Universidad Zaragoza
1542

Trabajo Fin de Grado (Apéndices)

Modelos dinámicos de haces de fibras. Comparación entre dos métodos de cálculo

Dynamical fiber-bundle models. Comparison between two approaches

Autor/es

Miguel Ángel Clemente Salvador

Director/es

Amalio Fernández-Pacheco Pérez

Javier Gómez Jiménez

Facultad / Escuela

Facultad de Ciencias

Año 2017

Repositorio de la Universidad de Zaragoza – Zaguan http://zaguan.unizar.es

Índice

1. Apéndice A

El código utilizado para las simulaciones en una dimensión es el siguiente:

```
/**
 *
 * Autor:      Miguel angel Clemente Salvador
 *            Grado en Fisica, cuarto Curso (TFG)
 *               Universidad de Zaragoza
 *
 * Proposito: el programa resuelve la dinamica de una cadena de N elementos,
 * controlada por la redistribucion local de la carga del elemento roto (LLS).
 * El objetivo es calcular el tiempo total de rotura de la cadena, comparando
 * la distribucion de tiempos de rotura para los dos metodos.
 *
 */

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

/*------Parametros globales-----------*/
#define N 10                // Numero de elementos
#define size_vect (int)N/2+1 // Tamano de los vectores de contaje. Como maximo, el
    numero de grietas distintas es N/2 (configuracion intercalada)
#define rho 2               // Exponente para el calculo de las cargas
#define sigma_0 1           // Carga inicial
#define N_simul 100000      // Numero de simulaciones total a realizar
#define N_interv_histo 50   // Numero de bins para el histograma

/*--Definir EXCLUSIVAMENTE uno de los dos metodos--*/
//#define Metodo_radiactivo
#define Metodo_clasico

/*---Structs---*/
/**
 * Contiene 2 enteros, cada uno guardara uno de los cumulos de los que
 * es vecino el elemento en cuestion. Si alguna de las componentes es 0,
 * se debe a que el elemento ya estaba roto, o bien es uno de los elementos
 * que no tienen vecinos (ninguno de los casos nos interesa para el recuento.
 */
struct almacenaje_vecinos{
    int cum1, cum2;
}; typedef struct almacenaje_vecinos Almacenaje_vecinos;
```

```
/*------Librerias------*/

#include "Librerias/Random.cpp"
#include "Librerias/Estadisticas_ficheros.cpp"
#include "Librerias/Inicializacion_vectores.cpp"
#include "Librerias/Reparto_cargas_1D.cpp"
#include "Librerias/Funciones_radiactivo_1D.cpp"
#include "Librerias/Funciones_clasico_1D.cpp"

/*-------------------Programa principal------------------*/
int main(){
    int simul;    // Indice para realizar cada simulacion
    int paso;     // Indice para cada paso temporal, o rotura de un elemento
    double T;     // Tiempo total transcurrido (parametro problema)
    double delta; // Tiempo incrementado en cada paso temporal, entre roturas de 2
        elementos

    double cargas_red[N];          // Vector que almacena las sigmas (cargas)
    int etiquetas[N];              // Vector que indexa las grietas (I)
    int rotos_grieta[size_vect];   // Vector que cuenta el numero de elementos rotos
        de cada grieta (R)
    int numero_vecinos[size_vect]; // Vector que cuenta el numero de vecinos no rotos
        de cada grieta (S)
    Almacenaje_vecinos vecinos[N]; // Vector donde se guardan los indices de los
        cumulos que son vecinos de cada elemento (V)
    int indice_elemento_roto;      // Indice del proximo elemento a romper
    int indice_nuevo;              // Indice a asignar a la proxima nueva grieta que
        aparezca

    #ifdef Metodo_clasico
    double tiempos_iniciales[N];   // Vector que almacena la distribucion inicial de
        tiempos
    double tiempos_nuevos[N];      // Vector que almacena los tiempos de rotura
        calculados en cada paso
    double tiempos_transcurridos[N]; // Vector que almacena el tiempo transcurrido
        para cada elemento (pesado con sigma)
    #endif // Metodo_clasico
    #ifdef Metodo_radiactivo
    double prob_rotura[N];                 // Vector que almacena la probabilidad de
        rotura de cada elemento
    #endif // Metodo_radiactivo

    FILE *fich_tiempos;                     // Fichero donde se escribe el resultado
        de T para cada simulacion (binario)
    char ruta_tiempos[]="tiempos_finales.dat"; // Ruta para el fichero fich_tiempos
    fich_tiempos=fopen(ruta_tiempos,"wb");
```

```
srand(time(NULL));

for(simul=0; simul<N_simul; simul++){   // Bucle para las 'N_simul' simulaciones
    /*---Inicializamos las variables a utilizar---*/
    indice_nuevo=1;
    inicializa_vector_double(cargas_red, sigma_0, N); // Todas las cargas valen
        signa_0 (parametro global)
    inicializa_vector(etiquetas, 0, N);            // Todas las etiquetas son 0,
        ya que los elementos estan sin romper
    inicializa_vector(rotos_grieta, 0, size_vect); // La componente 1 deberia ser
        N, pero no importa porque no se usa
    inicializa_vector(numero_vecinos, 0, size_vect);

    #ifdef Metodo_clasico
    inicializa_tiempos(tiempos_iniciales);         // Distribucion exponencial
        para los tiempos iniciales
    inicializa_vector_double(tiempos_transcurridos, 0, N);

    /*--- El primer paso lo hacemos fuera del bucle para aprovechar el vector de
