices[fila][colum]=indice_asignado;    // Se
              asigna el  ndice  adecuado al nuevo elemento roto
299         rotos_grieta[indice_asignado]++;                //
              Nuevo elemento para la grieta con la que e s t  en
              contacto
300     }
301     void encuentra_vecinos(int matriz_indices[N][M], int
          numero_vecinos[size_vect], Almacenaje_vecinos vecinos[N
          ][M])
302     {
303         int col_left, col_right, fila_up, fila_down;
304         int indice_up, indice_left, indice_right, indice_down;
305         inicializa_vector(numero_vecinos, 0, size_vect);
              // El n m e r o de vecinos se calcula cada vez que se
              ejecuta esta funci n
306
307         /*---Leemos la matriz:---*/
308         for(int i=0; i<N; i++){
309             fila_up=(i-1+2*N)%N;
310             fila_down=(i+1+N)%N;
311             for(int j=0; j<M; j++){
312                 col_left=(j-1+2*M)%M;
313                 col_right=(j+1+M)%M;
314                 if(matriz_indices[i][j]!=0){    // Los
```

```
                elementos ya rotos, como no tienen carga, no
                 los utilizaremos
315             vecinos[i][j].cum1=0;
316             vecinos[i][j].cum2=0;
317             vecinos[i][j].cum3=0;
318             vecinos[i][j].cum4=0;
319         }else{                          // Si no es 0,
                no es un vecino, sino un elemento roto
320             indice_up=matriz_indices[fila_up][j];
321             indice_left=matriz_indices[i][col_left];
322             indice_right=matriz_indices[i][col_right];
323             indice_down=matriz_indices[fila_down][j];
324
325             /* Guardamos los  ndices  de los c mulos
                   contiguos al elemento i, j.
326              * Si alguna es 0, se guarda un 0, que al
                    no corresponder a ning n  c mulo,
                    significa
327              * que en realidad no tiene vecino en esa
                    direcci n.
328              *
329              * Incrementamos el n mero de vecinos de
                    la(s) grieta(s) dada(s) por el(los)
                     ndice (s)
330              * del (de los) elemento(s) contiguo(s).
331              * Si alguno es 0, se incrementa el numero
                    de vecinos del "c mulo 0", que es el
                    c mulo
332              * de los elementos no rotos y no se
                    utiliza para calcular las cargas.
333              */
334
335             vecinos[i][j].cum1=indice_up;
336             numero_vecinos[indice_up]++;
337             if(indice_left!=indice_up)
338             {
339                 vecinos[i][j].cum2=indice_left;
340                 numero_vecinos[indice_left]++;
341                 if(indice_right!=indice_up &&
                       indice_right!=indice_left)
```

```
342                         {
343                             vecinos[i][j].cum3=indice_right;
344                             numero_vecinos[indice_right]++;
345                             if(indice_down!=indice_up &&
                                   indice_down!=indice_left &&
                                   indice_down!=indice_right) //Los
                                    4 son distintos
346                             {
347                                 vecinos[i][j].cum4=indice_down;
348                                 numero_vecinos[indice_down]++;
349                             }else             //3 son distintos
350                             {
351                                 vecinos[i][j].cum4=0;       //
                                       Si es doblemente vecino de
                                       alguno, solo lo guardamos 1
                                       vez
352                             }
353                         }else
354                         {
355                             if(indice_down!=indice_up &&
                                   indice_down!= indice_left) //3
                                   son distintos
356                             {
357                                 vecinos[i][j].cum3=indice_down;
358                                 numero_vecinos[indice_down]++;
359                             }else  //2 son distintos
360                             {
361                                 vecinos[i][j].cum3=0;
362                             }
363                             vecinos[i][j].cum4=0;
364                         }
365                     }else
366                     {
367                         if(indice_right!=indice_up){
368                             vecinos[i][j].cum2=indice_right;
369                             numero_vecinos[indice_right]++;
370                             if(indice_down!=indice_up &&
                                   indice_down!=indice_right){
371                                 vecinos[i][j].cum3=indice_down;
372                                 numero_vecinos[indice_down]++;
```

```
373                             }else
374                             {
375                                 vecinos[i][j].cum3=0;
376                             }
377                         }else
378                         {
379                             if(indice_down!=indice_up)
380                             {
381                                 vecinos[i][j].cum2=indice_down;
382                                 numero_vecinos[indice_down]++;
383                             }else
384                             {
385                                 vecinos[i][j].cum2=0;
386                             }
387                             vecinos[i][j].cum3=0;
388                         }
389                         vecinos[i][j].cum4=0;
390                     }
391                 }
392             }
393         }
394     }
395     void calcula_cargas_red(double cargas_red[N][M], int
        matriz_etiquetas[N][M], Almacenaje_vecinos vecinos[N][M
        ],
396                             int rotos_grieta[size_vect], int
                                numero_vecinos[size_vect])
397     {
398         int etiqueta;
399         for(int i=0; i<N; i++){
400             for(int j=0; j<M; j++){
401                 etiqueta=matriz_etiquetas[i][j];
402                 if(etiqueta!=0){      // Elemento ya roto, no
                    soporta carga
403                     cargas_red[i][j]=0;
404                 }else{                // Elemento sin romper
405                     /* Incrementamos las cargas, como la suma (
                        restringida a los vecinos no nulos),
                        mediante: r_k/s_k
406                      * siendo r el n mero de elementos rotos
```