        tiempos iniciales---*/
    delta=busca_minimo_vector(tiempos_iniciales, &indice_elemento_roto);
    T=delta;
    #endif // Metodo_clasico
    #ifdef Metodo_radiactivo
    delta=probabilidades_rotura(cargas_red, prob_rotura);
    T=delta;
    #endif // Metodo_radiactivo

    /* Bucle principal (metodo clasico):
       1) Se incrementa el tiempo transcurrido para cada elemento, con el valor de
           (delta*sigma^rho) (delta y sigma ya han sido calculadas en la iteracion
           anterior)
       2) Se rompe el elemento con menor tiempo (que ya habia sido obtenido al
           calcular delta)
       3) Se reindexan adecuadamente todas las grietas, igualando indices de las
           contiguas
       4) Se buscan todos los elementos que son vecinos de alguna de las grietas,
           y se cuentan separadamente para cada una de ellas. Ademas, se almacenan
           en el struct 'vecinos' los indices de las grietas que son vecinas a cada
           uno de los elementos, si las hay.
       5) Se calculan las cargas de red a partir de la informacion almacenada en
           el struct 'vecinos' y del numero de elementos rotos y vecinos de cada
           grieta.
       6) Se recalculan los tiempos de rotura a partir de los tiempos iniciales,
           los tiempos transcurridos y el nuevo valor de la carga soportada por
```

```
            cada elemento.
          7) Se busca el minimo valor del tiempo (guardando tambien las coordenadas
              del elemento para romperlo) y se incrementa el valor del tiempo total
              transcurrido.
         *
         * Bucle principal (metodo radiactivo):
          1) Se calcula el elemento que va a romperse en esta iteracion, a partir de
              las probabilidades de rotura (calculadas en la iteracion anterior)
          2) Se rompe el elemento, se reindexan las grietas y se reparten las cargas
              (puntos 3-5 anteriores)
          3) Se calculan las nuevas probabilidades de rotura (gammas) y la vida de la
              configuracion actual, que se suma al tiempo total transcurrido
         */
        for(paso=1; paso<N; paso++){     // El primer paso se realiza antes del bucle
            #ifdef Metodo_clasico
            incrementa_tiempos(tiempos_transcurridos, cargas_red, delta);
            /*---Se rompe el elemento y se reparten las cargas---*/
            reindexa_grietas(etiquetas, rotos_grieta, indice_elemento_roto,
                &indice_nuevo);
            encuentra_vecinos(etiquetas, numero_vecinos, vecinos);
            calcula_cargas(cargas_red, etiquetas, vecinos, rotos_grieta,
                numero_vecinos);

            recalcula_tiempos(tiempos_nuevos, tiempos_iniciales,
                tiempos_transcurridos, cargas_red);
            delta=busca_minimo_vector(tiempos_nuevos, &indice_elemento_roto);
            T+=delta;
            #endif // Metodo_clasico

            #ifdef Metodo_radiactivo
            indice_elemento_roto=calcula_elemento_roto(prob_rotura);
            /*---Se rompe el elemento y se reparten las cargas---*/
            reindexa_grietas(etiquetas, rotos_grieta, indice_elemento_roto,
                &indice_nuevo);
            encuentra_vecinos(etiquetas, numero_vecinos, vecinos);
            calcula_cargas(cargas_red, etiquetas, vecinos, rotos_grieta,
                numero_vecinos);

            delta=probabilidades_rotura(cargas_red, prob_rotura);
            T+=delta;
            #endif // Metodo_radiactivo
        }
        fwrite(&T, sizeof(double), 1, fich_tiempos); // Escribimos en un fichero el
            resultado, para analizarlo despues
        if(((100*simul)%N_simul)==0){       // Escribimos el progreso por pantalla
            printf("%d/100 completado\n", (int)(100.0*simul/N_simul));
```

```
        }
    }
    fclose(fich_tiempos);
    printf("100/100 completado\n");

    /*---Analisis estadistico---*/
    double T_med, T_var;
    med_var(ruta_tiempos, N_simul, &T_med, &T_var, 1); // Se obtiene la media y
        varianza de los resultados
    printf("<T>=%lf\n Varianza=%.12lf\n", T_med, T_var);

    FILE *medias_var;                       // Fichero donde se escriben los
        resultados finales de la media y la varianza
    char ruta_histograma[60];               // Ruta para el fichero de histograma

    #ifdef Metodo_clasico
    char ruta_medias[]="1D/medias_varianzas_clasico.txt"; // Ruta para el fichero
        medias_var
    sprintf(ruta_histograma,"1D/histogramas_clasico/histog_rho=%d_N=%d_sim=%d.txt",
        rho, N, N_simul);
    #endif // Metodo_clasico

    #ifdef Metodo_radiactivo
    char ruta_medias[]="1D/medias_varianzas_radiactivo.txt"; // Ruta para el fichero
        medias_var
    sprintf(ruta_histograma,"1D/histogramas_radiactivo/histog_rho=%d_N=%d_sim=%d.txt",
        rho, N, N_simul);
    #endif // Metodo_radiactivo

    histograma(ruta_tiempos, N_simul, N_interv_histo, ruta_histograma, 1); // Se
        obtiene el histograma
    medias_var=fopen(ruta_medias, "at");
    fprintf(medias_var, "%d\t%d\t%d\t%lf\t%.12lf\n", N, rho, N_simul, T_med, T_var);
    fclose(medias_var);

    return 1;
}

}
```

Las librerías utilizadas para el código 1D son las siguientes:

Reparto_cargas_1D.cpp:

```
/**
 * Dada la matriz que contiene los indices de cada grieta (argumento 1) y el indice
    del punto que se ha roto (argumento 3), comprueba si este elemento es vecino de
```

```
    alguna de las grietas, en cuyo caso le asigna el indice de una de ellas (orden de
    prioridad: left, right). Ademas, invoca a una de las funciones
    'iguala_indices_ngrietas' para que todas las grietas que ahora se han puesto en
    contacto tengan este mismo indice. Finalmente, se modifica adecuadamente el
    segundo parametro, que da cuenta del numero de elementos rotos que tiene cada
    grieta. El ultimo parametro solo se incrementa si el elemento roto ha formado una
    nueva grieta.
*/
void reindexa_grietas(int vector_indices[N], int *rotos_grieta, int roto, int
    *indice_nuevo){
   /*---Calculamos los indices de los elementos adyacentes, respetando las
       condiciones periodicas---*/
   int ind_anterior, ind_siguiente;

   ind_anterior=(roto+2*N-1)%N; // Si roto es 0, ind_anterior=N-1; en caso
       contrario, ind_anterior=roto-1
   ind_siguiente=(roto+N+1)%N;  // Si roto es N-1, ind_siguiente=0; en caso
       contrario, ind_siguiente=roto+1

   /*---Calculamos las etiquetas de los elementos adyacentes---*/
   int left, right, indice_asignado; // Indice_asignado: el que va a perdurar, y el
       que se asocia al nuevo roto
   int cambia;                       // Indice a sustituir por indice_asignado en la
       reindexacion
   left=vector_indices[ind_anterior];
   right=vector_indices[ind_siguiente];

   /*---Distinguimos casos segun el numero de vecinos rotos---*/
   switch (!!left + !!right) {  // '!!a' devuelve 1 si a!=0, 0 si a=0. Esto da el #
       de vecinos que no son 0
     case 0:  // Nueva grieta
       indice_asignado=*indice_nuevo;
       (*indice_nuevo)++;                      // La siguiente grieta nueva estara
           asociada al siguiente indice nuevo
     break;
     case 1:  // El elemento roto es vecino de una sola grieta. Se le asocia el
         indice no nulo.
       if(left!=0){
           indice_asignado=left;
       }else if(right!=0){
           indice_asignado=right;
       }
     break;
     case 2:  // El elemento roto es vecino de 2 grietas. Se le asocia el indice!=0,
         respetando prioridad (left, right)
       if(left!=0){
```

```
            indice_asignado=left;
            cambia=right;
        }else{
            indice_asignado=right;
            cambia=left;
        }
        /*--- Se igualan los indices de ambas grietas---*/
        if(cambia!=indice_asignado){ // Si es igual, no hay que hacer nada
        int ncambiados=0;
        for(int i=0; i<N; i++){
            if(vector_indices[i]==cambia){ // Cambiamos los del indice a cambiar por
                el indice a asignar
                vector_indices[i]=indice_asignado;
                ncambiados++;
            }
        }
        rotos_grieta[indice_asignado]+=ncambiados;
        rotos_grieta[cambia]=0;
    }
      break;
    }
    vector_indices[roto]=indice_asignado; // Se asigna el indice adecuado al nuevo
        elemento roto
    rotos_grieta[indice_asignado]++;     // Nuevo elemento para la grieta con la que
        esta en contacto
}

/**
 * Escanea el vector de etiquetas (argumento 1) y comprueba para cada elemento si los
     colindantes son vecinos de uno o varios cumulos, informacion que se guarda en el
     ultimo argumento: cada una de las 2 componentes de los elementos del vector puede
     guardar el indice del cumulo o cumulos de los que es vecino cada elemento. Si el
     elemento ya esta roto, le asignamos el valor de 0, para ignorarlo en el calculo
     de las cargas. Ademas, el algoritmo calcula el numero de vecinos que tiene cada
     cumulo (segundo argumento).
 */
void encuentra_vecinos(int vector_indices[N], int numero_vecinos[size_vect],
    Almacenaje_vecinos vecinos[N]){
    int left, right;
    int indice_left, indice_right;
    inicializa_vector(numero_vecinos, 0, size_vect); // El numero de vecinos se
        calcula cada vez que se ejecuta esta funcion

    /*---Leemos el vector:---*/
    for(int i=0; i<N; i++){
        left=(i-1+2*N)%N;
```

```
        right=(i+1+N)%N;
        if(vector_indices[i]!=0){ // Los elementos ya rotos, como no tienen carga, no
            los utilizaremos
            vecinos[i].cum1=0;
            vecinos[i].cum2=0;
        }else{                        // Si es 0, puede ser vecino de algun cumulo
            indice_left=vector_indices[left];
            indice_right=vector_indices[right];

            /* Guardamos los indices de los cumulos contiguos al elemento i. Si alguna
                es 0, se guarda un 0, que al no corresponder a ningun cumulo, significa
                que en realidad no tiene vecino en esa direccion.
             *
             * Incrementamos el numero de vecinos de la(s) grieta(s) dada(s) por
                 el(los) indice(s) del (de los) elemento(s) contiguo(s). Si alguno es
                 0, se incrementa el numero de vecinos del "cumulo 0", que es el cumulo
                 de los elementos no rotos y no se utiliza para calcular las cargas.
             */

            vecinos[i].cum1=indice_left;
            numero_vecinos[indice_left]++;
            if(indice_right!=indice_left){
                vecinos[i].cum2=indice_right;
                numero_vecinos[indice_right]++;
            }else{
                vecinos[i].cum2=0;
            }
        }
    }
}

/**
 * Calcula las cargas de la red mediante 1+suma de (r_k/s_k)*(1-delta(vecino, 0)),
    siendo {k} los 2 elementos contiguos al punto en cuestion, r el numero de
    elementos rotos de cada grieta y s el numero de vecinos de cada grieta. Es decir,
    si el elemento es vecino de una o mas grietas, el exceso de carga que soporta es
    r/s por cada grieta adyacente, siendo aditivas (los vecinos de dos grietas
    soportan mas carga que los vecinos de una sola). La carga es 1+este exceso.
 * Primer parametro: vector con las cargas de la red (salida del subalgoritmo).
 * Segundo parametro: vector de etiquetas de las grietas
 * Tercer parametro: vector de structs donde se almacenan los indices de los cumulos
    que son vecinos de cada elemento
 * Cuarto argumento: vector donde se almacena el numero de elementos rotos de cada
    grieta
 * Quinto elemento: vector donde se almacena el numero de vecinos de cada grieta
 */
```

```
void calcula_cargas(double cargas_red[N], int vector_etiquetas[N], Almacenaje_vecinos
    vecinos[N], int rotos_grieta[size_vect], int numero_vecinos[size_vect]){
    int etiqueta;
    for(int i=0; i<N; i++){
        etiqueta=vector_etiquetas[i];
        if(etiqueta!=0){    // Elemento ya roto, no soporta carga
            cargas_red[i]=0;
        }else{              // Elemento sin romper
            /* Incrementamos las cargas, como la suma (restringida a los vecinos no
                nulos), mediante: r_k/s_k siendo r el numero de elementos rotos de la
                grieta y s el numero de vecinos de la grieta.
             */
            cargas_red[i]=1;
            if((vecinos[i].cum1)!=0){
                cargas_red[i]+=((double)rotos_grieta[vecinos[i].cum1]/
                        (numero_vecinos[vecinos[i].cum1]));
            }
            if((vecinos[i].cum2)!=0){
                cargas_red[i]+=((double)rotos_grieta[vecinos[i].cum2]/
                        (numero_vecinos[vecinos[i].cum2]));
            }
        }
    }
}
```

Funciones_clasico_1D.cpp:

```
/**
 * Asocia a todos los elementos del vector (primer argumento) un numero aleatorio
     dado por la distribucion exponencial, es decir, un valor de -log(random entre 0 y
     1)
 */
void inicializa_tiempos(double vect[N]){
    for(int i=0; i<N; i++){
        vect[i]=-log(random_01());
    }
}

/**
 * Se lee un vector y se devuelve (por referencia) el indice del elemento de minimo
     valor positivo, asi como este valor
 */
double busca_minimo_vector(double vect[N], int *indice){
    double minimo=vect[0];
    *indice=0;
```

```
    for(int i=0; i<N; i++){
        if(minimo<=0){
            minimo=vect[i];  //Buscamos primero el tiempo de un elemento no roto, para
                comparar
            *indice=i;
        }
        if(vect[i]<minimo && vect[i]>0){
            minimo=vect[i];  //Buscamos el minimo tiempo de un elemento no roto
            *indice=i;
        }
    }
    return minimo;
}

/**
 * Se actualizan los tiempos transcurridos para cada elemento (argumento 1), a partir
     de las cargas de red (argumento 2) y del intervalo en que las han soportado
     (argumento 3).
 */
void incrementa_tiempos(double tiempos_transcurridos[N], double cargas[N], double
    delta){
    for(int i=0; i<N; i++){
        tiempos_transcurridos[i]+=delta*pow(cargas[i], rho);
    }
}

/**
 * Se devuelven los nuevos tiempos de rotura para cada elemento (argumento 1),
     calculados a partir de los tiempos iniciales (argumento 2), de los tiempos
     transcurridos (argumento 3) y de las cargas de red (argumento 4).
 */
void recalcula_tiempos(double tiempos_nuevos[N], double tiempos_iniciales[N],
                    double tiempos_transcurridos[N], double cargas[N]){
    for(int i=0; i<N; i++){
        if(cargas[i]>0){
            tiempos_nuevos[i]=(tiempos_iniciales[i]-tiempos_transcurridos[i])/
                 (pow(cargas[i], rho));
        }else{     //Para elementos rotos, asignamos el valor de 0, que no esta
            contemplado a la hora de buscar el minimo tiempo
            tiempos_nuevos[i]=0;
        }
    }
}
```