```
                    de la grieta y s el n mero de vecinos
                    de la grieta.
407                  */
408                 cargas_red[i][j]=1;
409                 if((vecinos[i][j].cum1)!=0){
410                     cargas_red[i][j]+=((double)rotos_grieta
                        [vecinos[i][j].cum1]/(numero_vecinos
                        [vecinos[i][j].cum1]));
411                 }
412                 if((vecinos[i][j].cum2)!=0){
413                     cargas_red[i][j]+=((double)rotos_grieta
                        [vecinos[i][j].cum2]/(numero_vecinos
                        [vecinos[i][j].cum2]));
414                 }
415                 if((vecinos[i][j].cum3)!=0){
416                     cargas_red[i][j]+=((double)rotos_grieta
                        [vecinos[i][j].cum3]/(numero_vecinos
                        [vecinos[i][j].cum3]));
417                 }
418                 if((vecinos[i][j].cum4)!=0){
419                     cargas_red[i][j]+=((double)rotos_grieta
                        [vecinos[i][j].cum4]/(numero_vecinos
                        [vecinos[i][j].cum4]));
420                 }
421             }
422         }
423     }
424 }
425 /*
    ************************************************************
     */
426
427 void inicializa_cargas(double cargas[N][M])
428 {
429     int i,j;
430     for (i=0;i<N;i++)
431     {
432         for(j=0;j<M;j++)
433         {
434             cargas[i][j]=1;
```

```
435             }
436         }
437     }
438     void inicializa_grietas(int grietas[N][M])
439     {
440         int i,j;
441         for (i=0;i<N;i++)
442         {
443             for(j=0;j<M;j++)
444             {
445                 grietas[i][j]=0;
446             }
447         }
448     }
449     double probabilidades_ruptura(double cargas[N][M], double
        probabilidades[N][M])
450     {
451         double gamma_total=0;       // Tiempo de vida de la
            configuracion actual
452         double gamma[N][M];         // Matriz para el calculo de
            las probabilidades
453
454         //Inicializo gamma
455         for(int i=0; i<N; i++){
456             for(int j=0; j<M; j++){
457                 gamma[i][j]=0;
458             }
459         }
460
461
462         for(int i=0; i<N; i++)//Calculo las gammas
463         {
464             for(int j=0; j<M; j++)
465             {
466                 if(cargas[i][j]!=0)
467                 {
468                     gamma[i][j]=exp(cargas[i][j]);
469                     gamma_total+=gamma[i][j];
470                 }
471             }
```

```
472      }
473
474      for(int i=0; i<N; i++){   // Calculo de las
             probabilidades
475          for(int j=0; j<M; j++){
476              probabilidades[i][j]=(gamma[i][j])/(gamma_total
                 );
477          }
478      }
479
480      return 1/gamma_total;     // Se devuelve el valor de
             delta
481  }
482  void calcula_elemento_a_romper(double probabilidades[N][M],
         int *fila, int *columna)
483  {
484      double numero=random_01();
485      double acum_prob=0;
486      int j, i;
487
488      for(i=0; i<N; i++){
489          for(j=0; j<M; j++){
490              acum_prob+=probabilidades[i][j];
491              if(acum_prob>numero){   // El indice se
                     devuelve cuando el numero esta entre dos
                     separaciones de segmentos
492                  *fila=i;
493                  *columna=j;
494                  return;   // Para salir del bucle
495              }
496          }
497      }
498  }
499  void Histograma (double *data,double *Hist, int N_data,int
         N_intervalos, double *d, double *m, double *Max)
500  {
501      /*
502      *data -> input, Datos sobre los que se genera el
             histograma
503      *Hist -> output Histograma calculado
```

```
504       N_data -> input Numero de datos
505       N_intervalos -> input, Numero de intervalos del
              histograma
506       *d -> output Medida de cada intervalo del histograma
507       *m -> output Valor minimo de los datos
508       *M -> output Valor maximo de los datos
509       */
510       int i,Indice;// Importante que el indice sea int ya que
              va a redondear al entero
511       double Norm,delta,minimo,maximo;
512       for (i=0;i<N_intervalos;i++)
513       {
514           Hist[i]=0;
515       }
516       minimo=10000000;
517       maximo=-10000000;
518       for (i=0;i<N_data;i++) //Calculo el minimo y maximo
              valor
519       {
520           if(data[i]>maximo)maximo=data[i];
521           if(data[i]<minimo)minimo=data[i];
522       }
523       delta=(maximo-minimo)/N_intervalos;
524       if(delta==0)
525       {
526           printf("No se pueden calcular los intervalos");
527           exit(1);
528       }
529       //Nucleo del programa
530       for(i=0;i<N_data;i++)
531       {
532       Indice=(data[i]-minimo)/delta;
533       Hist[Indice]=Hist[Indice]+1;
534       }
535       *d=delta;
536       *m=minimo;
537       *Max=maximo;
538       //Ahora normalizo
539
540       Norm=1.0/(N_data*delta);
```

```
541     for(i=0;i<N_intervalos;i++)
542     {
543         Hist[i]=Hist[i]*Norm;
544     }
545 }
546
547
548
549
550 /*************************** PARISI RAPUANO
       ***************/
551 #define NormParisi (2.3283063671E-10F)  //Para normalizar
       el valor generado en la rueda de Parisi-Rapuano
552
553 double random_01()
554 {
555
556     int i;
557     unsigned int rueda[256], aleatorio;
558     unsigned char indice_ran, indice1, indice2, indice3;
559     //Inicializar rueda
560     for(i = 0; i < 256; i++)
561         rueda[i] = (rand()<<16) + rand();
562     //Inicializar indices
563     indice_ran = 0; indice1 = 0; indice2 = 0; indice3 = 0;
564     //Modificamos los indices
565     indice1 = indice_ran - 24;
566     indice2 = indice_ran - 55;
567     indice3 = indice_ran - 61;
568     //Modificamos la rueda
569     rueda[indice_ran] = rueda[indice1] + rueda[indice2];
570     //Generamos un numero aleatorio entre 0 y 2^32-1
571     aleatorio = (rueda[indice_ran]^rueda[indice3]);
572     //Cambiamos la posicion base para el siguiente numero
           aleatorio
573     indice_ran++;
574     //Devolvemos el numero aleatorio normalizado, entre 0 y
           1
575     return aleatorio * NormParisi;
576 }
```