Funciones_radiactivo_1D.cpp:

```
/**
 * Calcula la probabilidad de ruptura de cada elemento a partir del
 * vector de red (primer parametro) y la almacena en un vector (segundo parametro).
 * Ademas, devuelve la delta de la configuracion, necesaria para calcular T
 */
double probabilidades_ruptura(double *red, double *prob){
    double gamma_total=0;    // Tiempo de vida de la configuracion actual
    double gamma[N];         // Vector para el calculo de las probabilidades

    for(int i=0; i<N; i++){ // Calculo de las gammas
        gamma[i]=pow((red[i]),rho);
        gamma_total+=gamma[i];
    }
    for(int i=0; i<N; i++){ // Calculo de las probabilidades
        prob[i]=(gamma[i])/(gamma_total);
    }

    return 1/gamma_total;   // Se devuelve el valor de delta
}

/**
 * A partir del vector de probabilidades, lanza un numero entre 0 y 1 y determina
 * el siguiente elemento en romperse, que develve con su indice
 */
int calcula_elemento_roto(double *probabilidades){
    double numero=random_01();
    double acum_prob=0;
    int indice;

    /*
     * El intervalo [0,1) se divide en segmentos, cada uno de una longitud
     * igual a la probabilidad de ruptura del elemento. El numero entre 0 y 1
     * determina el segmento en el que cae, al cual se le asocia el indice, que
     * es el parametro de salida
     */
    for(indice=0;indice<N;indice++){
        acum_prob+=probabilidades[indice];
        if(acum_prob>numero){
            return indice;    // El indice se devuelve cuando el numero esta entre dos
                separaciones de segmentos
        }
    }
};
```

El código utilizado para las simulaciones en dos dimensiones es:

```c
/**
 *
 * Autor:      Miguel Angel Clemente Salvador
 *            Grado en Fisica, cuarto Curso (TFG)
 *               Universidad de Zaragoza
 *
 * Proposito: el programa resuelve la dinamica de una red de NxM elementos,
    controlada por la redistribucion local de la carga del elemento roto (LLS). El
    objetivo es calcular el tiempo total de rotura de la red, comparando la
    distribucion de tiempos de rotura para los dos metodos, y ver si el tiempo de
    rotura no es nulo para N, M tendiendo a infinito.
 *
 */

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

/*------Parametros globales-----------*/
#define N 50                 // Numero de elementos (en Y): numero de filas
#define M 50                 // Numero de elementos (en X): numero de columnas
#define size_vect (int)N*M/2+1 // Tamano de los vectores que se utilizaran
#define rho 2                // Indice de heterogeneidad
#define sigma_0 1            // Carga inicial
#define N_simul 100000       // Numero total de simulaciones a realizar
#define N_interv_histo 50    // Numero de bins para el histograma
#define animacion            // Para obtener una animacion del proceso de rotura de
    las grietas
#define evol_tamano          // Para observar como crece el tamano medio de las
    grietas con el tiempo

//#define Metodo_radiactivo
#define Metodo_clasico

/*---Structs---*/
/**
 * Contiene 4 enteros, cada uno guardara uno de los cumulos de los que
 * es vecino el elemento en cuestion. Si alguna de las componentes es 0,
 * se debe a que el elemento ya estaba roto, o bien es uno de los elementos
 * que no tienen vecinos (ninguno de los casos nos interesa para el recuento).
 */
struct almacenaje_vecinos{
    int cum1, cum2, cum3, cum4;
}; typedef struct almacenaje_vecinos Almacenaje_vecinos;
```

```
/*---Declaracion de funciones y librerias---*/

#include "Librerias/Random.cpp"
#include "Librerias/Estadisticas_ficheros.cpp"
#include "Librerias/Inicializacion_arrays.cpp"
#include "Librerias/Reparto_cargas.cpp"
#include "Librerias/Funciones_radiactivo.cpp"
#include "Librerias/Funciones_clasico.cpp"

/**
  * En caso de estar activado el modo animacion, escribe en un fichero de texto el
      estado de cada elemento (roto=2, vecino de una o varias grietas=1, ni roto ni
      vecino=0) en cada paso temporal, para hacer una animacion de la ruptura de la
      grieta
  */
void escribe_animacion(int [N][M], Almacenaje_vecinos [N][M], int);

/**
  * Devuelve el tamano medio de las grietas en un instante dado, a partir del
  * vector que guarda el numero de elementos rotos en cada grieta
  */
double calcula_tamano_medio(int *);

/*-------------------Programa principal------------------*/
int main(){
    int simul;    // Indice para realizar cada simulacion
    int paso;     // Indice para cada paso temporal, o ruptura de un elemento
    double T;     // Tiempo total transcurrido (parametro problema)
    double delta; // Tiempo incrementado en cada paso temporal, entre rupturas de 2
        elementos

    double cargas_red[N][M];                // Matriz que almacena las sigmas (cargas)
    int matriz_etiquetas[N][M];             // Matriz donde se indexan las grietas (I)
    int rotos_grieta[size_vect];            // Vector para guardar el numero de
        elementos rotos de cada grieta (R)
    int numero_vecinos[size_vect];          // Vector para guardar el numero de
        vecinos de cada grieta (S)
    Almacenaje_vecinos vecinos[N][M];       // Matriz donde se guardan los indices de
        los cumulos que son vecinos de cada elemento (V)
    int fila_roto, columna_roto;            // Indices del elemento a romper
    int indice_nuevo;                       // Indice de la proxima grieta a indexarse

    #ifdef evol_tamano
    double tamano_medio[N*M];               // Vector que guarda el tamano medio de
        las grietas en cada paso de tiempo
```

```
inicializa_vector_double(tamano_medio, 0, N*M);
#endif // evol_tamano

#ifdef Metodo_clasico
double tiempos_iniciales[N][M];         // Matriz que almacena la distribucion
    inicial de tiempos
double tiempos_nuevos[N][M];            // Matriz que almacena los tiempos de
    ruptura calculados en cada paso
double tiempos_transcurridos[N][M];     // Matriz que almacena el tiempo
    transcurrido para cada elemento (pesado con sigma)
#endif // Metodo_clasico
#ifdef Metodo_radiactivo
double prob_ruptura[N][M];              // Matriz que almacena la probabilidad de
    ruptura de cada elemento
#endif // Metodo_radiactivo

FILE *fich_tiempos;                     // Fichero donde se escribe el resultado
    de T para cada simulacion (binario)
char ruta_tiempos[]="tiempos_finales.dat"; // Ruta para el fichero fich_tiempos
fich_tiempos=fopen(ruta_tiempos,"wb");

srand(time(NULL));

for(simul=0; simul<N_simul; simul++){   // Bucle para las 'N_simul' simulaciones
    /*---Inicializamos las variables a utilizar---*/
    indice_nuevo=1;
    inicializa_matriz_double(cargas_red, sigma_0); // Todas las cargas valen
        signa_0 (parametro global)
    inicializa_matriz(matriz_etiquetas, 0);        // Todas las etiquetas son 0,
        ya que los elementos estan sin romper
    inicializa_vector(rotos_grieta, 0, size_vect); // La componente 1 deberia ser
        N*M, pero no importa porque no se usa
    inicializa_vector(numero_vecinos, 0, size_vect);

    #ifdef evol_tamano
    tamano_medio[0]+=calcula_tamano_medio(rotos_grieta);
    #endif // evol_tamano

    #ifdef Metodo_clasico
    inicializa_tiempos(tiempos_iniciales);         // Distribucion exponencial
        para los tiempos iniciales
    inicializa_matriz_double(tiempos_transcurridos, 0);

    /*--- El primer paso lo hacemos fuera del bucle para aprovechar el vector de
        tiempos iniciales---*/
    delta=busca_minimo_matriz(tiempos_iniciales, &fila_roto, &columna_roto);
```

```
      T=delta;
      #endif // Metodo_clasico
      #ifdef Metodo_radiactivo
      delta=probabilidades_ruptura(cargas_red, prob_ruptura);
      T=delta;
      #endif // Metodo_radiactivo

      #ifdef animacion
      if(simul==0){ // Hacemos la animacion solo para la primera simulacion
          escribe_animacion(matriz_etiquetas, vecinos, 0);
      }
      #endif // animacion

      /* Bucle principal (metodo clasico):
         1) Se incrementa el tiempo transcurrido para cada elemento, con el valor de
            (delta*sigma^rho) (delta y sigma ya han sido calculadas en la iteracion
            anterior)
         2) Se rompe el elemento con menor tiempo (que ya habia sido obtenido al
            calcular delta)
         3) Se reindexan adecuadamente todas las grietas, igualando indices de las
            contiguas
         4) Se buscan todos los elementos que son vecinos de alguna de las grietas,
            y se cuentan separadamente para cada una de ellas. Ademas, se almacenan
            en el struct 'vecinos' los indices de las grietas que son vecinas a cada
            uno de los elementos, si las hay.
         5) Se calculan las cargas de red a partir de la informacion almacenada en
            el struct 'vecinos' y del numero de elementos rotos y vecinos de cada
            grieta.
         6) Se recalculan los tiempos de ruptura a partir de los tiempos iniciales,