```
577 /********************************* FIN PARISI RAPUANO
       ***********/
```

Código utilizado para las simulaciones 2D (entorno de Moore):

```
1
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <time.h>
5  #include <math.h>
6
7
8  int size_vect,N,M,NSim;
9
10
11 /* ********************STRUCT***************** */
12 struct almacenaje_vecinos{
13     int cum1, cum2, cum3, cum4;
14 }; typedef struct almacenaje_vecinos Almacenaje_vecinos;
15 /* ***************************************** */
16
17 /* ********************* FUNCIONES ***************** */
18 double random_01();
19 double probabilidades_ruptura(double cargas[N][M], double
       probabilidades[N][M]);
20 void calcula_elemento_a_romper(double probabilidades[N][M],
       int *fila, int *columna);
21 /********Reparto de cargas***********/
22 void reindexa_grietas(int matriz_indices[N][M], int *
       rotos_grieta, int colum, int fila, int *indice_nuevo);
       //Matriz_indices=grietas
23 void iguala_indices_2grietas(int matriz[N][M], int *
       rotos_grieta, int indice_asignado, int cambia); //Matriz
       =grietas
24 void iguala_indices_3grietas(int matriz[N][M], int *
       rotos_grieta, int indice_asignado, int cambia1, int
       cambia2);
25 void iguala_indices_4grietas(int matriz[N][M], int *
       rotos_grieta, int indice_asignado, int cambia1, int
       cambia2, int cambia3);
```

```
26 void encuentra_vecinos(int matriz_indices[N][M], int
      numero_vecinos[size_vect], Almacenaje_vecinos vecinos[N
      ][M]);
27 void calcula_cargas_red(double cargas_red[N][M], int
      matriz_etiquetas[N][M], Almacenaje_vecinos vecinos[N][M
      ],
28 int rotos_grieta[size_vect], int numero_vecinos[size_vect])
      ;
29 /*********************************/
30 void inicializa_vector(int *vector, int valor, int tamano);
31 void inicializa_cargas(double cargas[N][M]);
32 void inicializa_grietas(int grietas[N][M]);
33 void Histograma (double *,double *, int,int , double *,
      double *, double *);
34 /* ************************************************ */
35
36
37
38 int main ()
39 {
40     //Voy a poner aqui las modificaciones para haver TvsN
41     /*********************************/
42     int TamanoFinal,TamanoInicial;
43
44     TamanoInicial=200;
45     TamanoFinal=200;
46
47     /*********************************/
48
49     srand(time(NULL));
50
51     int i,j,Npasos;
52
53     FILE*f;
54     char name3[128];
55     sprintf(name3, "TvsNMoore(nev1).txt");
56     f=fopen(name3,"w");
57
58     for(TamanoInicial;TamanoInicial<=TamanoFinal;
          TamanoInicial+=10)
```

```
59      {
60
61          NSim=2000;
62          N=TamanoInicial;
63          M=TamanoInicial;
64          size_vect=(int)N*M/2+1;
65          Npasos=N*M;
66          printf("\nN=M= %d\n",N);
67
68          if(TamanoInicial>=50)
69          {
70                  if(TamanoInicial>=100)
71                  {
72                      NSim=(int)NSim/100;
73                  }else
74                  {
75                      NSim=(int)NSim/10;
76                  }
77          }
78
79          double cargas[N][M]; //Matriz de cargas
80          int grietas[N][M]; //Matriz que almazena las
               grietas
81          int rotos_grieta[size_vect];//Almacena el tama o
               de cada grieta
82          int perimetros[size_vect];//Almacena el numero de
               vecinos de cada grieta
83          double probabilidades[N][M]; //Matriz para las
               probabilidades de ruptura
84          int fila,columna; //Indices del elemento que se va
               a romper
85          int indice_nuevo; //Indice de la proxima grieta a
               indexarse
86          double delta,T; //Tiempo para cada paso y tiempo
               total
87          double v[NSim]; //Vector donde guardamos el valor
               de T para cada simulacion
88
89
90          Almacenaje_vecinos vecinos [N][M]; //Hoshen-
```

```
            Kopelman: para cada elemento asocio un vector
            con sus cuatro vecions
91
92          for(j=0;j<NSim;j++)
93          {
94              indice_nuevo=1;
95              inicializa_cargas(cargas);
96              inicializa_grietas(grietas);
97              for(i=0;i<size_vect;i++)
98                  rotos_grieta[i]=0;
99
100             //f=fopen("grietas.txt","w");
101             T=0;
102             delta=0;
103             delta=probabilidades_ruptura(cargas,
                   probabilidades);
104             T+=delta;
105             for(i=0;i<Npasos-1;i++)//Debemos contar hasta N
                   -1 porque sino estariamos asociando delta al
                    vector cuando se ha roto por completo
106             {
107
108
109                 calcula_elemento_a_romper(probabilidades,&
                       fila,&columna);
110
111                 //Rompo el elemento indicado
112                 cargas[fila][columna]=0;
113                 //Inicia el reparto de cargas
114                 reindexa_grietas(grietas,rotos_grieta,
                       columna,fila,&indice_nuevo);
115                 encuentra_vecinos(grietas,perimetros,
                       vecinos);
116                 calcula_cargas_red(cargas,grietas,vecinos,
                       rotos_grieta,perimetros);
117
118                 delta=0;
119                 delta=probabilidades_ruptura(cargas,
                       probabilidades);
120                 T+=delta;
```