            los tiempos transcurridos y el nuevo valor de la carga soportada por
            cada elemento.
         7) Se busca el minimo valor del tiempo (guardando tambien las coordenadas
            del elemento para romperlo) y se incrementa el valor del tiempo total
            transcurrido.
       *
       * Bucle principal (metodo radiactivo):
         1) Se calcula el elemento que va a romperse en esta iteracion, a partir de
            las probabilidades de ruptura (calculadas en la iteracion anterior)
         2) Se rompe el elemento, se reindexan las grietas y se reparten las cargas
            (puntos 3-5 anteriores)
         3) Se calculan las nuevas probabilidades de ruptura (gammas) y la vida de
            la configuracion actual, que se suma al tiempo total transcurrido
       */
      for(paso=1; paso<N*M; paso++){    // El primer paso se realiza antes del bucle
         #ifdef Metodo_clasico
         incrementa_tiempos(tiempos_transcurridos, cargas_red, delta);
```

```
        /*---Se rompe el elemento y se reparten las cargas---*/
        reindexa_grietas(matriz_etiquetas, rotos_grieta, columna_roto, fila_roto,
            &indice_nuevo);
        encuentra_vecinos_2D(matriz_etiquetas, numero_vecinos, vecinos);
        calcula_cargas_red_2D(cargas_red, matriz_etiquetas, vecinos, rotos_grieta,
            numero_vecinos);

        recalcula_tiempos(tiempos_nuevos, tiempos_iniciales,
            tiempos_transcurridos, cargas_red);
        delta=busca_minimo_matriz(tiempos_nuevos, &fila_roto, &columna_roto);
        T+=delta;
        #endif // Metodo_clasico

        #ifdef Metodo_radiactivo
        calcula_elemento_roto(prob_ruptura, &fila_roto, &columna_roto);
        /*---Se rompe el elemento y se reparten las cargas---*/
        reindexa_grietas(matriz_etiquetas, rotos_grieta, columna_roto, fila_roto,
            &indice_nuevo);
        encuentra_vecinos_2D(matriz_etiquetas, numero_vecinos, vecinos);
        calcula_cargas_red_2D(cargas_red, matriz_etiquetas, vecinos, rotos_grieta,
            numero_vecinos);

        delta=probabilidades_ruptura(cargas_red, prob_ruptura);
        T+=delta;
        #endif // Metodo radiactivo

        #ifdef animacion
        if(simul==0){ // Hacemos la animacion solo para la primera simulacion
            escribe_animacion(matriz_etiquetas, vecinos, paso);
        }
        #endif // animacion
        #ifdef evol_tamano
        tamano_medio[paso]+=calcula_tamano_medio(rotos_grieta);
        #endif // evol_tamano
    }
    fwrite(&T, sizeof(double), 1, fich_tiempos); // Escribimos en un fichero el
        resultado, para analizarlo despues
    if(((100*simul)%N_simul)==0){              // Escribimos el progreso por
        pantalla
        printf("%d/100 completado\n", (int)(100.0*simul/N_simul));
    }
}
fclose(fich_tiempos);
printf("100/100 completado\n");

/*--------Analisis estadistico--------*/
```

```
double T_med, T_var;
med_var(ruta_tiempos, N_simul, &T_med, &T_var, 1); // Se obtiene la media y
    varianza de los resultados
printf("<T>=%lf\n Varianza=%.12lf\n", T_med, T_var);

FILE *medias_var;                       // Fichero donde se escriben los
    resultados finales de la media y la varianza
char ruta_medias[60];                   // Ruta para el fichero medias_var
char ruta_histograma[60];               // Ruta para el fichero de histograma
if(N==1 || M==1){
    #ifdef Metodo_clasico
    sprintf(ruta_medias,"1D/medias_varianzas_clasico.txt");
    sprintf(ruta_histograma,"1D/histogramas_clasico/histog_rho=%d_%dx%d_sim=%d.txt",
        rho, N, M, N_simul);
    #endif // Metodo_clasico

    #ifdef Metodo_radiactivo
    sprintf(ruta_medias,"1D/medias_varianzas_radiactivo.txt");
    sprintf(ruta_histograma,
        "1D/histogramas_radiactivo/histog_rho=%d_%dx%d_sim=%d.txt", rho, N, M,
            N_simul);
    #endif // Metodo_radiactivo
}else{
    #ifdef Metodo_clasico
    sprintf(ruta_medias,"2D/medias_varianzas_clasico.txt");
    sprintf(ruta_histograma,
        "2D/histogramas_clasico/histog_rho=%d_%dx%d_sim=%d.txt", rho, N, M,
            N_simul);
    #endif // Metodo_clasico

    #ifdef Metodo_radiactivo
    sprintf(ruta_medias,"2D/medias_varianzas_radiactivo.txt");
    sprintf(ruta_histograma,
        "2D/histogramas_radiactivo/histog_rho=%d_%dx%d_sim=%d.txt", rho, N, M,
            N_simul);
    #endif // Metodo_radiactivo
}

medias_var=fopen(ruta_medias, "at");
fprintf(medias_var, "%d\t%d\t%d\t%d\t%lf\t%.12lf\n", N, M, rho, N_simul, T_med,
    T_var);
fclose(medias_var);

histograma(ruta_tiempos, N_simul, N_interv_histo, ruta_histograma, 1); // Se
    obtiene el histograma
```

```
    #ifdef evol_tamano
    FILE *fich_tamano;                      // Fichero donde se escribira la evolucion
        del tamano medio de las grietas
    char ruta_tamano[60];
    #ifdef Metodo_clasico
    sprintf(ruta_tamano, "Tamano/clasico/tamano_2D_N=%d_M=%d_rho=%d_simul=%d.txt",
        N, M, rho, N_simul);
    #endif // Metodo_clasico
    #ifdef Metodo_radiactivo
    sprintf(ruta_tamano,
        "Tamano/radiactivo/tamano_2D_N=%d_M=%d_rho=%d_simul=%d.txt", N, M, rho,
        N_simul);
    #endif
    fich_tamano=fopen(ruta_tamano, "w");
    for(int i=0; i<N*M; i++){
        fprintf(fich_tamano, "%d\t%lf\n", i, tamano_medio[i]/N_simul);
    }
    fclose(fich_tamano);
    #endif // evol_tamano

    return 1;
}

/*--------------Funciones------------------------------*/
/**
  * En caso de estar activado el modo animacion, escribe en un fichero de texto el
      estado
  * de cada elemento (roto=2, limite de una o varias grietas=1, ni roto ni vecino=0)
  * en cada paso temporal, para hacer una animacion de la ruptura de la grieta
  */
void escribe_animacion(int etiquetas[N][M], Almacenaje_vecinos vecinos[N][M], int
    paso){
    FILE *archivo_animacion;
    char ruta_animacion[60];
    #ifdef Metodo_radiactivo
    sprintf(ruta_animacion,
        "Animacion/Radiactivo/rho=%d_N=%d_M=%d/animacion_paso=%d.txt", rho, N, M,
        paso);
    #endif // Metodo_radiactivo
    #ifdef Metodo_clasico
    sprintf(ruta_animacion,
        "Animacion/Clasico/rho=%d_N=%d_M=%d/animacion_paso=%d.txt", rho, N, M, paso);
    #endif // Metodo_radiactivo
    archivo_animacion=fopen(ruta_animacion, "w");
    for(int i=0; i<N; i++){
```

```
        for(int j=0; j<M; j++){
            if(etiquetas[i][j]!=0){
                fprintf(archivo_animacion, "2\t"); // Elemento roto
            }else if(vecinos[i][j].cum1!=0 || vecinos[i][j].cum2!=0 ||
                vecinos[i][j].cum3!=0 || vecinos[i][j].cum4!=0){
                fprintf(archivo_animacion, "1\t"); // Elemento limite de una o varias
                    grietas
            }else{
                fprintf(archivo_animacion, "0\t"); // Elemento intacto (ni roto ni
                    vecino)
            }
        }
        fprintf(archivo_animacion, "\n");
    }
    fclose(archivo_animacion);
}

/**
  * Devuelve el tamano medio de las grietas en un instante dado, a partir del
  * vector que guarda el numero de elementos rotos en cada grieta
  */
double calcula_tamano_medio(int v[size_vect]){
    double suma=0;
    int contador=0;
    for(int i=0; i<size_vect; i++){
        if(v[i]!=0){
            suma+=v[i];
            contador++;
        }
    }
    return suma/contador;
}
```