```
121                 }
122                 //fclose(f);
123 
124                 /* Progreso por pantalla */
125                 if(j*100%NSim==0)
126                     printf("Progreso = %d%%\n",j*100/NSim);
127                 /* ********************  */
128 
129                 v[j]=T;
130 
131             }
132 
133             //Calculo el valor medio de T as  como su error.
134 
135             double media,var,err;
136             var=0;
137             media=0;
138 
139 
140             for(i=0;i<NSim;i++)
141             {
142                 media+=v[i];
143                 var+=v[i]*v[i];
144             }
145 
146             media=media/(double)NSim;
147             var=var/(double)NSim;
148             var=var-media*media;
149             var=sqrt(var);
150             err=var/sqrt((double)NSim);
151 
152             printf("\nN=M=%d\t<T>=%f\terr=%f\tNSim=%d\n",N,
                    media,err,NSim);
153             fprintf(f," %d\t %d\t %f\t %f\n",N,(int)N*N,media,err);
154 
155         }
156         fclose(f);
157 
158     }
159     /* ****************REPARTO DE CARGAS*************** */
```

```
160 void inicializa_vector(int *vector, int valor, int tamano)
161 {
162     for(int i=0; i<tamano; i++){
163         vector[i]=valor;
164     }
165 }
166 void iguala_indices_2grietas(int matriz[N][M], int *
       rotos_grieta, int indice_asignado, int cambia)
167 {
168     if(cambia!=indice_asignado){    //Si es igual, no hay
          que hacer nada  No  se tendria que cumplir siempre
          que es distinto?
169         int ncambiados=0;
170         for(int i=0; i<N; i++){
171             for(int j=0; j<M; j++){
172                 if(matriz[i][j]==cambia){   // Cambiamos
                       los del  ndice  a cambiar por el  ndice
                        a asignar
173                     matriz[i][j]=indice_asignado;
174                     ncambiados++;
175                 }
176             }
177         }
178         rotos_grieta[indice_asignado]+=ncambiados;
179         rotos_grieta[cambia]=0;
180     }
181 }
182
183 void iguala_indices_3grietas(int matriz[N][M], int *
       rotos_grieta, int indice_asignado, int cambia1, int
       cambia2)
184 {
185     if(indice_asignado==cambia1){            // Si el
          primero ya es igual, no hace falta reindexarlo
186         iguala_indices_2grietas(matriz, rotos_grieta,
              indice_asignado, cambia2);
187     }else if(indice_asignado==cambia2){      // Si el
          segundo ya es igual, no hace falta reindexarlo
188         iguala_indices_2grietas(matriz, rotos_grieta,
              indice_asignado, cambia1);
```

```
189      }else{
190          int ncambiados=0;
191          for(int i=0; i<N; i++){
192              for(int j=0; j<M; j++){
193                  if(matriz[i][j]==cambia1 || matriz[i][j]==
                        cambia2){
194                      matriz[i][j]=indice_asignado;
195                      ncambiados++;
196                  }
197              }
198          }
199          rotos_grieta[indice_asignado]+=ncambiados;
200          rotos_grieta[cambia1]=0;
201          rotos_grieta[cambia2]=0;
202      }
203  }
204  void iguala_indices_4grietas(int matriz[N][M], int *
        rotos_grieta, int indice_asignado, int cambia1, int
        cambia2, int cambia3)
205  {
206      if(indice_asignado==cambia1){           // Si el
            tercero ya es el asignado no hace falta cambiarlo
207          iguala_indices_3grietas(matriz, rotos_grieta,
                indice_asignado, cambia2, cambia3);
208      }else if(indice_asignado==cambia2){     // Si el
            segundo ya es el asignado no hace falta cambiarlo
209          iguala_indices_3grietas(matriz, rotos_grieta,
                indice_asignado, cambia1, cambia3);
210      }else if(indice_asignado==cambia3){     // Si el
            tercero ya es el asignado no hace falta cambiarlo
211              iguala_indices_3grietas(matriz, rotos_grieta,
                    indice_asignado, cambia1, cambia2);
212      }else{
213          int ncambiados=0;
214          for(int i=0; i<N; i++){
215              for(int j=0; j<M; j++){
216                  if(matriz[i][j]==cambia1 || matriz[i][j]==
                        cambia2 || matriz[i][j]==cambia3){
217                      matriz[i][j]=indice_asignado;
218                      ncambiados++;
```

```
219                     }
220                 }
221             }
222             rotos_grieta[indice_asignado]+=ncambiados;
223             rotos_grieta[cambia1]=0;
224             rotos_grieta[cambia2]=0;
225             rotos_grieta[cambia3]=0;
226         }
227     }
228     //Nota: Matriz indices=grietas
229     void reindexa_grietas(int matriz_indices[N][M], int *
        rotos_grieta, int colum, int fila, int *indice_nuevo)
230     {
231         /*---Calculamos las coordenadas de los elementos
            adyacentes, respetando las condiciones peri dicas
            ---*/
232         int fila_up, fila_down, colum_left, colum_right;
233 
234         colum_left=(colum+2*M-1)%M;     // Si colum es 0,
            colum_left=M-1; en caso contrario, colum_left=colum
            -1
235         fila_up=(fila+2*N-1)%N;         // Si fila es 0,
            fila_up=M-1; en caso contrario, fila_up=fila-1
236         colum_right=(colum+M+1)%M;      // Si colum es M-1,
            colum_right=0; en caso contrario, colum_right=colum
            +1
237         fila_down=(fila+N+1)%N;         // Si fila es N-1,
            fila_down=0; en caso contrario, fila_down=fila+1
238 
239         /*---Calculamos los  ndices  de los elementos
            adyacentes---*/
240         int indice[8];    //vector en el que voy a guardar los
            indices de los vecinonos (N,S,E,O,NE,NO,SE,SO)
241         int indice_asignado; //Indice que va a perdurar
242         int a,b,c;           //indices auxiliares
243         int i; //contador
244 
245 
246         indice[0]=matriz_indices[fila_up][colum];
247         indice[1]=matriz_indices[fila_down][colum];
```

```
248       indice[2]=matriz_indices[fila][colum_left];
249       indice[3]=matriz_indices[fila][colum_right];
250       indice[4]=matriz_indices[fila_up][colum_right];
251       indice[5]=matriz_indices[fila_down][colum_right];
252       indice[6]=matriz_indices[fila_up][colum_left];
253       indice[7]=matriz_indices[fila_down][colum_left];
254
255       i=0;
256       /*---Distinguimos casos seg n el n mero de vecinos
             rotos---*/
257       switch (!!indice[0] + !!indice[1] + !!indice[2] + !!
             indice[3] + !!indice[4] + !!indice[5] + !!indice[6]
             + !!indice[7]) {     // '!!a' devuelve 1 si a!=0, 0
             si a=0. Esto da el # de vecinos que no son 0
258         case 0:   // Nueva grieta
259           indice_asignado=*indice_nuevo;
260           (*indice_nuevo)++; // La siguiente grieta nueva
                 estar  asociada al siguiente  ndice  nuevo
261         break;
262         case 1:   // El elemento roto es vecino de una s la
               grieta. Se le asocia el  ndice  no nulo.
263
264           while(indice[i]==0)i++;
265           indice_asignado=indice[i];
266
267         break;
268         case 2:   // El elemento roto es vecino de 2 grietas.
               Se le asocia el  ndice !=0, respetando prioridad
              (N,S,E,O,NE,NO,SE,SO)
269
270           while(indice[i]==0)i++;
271           indice_asignado=indice[i];
272           i++;
273           while(indice[i]==0)i++;
274           a=indice[i];
275
276           iguala_indices_2grietas(matriz_indices,
                 rotos_grieta, indice_asignado, a); // Se igualan
                  los  ndices  de ambas grietas
277         break;
```

```
278         case 3:   // El elemento roto es vecino de 3 grietas.
                Se le asocia el  ndice !=0, respetando prioridad
               (up, left, right, down)
279
280           while(indice[i]==0)i++;
281           indice_asignado=indice[i];
282           i++;
283           while(indice[i]==0)i++;
284           a=indice[i];
285           i++;
286           while(indice[i]==0)i++;
287           b=indice[i];
288
289           iguala_indices_3grietas(matriz_indices,
                 rotos_grieta, indice_asignado, a, b);
290
291         break;
292         case 4:   // El elemento roto es vecino de 4 c mulos
              .Se le asocia el  ndice !=0, respetando prioridad
              (up, left, right, down)
293
294           while(indice[i]==0)i++;
295           indice_asignado=indice[i];
296           i++;
297           while(indice[i]==0)i++;
298           a=indice[i];
299           i++;
300           while(indice[i]==0)i++;
301           b=indice[i];
302           i++;
303           while(indice[i]==0)i++;
304           c=indice[i];
305
306           iguala_indices_4grietas(matriz_indices,
                 rotos_grieta, indice_asignado, a, b, c);   // Se
                  igualan los  ndices  de las 4 grietas
307
308       break;
309       case 5:     //Solo pueden existir 3 grietas distintas
310
```

```
311
312             while(indice[i]==0)i++;
313             indice_asignado=indice[i];
314
315             //Veo si todos son la misma grieta, dos o 3
                   distintas
316             a=0;
317             b=0;
318             for(i=0;i<8;i++)
319             {
320                 if(indice[i]!=indice_asignado){a=indice[i];}
321             }
322             for(i=0;i<8;i++)
323             {
324                 if(indice[i]!=indice_asignado && indice[i]!=a){
                       b=indice[i];}
325             }
326
327             switch(!!a+!!b){             // !!a devuelve 1 si a
                   !=0
328                 case 0:
329                     //No hacemos nada, el elemento nuevo es
                           vecino de una sola grieta, le asociamos
                           el indice asignado(Hecho arriba)
330                 break;
331                 case 1:
332                     if(a!=0)
333                     {
334                         iguala_indices_2grietas(matriz_indices,
                               rotos_grieta, indice_asignado, a);
335                     }else
336                     {
337                         iguala_indices_2grietas(matriz_indices,
                               rotos_grieta, indice_asignado, b);
338                     }
339                 break;
340                 case 2:
341                     iguala_indices_3grietas(matriz_indices,
                           rotos_grieta, indice_asignado, a, b);
342             }
```

```
343
344         break;
345         case 6: //Solo pueden existir 2 grietas distintas
346
347             while(indice[i]==0)i++;
348             indice_asignado=indice[i];
349
350             a=0;
351             for(i=0;i<8;i++)
352             {
353                 if(indice[i]!=indice_asignado){a=indice[i];}
354             }
355
356             /* Si a=0, todo es la misma grieta, no hago nada (
                   ya hemos dado valor a "indice_asignado" */
357             if(a!=0)
358             {
359                 iguala_indices_2grietas(matriz_indices,
                       rotos_grieta, indice_asignado, a);
360             }
361
362         break;
363         case 7: //Todo es la misma grieta
364
365             while(indice[i]==0)i++;
366             indice_asignado=indice[i];
367
368         break;
369         case 8: //la grieta rodea nuestro punto, todos los
               indices seran !=0 y tendran el mismo valor
370
371             indice_asignado=indice[0];
372
373         }
374         matriz_indices[fila][colum]=indice_asignado;    // Se
               asigna el  ndice  adecuado al nuevo elemento roto
375         rotos_grieta[indice_asignado]++;                //
               Nuevo elemento para la grieta con la que e s t  en
               contacto
376     }
```