Las librerías utilizadas para el código 2D son las siguientes:

Reparto_cargas_2D.cpp:

```
/**
 * En el caso de que el elemento roto sea vecino de 2 grietas, se debe asociar el
    mismo indice a las 2, ya que ahora estan en contacto. Por tanto, se asigna el
    valor del parametro 3 a los puntos de la matriz de indices (parametro 1) que
    tengan el valor indicado mediante el ultimo parametro. Si el indice ya es el
    asignado, no hace falta cambiarlo, por lo que la funcion no hace nada. Ademas,
    reagrupa el numero de elementos rotos de la grieta correspondiente a
    indice_asignado, sumandoles los elementos que hay en la grieta correspondiente al
    indice a cambiar, y volviendose 0 el numero de elementos de esta ultima.
```

```
 */
void iguala_indices_2grietas(int matriz[N][M], int *rotos_grieta, int
    indice_asignado, int cambia){
    if(cambia!=indice_asignado){ //Si es igual, no hay que hacer nada
        int ncambiados=0;
        for(int i=0; i<N; i++){
            for(int j=0; j<M; j++){
                if(matriz[i][j]==cambia){ // Cambiamos los del indice a cambiar por el
                    indice a asignar
                    matriz[i][j]=indice_asignado;
                    ncambiados++;
                }
            }
        }
        rotos_grieta[indice_asignado]+=ncambiados;
        rotos_grieta[cambia]=0;
    }
}

/**
 * En el caso de que el elemento roto sea vecino de 3 grietas, se debe asociar el
     mismo indice a las 3, ya que ahora estan en contacto. Por tanto, se asigna el
     valor del parametro 3 a los puntos de la matriz de indices (parametro 1) que
     tengan alguno de los valores indicados mediante los 2 ulimos parametros. Si
     alguno de los indices ya es el asignado, no hace falta cambiarlo, por lo que
     invoca a la funcion iguala_indices_2grietas con el otro indice (que en general es
     distinto del asignado). Ademas, reagrupa el numero de elementos rotos de la
     grieta correspondiente a indice_asignado, sumandoles los elementos que hay en las
     grietas correspondientes a cambia1 y cambia2, y volviendose 0 el numero de
     elementos de estas ultimas.
 */
void iguala_indices_3grietas(int matriz[N][M], int *rotos_grieta, int
    indice_asignado, int cambia1, int cambia2){
    if(indice_asignado==cambia1){        // Si el primero ya es igual, no hace falta
        reindexarlo
        iguala_indices_2grietas(matriz, rotos_grieta, indice_asignado, cambia2);
    }else if(indice_asignado==cambia2){  // Si el segundo ya es igual, no hace falta
        reindexarlo
        iguala_indices_2grietas(matriz, rotos_grieta, indice_asignado, cambia1);
    }else{
        int ncambiados=0;
        for(int i=0; i<N; i++){
            for(int j=0; j<M; j++){
                if(matriz[i][j]==cambia1 || matriz[i][j]==cambia2){
                    matriz[i][j]=indice_asignado;
                    ncambiados++;
```

```
            }
        }
    }
    rotos_grieta[indice_asignado]+=ncambiados;
    rotos_grieta[cambia1]=0;
    rotos_grieta[cambia2]=0;
  }
}

/**
 * En el caso de que el elemento roto sea vecino de 4 grietas, se debe asociar el
    mismo indice a las 4, ya que ahora estan en contacto. Por tanto, se asigna el
    valor del parametro 2 a los puntos de la matriz de indices (parametro 1) que
    tengan alguno de los valores indicados mediante los 3 ulimos parametros. Si
    alguno de los indices ya es el asignado, no hace falta cambiarlo, por lo que
    invoca a la funcion iguala_indices_3grietas con los otros dos indices (que en
    general son distintos del asignado). Ademas, reagrupa el numero de elementos
    rotos de la grieta correspondiente a indice_asignado, sumandoles los elementos
    que hay en las grietas correspondientes a cambia1, cambia2 y cambia3, y
    volviendose 0 el numero de elementos de estas ultimas.
 */
void iguala_indices_4grietas(int matriz[N][M], int *rotos_grieta, int
    indice_asignado, int cambia1, int cambia2, int cambia3){
   if(indice_asignado==cambia1){       // Si el tercero ya es el asignado no hace
       falta cambiarlo
       iguala_indices_3grietas(matriz, rotos_grieta, indice_asignado, cambia2,
           cambia3);
   }else if(indice_asignado==cambia2){ // Si el segundo ya es el asignado no hace
       falta cambiarlo
       iguala_indices_3grietas(matriz, rotos_grieta, indice_asignado, cambia1,
           cambia3);
   }else if(indice_asignado==cambia3){ // Si el tercero ya es el asignado no hace
       falta cambiarlo
           iguala_indices_3grietas(matriz, rotos_grieta, indice_asignado, cambia1,
               cambia2);
   }else{
       int ncambiados=0;
       for(int i=0; i<N; i++){
           for(int j=0; j<M; j++){
               if(matriz[i][j]==cambia1 || matriz[i][j]==cambia2 ||
                   matriz[i][j]==cambia3){
                   matriz[i][j]=indice_asignado;
                   ncambiados++;
               }
           }
       }
```

```
        rotos_grieta[indice_asignado]+=ncambiados;
        rotos_grieta[cambia1]=0;
        rotos_grieta[cambia2]=0;
        rotos_grieta[cambia3]=0;
    }
}

/**
 * Dada la matriz que contiene los indices de cada grieta (argumento 1) y las
     coordenadas del punto que se ha roto (argumentos 3 y 4), comprueba si este
     elemento es vecino de alguna de las grietas, en cuyo caso le asigna el indice de
     una de ellas (orden de prioridad: up, left, right, down). Ademas, invoca a una de
     las funciones 'iguala_indices_ngrietas' para que todas las grietas que ahora se
     han puesto en contacto tengan este mismo indice. Finalmente, se modifica
     adecuadamente el segundo parametro, que da cuenta del numero de elementos rotos
     que tiene cada grieta. El ultimo parametro solo se incrementa si el elemento roto
     ha formado una nueva grieta.
 */
void reindexa_grietas(int matriz_indices[N][M], int *rotos_grieta, int colum, int
    fila, int *indice_nuevo){
    /*---Calculamos las coordenadas de los elementos adyacentes, respetando las
        condiciones periodicas---*/
    int fila_up, fila_down, colum_left, colum_right;

    colum_left=(colum+2*M-1)%M;  // Si colum es 0, colum_left=M-1; en caso contrario,
        colum_left=colum-1
    fila_up=(fila+2*N-1)%N;      // Si fila es 0, fila_up=M-1; en caso contrario,
        fila_up=fila-1
    colum_right=(colum+M+1)%M;   // Si colum es M-1, colum_right=0; en caso contrario,
        colum_right=colum+1
    fila_down=(fila+N+1)%N;      // Si fila es N-1, fila_down=0; en caso contrario,
        fila_down=fila+1

    /*---Calculamos los indices de los elementos adyacentes---*/
    int up, down, left, right, indice_asignado; // Indice_asignado: el que va a
        perdurar, y el que se asocia al nuevo roto
    int cambia;                                 // Indice a sustituir por
        indice_asignado en la reindexacion
    up=matriz_indices[fila_up][colum];
    down=matriz_indices[fila_down][colum];
    left=matriz_indices[fila][colum_left];
    right=matriz_indices[fila][colum_right];

    /*---Distinguimos casos segun el numero de vecinos rotos---*/
    switch (!!up + !!left + !!down + !!right) { // '!!a' devuelve 1 si a!=0, 0 si a=0.
        Esto da el # de vecinos que no son 0
```

```
    case 0:  // Nueva grieta
      indice_asignado=*indice_nuevo;
      (*indice_nuevo)++;                    // La siguiente grieta nueva estara
          asociada al siguiente indice nuevo
    break;
    case 1:  // El elemento roto es vecino de una sola grieta. Se le asocia el
        indice no nulo.
      if(up!=0){
          indice_asignado=up;
      }else if(left!=0){
          indice_asignado=left;
      }else if(right!=0){
          indice_asignado=right;
      }else{
          indice_asignado=down;
      }
    break;
    case 2:  // El elemento roto es vecino de 2 grietas. Se le asocia el indice!=0,
        respetando prioridad (up, left, right, down)
      if(up!=0){
          indice_asignado=up;
          cambia=down+left+right;  // Alguno de estos 3 es no nulo, el resto son
              nulos