```
377 void encuentra_vecinos(int matriz_indices[N][M], int
       numero_vecinos[size_vect], Almacenaje_vecinos vecinos[N
       ][M])
378 {
379     int col_left, col_right, fila_up, fila_down;
380     int indice_up, indice_left, indice_right, indice_down;
381     inicializa_vector(numero_vecinos, 0, size_vect);
          // El n mero de vecinos se calcula cada vez que se
          ejecuta esta funci n
382     int indice[8]; //Vector en el que guardo el valor de
          los indices de las grietas (N,S,E,O,NE,NO,SE,SO)
383     int k;//Contador
384
385     /*---Leemos la matriz:---*/
386     for(int i=0; i<N; i++){
387         fila_up=(i-1+2*N)%N;
388         fila_down=(i+1+N)%N;
389         for(int j=0; j<M; j++){
390             col_left=(j-1+2*M)%M;
391             col_right=(j+1+M)%M;
392             if(matriz_indices[i][j]!=0){    // Los
                  elementos ya rotos, como no tienen carga, no
                   los utilizaremos
393                 vecinos[i][j].cum1=0;
394                 vecinos[i][j].cum2=0;
395                 vecinos[i][j].cum3=0;
396                 vecinos[i][j].cum4=0;
397             }else{                          // Si no es 0,
                  no es un vecino, sino un elemento roto
398                 indice[0]=matriz_indices[fila_up][j]; //N
399                 indice[1]=matriz_indices[fila_down][j]; //S
400                 indice[2]=matriz_indices[i][col_left]; //O
401                 indice[3]=matriz_indices[i][col_right]; //E
402                 indice[4]=matriz_indices[fila_up][col_right
                      ]; //NE
403                 indice[5]=matriz_indices[fila_down][
                      col_right]; //SE
404                 indice[6]=matriz_indices[fila_up][col_left
                      ]; //NO
405                 indice[7]=matriz_indices[fila_down][
```

```
                        col_left]; //SO
406
407                     k=0;
408                     //Como maximo puede haber 4 grietas,
                           utilizaremos la misma notaci n que el
                           codigo anterior descartando los indices
                           repetidos.
409                     switch (!!indice[0] + !!indice[1] + !!
                           indice[2] + !!indice[3] + !!indice[4] +
                           !!indice[5] + !!indice[6] + !!indice[7])
                            {
410                     case 0: //Todos son 0
411                         indice_up=0;
412                         indice_left=0;
413                         indice_right=0;
414                         indice_down=0;
415                     break;
416                     case 1:
417                         while(indice[k]==0)k++;
418                         indice_up=indice[k];
419                         indice_left=0;
420                         indice_right=0;
421                         indice_down=0;
422                     break;
423                     case 2:  //No hace falta mirar si son
                           iguales ya que lo tiene en cuenta el
                           algoritmo
424                         while(indice[k]==0)k++;
425                         indice_up=indice[k];
426                         k++;
427                         while(indice[k]==0)k++;
428                         indice_down=indice[k];
429                         indice_right=0;
430                         indice_left=0;
431                     break;
432                     case 3:
433                         while(indice[k]==0)k++;
434                         indice_up=indice[k];
435                         k++;
436                         while(indice[k]==0)k++;
```

```
437                         indice_down=indice[k];
438                         k++;
439                         while(indice[k]==0)k++;
440                         indice_right=indice[k];
441                         indice_left=0;
442                     break;
443                     case 4:
444                         while(indice[k]==0)k++;
445                         indice_up=indice[k];
446                         k++;
447                         while(indice[k]==0)k++;
448                         indice_down=indice[k];
449                         k++;
450                         while(indice[k]==0)k++;
451                         indice_right=indice[k];
452                         k++;
453                         while(indice[k]==0)k++;
454                         indice_left=indice[k];
455                     break;
456                     case 5: //Puede haber como m x i m o  3
                           grietas, descarto los indices que son
                           iguales, para no coger los 3 primeros
                           distintos de cero y que sean los mismos
457                         while(indice[k]==0)k++;
458                         indice_up=indice[k];
459
460                         //Veo si todos son la misma grieta, 2 o
                                3 distintas
461                         indice_down=0;
462                         indice_right=0;
463                         indice_left=0;
464                         for(k=0;k<8;k++)
465                         {
466                             if(indice[k]!=indice_up){
                                   indice_down=indice[k];}
467                         }
468                         for(k=0;k<8;k++)
469                         {
470                             if(indice[k]!=indice_up && indice[k
                                   ]!=indice_down){indice_right=
```

```
                                indice[k];}
471                         }
472                     break;
473                     case 6: //Como m x i m o hay 2 grietas
474                         while(indice[k]==0)k++;
475                         indice_up=indice[k];
476
477                         indice_down=0;
478                         indice_right=0;
479                         indice_left=0;
480                         for(k=0;k<8;k++)
481                         {
482                             if(indice[k]!=indice_up){
                                   indice_down=indice[k];}
483                         }
484                     break;
485                     case 7: //Hay una sola grieta con todos los
                        indices iguales y uno distinto de 0
486                         while(indice[k]==0)k++;
487                         indice_up=indice[k];
488                         indice_down=indice[k];
489                         indice_left=indice[k];
490                         indice_right=0;
491                     break;
492                     case 8:
493                         indice_up=indice[0];
494                         indice_down=indice[0];
495                         indice_left=indice[0];
496                         indice_right=indice[0];
497                     }
498
499                     /* Guardamos los  ndices  de los c mulos
                           contiguos al elemento i, j.
500                      * Si alguna es 0, se guarda un 0, que al
                           no corresponder a n i n g n  c mulo,
                           significa
501                      * que en realidad no tiene vecino en esa
                           direcci n.
502                      *
503                      * Incrementamos el  n m e r o de vecinos de
```

```
                        la(s) grieta(s) dada(s) por el(los)
                         ndice (s)
504                     * del (de los) elemento(s) contiguo(s).
505                     * Si alguno es 0, se incrementa el numero
                           de vecinos del "c mulo 0", que es el
                           c mulo
506                     * de los elementos no rotos y no se
                           utiliza para calcular las cargas.
507                     */
508
509                    vecinos[i][j].cum1=indice_up;
510                    numero_vecinos[indice_up]++;
511                    if(indice_left!=indice_up)
512                    {
513                        vecinos[i][j].cum2=indice_left;
514                        numero_vecinos[indice_left]++;
515                        if(indice_right!=indice_up &&
                              indice_right!=indice_left)
516                        {
517                            vecinos[i][j].cum3=indice_right;
518                            numero_vecinos[indice_right]++;
519                            if(indice_down!=indice_up &&
                                 indice_down!=indice_left &&
                                 indice_down!=indice_right) //Los
                                  4 son distintos
520                            {
521                                vecinos[i][j].cum4=indice_down;
522                                numero_vecinos[indice_down]++;
523                            }else             //3 son distintos
524                            {
525                                vecinos[i][j].cum4=0;       //
                                     Si es doblemente vecino de
                                     alguno, solo lo guardamos 1
                                     vez
526                            }
527                        }else
528                        {
529                            if(indice_down!=indice_up &&
                                 indice_down!= indice_left) //3
                                 son distintos
```

```
                            {
                                vecinos[i][j].cum3=indice_down;
                                numero_vecinos[indice_down]++;
                            }else  //2 son distintos
                            {
                                vecinos[i][j].cum3=0;
                            }
                            vecinos[i][j].cum4=0;
                        }
                    }else
                    {
                        if(indice_right!=indice_up){
                            vecinos[i][j].cum2=indice_right;
                            numero_vecinos[indice_right]++;
                            if(indice_down!=indice_up &&
                               indice_down!=indice_right){
                                vecinos[i][j].cum3=indice_down;
                                numero_vecinos[indice_down]++;
                            }else
                            {
                                vecinos[i][j].cum3=0;
                            }
                        }else
                        {
                            if(indice_down!=indice_up)
                            {
                                vecinos[i][j].cum2=indice_down;
                                numero_vecinos[indice_down]++;
                            }else
                            {
                                vecinos[i][j].cum2=0;
                            }
                            vecinos[i][j].cum3=0;
                        }
                        vecinos[i][j].cum4=0;
                    }
                }
            }
        }
    }
}
```

```
569 void calcula_cargas_red(double cargas_red[N][M], int
       matriz_etiquetas[N][M], Almacenaje_vecinos vecinos[N][M
       ],
570                          int rotos_grieta[size_vect], int
                                numero_vecinos[size_vect])
571 {
572     int etiqueta;
573     for(int i=0; i<N; i++){
574         for(int j=0; j<M; j++){
575             etiqueta=matriz_etiquetas[i][j];
576             if(etiqueta!=0){      // Elemento ya roto, no
                   soporta carga
577                 cargas_red[i][j]=0;
578             }else{                // Elemento sin romper
579                 /* Incrementamos las cargas, como la suma (
                       restringida a los vecinos no nulos),
                       mediante: r_k/s_k
580                  * siendo r el n mero de elementos rotos
                       de la grieta y s el n mero de vecinos
                       de la grieta.
581                  */
582                 cargas_red[i][j]=1;
583                 if((vecinos[i][j].cum1)!=0){
584                     cargas_red[i][j]+=((double)rotos_grieta
                           [vecinos[i][j].cum1]/(numero_vecinos
                           [vecinos[i][j].cum1]));
585                 }
586                 if((vecinos[i][j].cum2)!=0){
587                     cargas_red[i][j]+=((double)rotos_grieta
                           [vecinos[i][j].cum2]/(numero_vecinos
                           [vecinos[i][j].cum2]));
588                 }
589                 if((vecinos[i][j].cum3)!=0){
590                     cargas_red[i][j]+=((double)rotos_grieta
                           [vecinos[i][j].cum3]/(numero_vecinos
                           [vecinos[i][j].cum3]));
591                 }
592                 if((vecinos[i][j].cum4)!=0){
593                     cargas_red[i][j]+=((double)rotos_grieta
                           [vecinos[i][j].cum4]/(numero_vecinos
```

```
                        [vecinos[i][j].cum4]));
                }
            }
        }
    }
}
/*
   ************************************************************
    */

void inicializa_cargas(double cargas[N][M])
{
    int i,j;
    for (i=0;i<N;i++)
    {
        for(j=0;j<M;j++)
        {
            cargas[i][j]=1;
        }
    }
}
void inicializa_grietas(int grietas[N][M])
{
    int i,j;
    for (i=0;i<N;i++)
    {
        for(j=0;j<M;j++)
        {
            grietas[i][j]=0;
        }
    }
}
double probabilidades_ruptura(double cargas[N][M], double
   probabilidades[N][M])
{
    double gamma_total=0;      // Tiempo de vida de la
       configuracion actual
    double gamma[N][M];        // Matriz para el calculo de
        las probabilidades

```

```
628     //Inicializo gamma
629     for(int i=0; i<N; i++){
630         for(int j=0; j<M; j++){
631             gamma[i][j]=0;
632         }
633     }
634 
635 
636     for(int i=0; i<N; i++)//Calculo las gammas
637     {
638         for(int j=0; j<M; j++)
639         {
640             if(cargas[i][j]!=0)
641             {
642                 gamma[i][j]=exp(cargas[i][j]);
643                 gamma_total+=gamma[i][j];
644             }
645         }
646     }
647 
648     for(int i=0; i<N; i++){   // Calculo de las
          probabilidades
649         for(int j=0; j<M; j++){
650             probabilidades[i][j]=(gamma[i][j])/(gamma_total
                  );
651         }
652     }
653 
654     return 1/gamma_total;     // Se devuelve el valor de
          delta
655 }
656 void calcula_elemento_a_romper(double probabilidades[N][M],
      int *fila, int *columna)
657 {
658     double numero=random_01();
659     double acum_prob=0;
660     int j, i;
661 
662     for(i=0; i<N; i++){
663         for(j=0; j<M; j++){
```

```
664                 acum_prob+=probabilidades[i][j];
665                 if(acum_prob>numero){   // El indice se
                       devuelve cuando el numero esta entre dos
                       separaciones de segmentos
666                     *fila=i;
667                     *columna=j;
668                     return;   // Para salir del bucle
669                 }
670             }
671         }
672     }
673     void Histograma (double *data,double *Hist, int N_data,int
           N_intervalos, double *d, double *m, double *Max)
674     {
675         /*
676         *data -> input, Datos sobre los que se genera el
               histograma
677         *Hist -> output Histograma calculado
678         N_data -> input Numero de datos
679         N_intervalos -> input, Numero de intervalos del
               histograma
680         *d -> output Medida de cada intervalo del histograma
681         *m -> output Valor minimo de los datos
682         *M -> output Valor maximo de los datos
683         */
684         int i,Indice;// Importante que el indice sea int ya que
               va a redondear al entero
685         double Norm,delta,minimo,maximo;
686         for (i=0;i<N_intervalos;i++)
687         {
688             Hist[i]=0;
689         }
690         minimo=10000000;
691         maximo=-10000000;
692         for (i=0;i<N_data;i++) //Calculo el minimo y maximo
               valor
693         {
694             if(data[i]>maximo)maximo=data[i];
695             if(data[i]<minimo)minimo=data[i];
696         }
```

```
697      delta=(maximo-minimo)/N_intervalos;
698      if(delta==0)
699      {
700          printf("No se pueden calcular los intervalos");
701          exit(1);
702      }
703      //Nucleo del programa
704      for(i=0;i<N_data;i++)
705      {
706      Indice=(data[i]-minimo)/delta;
707      Hist[Indice]=Hist[Indice]+1;
708      }
709      *d=delta;
710      *m=minimo;
711      *Max=maximo;
712      //Ahora normalizo
713
714      Norm=1.0/(N_data*delta);
715      for(i=0;i<N_intervalos;i++)
716      {
717          Hist[i]=Hist[i]*Norm;
718      }
719  }
720
721
722
723
724  /*************************** PARISI RAPUANO
        ***************/
725  #define NormParisi (2.3283063671E-10F)  //Para normalizar
        el valor generado en la rueda de Parisi-Rapuano
726
727  double random_01()
728  {
729
730      int i;
731      unsigned int rueda[256], aleatorio;
732      unsigned char indice_ran, indice1, indice2, indice3;
733      //Inicializar rueda
734      for(i = 0; i < 256; i++)
```

```
735             rueda[i] = (rand()<<16) + rand();
736         //Inicializar indices
737         indice_ran = 0; indice1 = 0; indice2 = 0; indice3 = 0;
738         //Modificamos los indices
739         indice1 = indice_ran - 24;
740         indice2 = indice_ran - 55;
741         indice3 = indice_ran - 61;
742         //Modificamos la rueda
743         rueda[indice_ran] = rueda[indice1] + rueda[indice2];
744         //Generamos un numero aleatorio entre 0 y 2^32-1
745         aleatorio = (rueda[indice_ran]^rueda[indice3]);
746         //Cambiamos la posicion base para el siguiente numero
               aleatorio
747         indice_ran++;
748         //Devolvemos el numero aleatorio normalizado, entre 0 y
               1
749         return aleatorio * NormParisi;
750     }
751     /******************************** FIN PARISI RAPUANO
           ***********/
```

Anexo II

Se adjuntan los datos que aparecen representados en la *Figura 5*.

Potencial (ρ =5)		
N	T_f	ΔT_f
10	0,174262	0,000499
50	0,073546	0,000643
100	0,055192	0,000592
500	0,031679	0,000415
1000	0,025958	0,000327
5000	0,017337	0,000199
10000	0,014097	0,000277
50000	0,009626	0,000412
100000	0,009368	0,000516

Potencial (ρ =2)		
N	T_f	ΔT_f
10	0,504001	0,000785
50	0,382273	0,001509
100	0,346003	0,001414
500	0,294727	0,001297
1000	0,274726	0,001225
5000	0,250227	0,000893
10000	0,239536	0,001241
50000	0,220487	0,002461
100000	0,207261	0,004790

Exponencial		
N	T_f	ΔT_f
10	0,202656	0,000652
50	0,125851	0,001027
100	0,104240	0,000873
500	0,075453	0,000675
1000	0,065941	0,000376
5000	0,051740	0,000408
10000	0,046834	0,000427
50000	0,037576	0,000686
100000	0,033374	0,001443

Anexo III

Se adjuntan los datos que aparecen representados en la *Figura 7*.

Entorno de Moore		
N	T_f	ΔT_f
20	0,217594	0,000010
30	0,217204	0,000007
40	0,217075	0,000005
50	0,217010	0,000013
60	0,216970	0,000011
70	0,216961	0,000010
80	0,216936	0,000009
90	0,216928	0,000008
100	0,216934	0,000018
110	0,216924	0,000018
120	0,216930	0,000019
130	0,216915	0,000019
140	0,216926	0,000016
150	0,216910	0,000021
160	0,216906	0,000016
170	0,216921	0,000013
180	0,216908	0,000010
190	0,216918	0,000011
200	0,216913	0,000012
25	0,217358	0,000008

Entorno de Von Neumann		
N	T_f	ΔT_f
20	0,207426	0,000040
30	0,206722	0,000029
40	0,206465	0,000023
50	0,206290	0,000058
60	0,206252	0,000046
70	0,206194	0,000044
80	0,206134	0,000039
90	0,206156	0,000032
100	0,206058	0,000096
110	0,206052	0,000079
120	0,206098	0,000064
130	0,206156	0,000089
140	0,206031	0,000085
150	0,206052	0,000068
160	0,206101	0,000052
170	0,206037	0,000061
180	0,206056	0,000054
190	0,206112	0,000065
200	0,206061	0,000046