      }else if(left!=0){
          indice_asignado=left;
          cambia=down+right;       // Uno es nulo y el otro no
      }else{
          indice_asignado=right;
          cambia=down;
      }
      iguala_indices_2grietas(matriz_indices, rotos_grieta, indice_asignado,
          cambia); // Se igualan los indices de ambas grietas
    break;
    case 3:  // El elemento roto es vecino de 3 grietas. Se le asocia el indice!=0,
        respetando prioridad (up, left, right, down)
      if(up==0){
          indice_asignado=left;
          iguala_indices_3grietas(matriz_indices, rotos_grieta, indice_asignado,
              right, down); // Se igualan los indices de las 3 grietas
      }else{
          indice_asignado=up;
          if(left==0){
              iguala_indices_3grietas(matriz_indices, rotos_grieta, indice_asignado,
                  right, down);
          }else{
              if(right==0){
```

```
                iguala_indices_3grietas(matriz_indices, rotos_grieta,
                    indice_asignado, left, down);
            }else{
                iguala_indices_3grietas(matriz_indices, rotos_grieta,
                    indice_asignado, right, left);
            }
        }
      }
     break;
     case 4:  // El elemento roto es vecino de 4 cumulos .Se le asocia el indice!=0,
         respetando prioridad (up, left, right, down)
       indice_asignado=up;
       iguala_indices_4grietas(matriz_indices, rotos_grieta, indice_asignado, down,
           left, right); // Se igualan los indices de las 4 grietas
   }
   matriz_indices[fila][colum]=indice_asignado; // Se asigna el indice adecuado al
       nuevo elemento roto
   rotos_grieta[indice_asignado]++;           // Nuevo elemento para la grieta con
       la que esta en contacto
}

/**
 * Escanea la matriz que indexa a las grietas (argumento 1) y comprueba para cada
    elemento si los colindantes son vecinos de uno o varios cumulos, informacion que
    se guarda en el ultimo argumento: cada una de las 4 componentes de los elementos
    de matriz puede guardar el indice del cumulo o cumulos de los que es vecino cada
    elemento. Si el elemento ya es uno de los que estan rotos, ponemos que este "es
    vecino del cumulo de los elementos sin romper" (0), para ignorarlos en el calculo
    de las cargas. Ademas, el algoritmo calcula el numero de vecinos que tiene cada
    cumulo (segundo argumento).
 */
void encuentra_vecinos_2D(int matriz_indices[N][M], int numero_vecinos[size_vect],
    Almacenaje_vecinos vecinos[N][M]){
   int col_left, col_right, fila_up, fila_down;
   int indice_up, indice_left, indice_right, indice_down;
   inicializa_vector(numero_vecinos, 0, size_vect); // El numero de vecinos se
       calcula cada vez que se ejecuta esta funcion

   /*---Leemos la matriz:---*/
   for(int i=0; i<N; i++){
      fila_up=(i-1+2*N)%N;
      fila_down=(i+1+N)%N;
      for(int j=0; j<M; j++){
         col_left=(j-1+2*M)%M;
         col_right=(j+1+M)%M;
         if(matriz_indices[i][j]!=0){ // Los elementos ya rotos, como no tienen
```

```
            carga, no los utilizaremos
            vecinos[i][j].cum1=0;
            vecinos[i][j].cum2=0;
            vecinos[i][j].cum3=0;
            vecinos[i][j].cum4=0;
        }else{                      // Si no es 0, no es un vecino, sino un
            elemento roto
            indice_up=matriz_indices[fila_up][j];
            indice_left=matriz_indices[i][col_left];
            indice_right=matriz_indices[i][col_right];
            indice_down=matriz_indices[fila_down][j];

            /* Guardamos los indices de los cumulos contiguos al elemento i, j. Si
                alguna es 0, se guarda un 0, que al no corresponder a ningun
                cumulo, significa que en realidad no tiene vecino en esa direccion.
             *
             * Incrementamos el numero de vecinos de la(s) grieta(s) dada(s) por
                el(los) indice(s) del (de los) elemento(s) contiguo(s). Si alguno
                es 0, se incrementa el numero de vecinos del "cumulo 0", que es el
                cumulo de los elementos no rotos y no se utiliza para calcular las
                cargas.
             */

            vecinos[i][j].cum1=indice_up;
            numero_vecinos[indice_up]++;
            if(indice_left!=indice_up){
                vecinos[i][j].cum2=indice_left;
                numero_vecinos[indice_left]++;
                if(indice_right!=indice_up && indice_right!=indice_left){
                    vecinos[i][j].cum3=indice_right;
                    numero_vecinos[indice_right]++;
                    if(indice_down!=indice_up && indice_down!=indice_left &&
                        indice_down!=indice_right){
                        vecinos[i][j].cum4=indice_down;
                        numero_vecinos[indice_down]++;
                    }else{
                        vecinos[i][j].cum4=0;    // Si es doblemente vecino de
                            alguno, solo lo guardamos 1 vez
                    }
                }else{
                    if(indice_down!=indice_up && indice_down!= indice_left){
                        vecinos[i][j].cum3=indice_down;
                        numero_vecinos[indice_down]++;
                    }else{
                        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;
            }
        }
    }
}

/**
 * Calcula las cargas de la red mediante 1+suma de (r_k/s_k)*(1-delta(vecino, 0))
    siendo {k} los 4 elementos contiguos al punto en cuestion, r el numero de
    elementos rotos de cada grieta y s el numero de vecinos de cada grieta. Es decir,
    si el elemento es vecino de una o mas grietas, el exceso de carga que soporta es
    r/s por cada grieta adyacente, siendo aditivas (los vecinos de mas de una grieta
    soportan mas carga que los vecinos de una sola). La carga es 1+este exceso.
 * Primer parametro: matriz con las cargas de la red (salida del subalgoritmo).
 * Segundo parametro: matriz donde se han indexado las grietas
 * Tercer parametro: matriz de structs donde se almacenan los indices de los cumulos
    que son vecinos de cada elemento
 * Cuarto argumento: vector donde se almacena el numero de elementos rotos de cada
    grieta
 * Quinto elemento: vector donde se almacena el numero de vecinos de cada grieta
 */
void calcula_cargas_red_2D(double cargas_red[N][M], int matriz_etiquetas[N][M],
   Almacenaje_vecinos vecinos[N][M],
                           int rotos_grieta[size_vect], int numero_vecinos[size_vect]){
```

```
    int etiqueta;
    for(int i=0; i<N; i++){
        for(int j=0; j<M; j++){
            etiqueta=matriz_etiquetas[i][j];
            if(etiqueta!=0){    // Elemento ya roto, no soporta carga
                cargas_red[i][j]=0;
            }else{              // Elemento sin romper
                /* Incrementamos las cargas, como la suma (restringida a los vecinos no
                    nulos), mediante: r_k/s_k, siendo r el numero de elementos rotos de
                    la grieta y s el numero de vecinos de la grieta.
                 */
                cargas_red[i][j]=1;
                if((vecinos[i][j].cum1)!=0){
                    cargas_red[i][j]+=((double)rotos_grieta[vecinos[i][j].cum1]/
                            (numero_vecinos[vecinos[i][j].cum1]));
                }
                if((vecinos[i][j].cum2)!=0){
                    cargas_red[i][j]+=((double)rotos_grieta[vecinos[i][j].cum2]/
                            (numero_vecinos[vecinos[i][j].cum2]));
                }
                if((vecinos[i][j].cum3)!=0){
                    cargas_red[i][j]+=((double)rotos_grieta[vecinos[i][j].cum3]/
                            (numero_vecinos[vecinos[i][j].cum3]));
                }
                if((vecinos[i][j].cum4)!=0){
                    cargas_red[i][j]+=((double)rotos_grieta[vecinos[i][j].cum4]/
                            (numero_vecinos[vecinos[i][j].cum4]));
                }
            }
        }
    }
}
```

Funciones_clasico_2D.cpp:

```
/**
 * Asocia a todos los elementos de la matriz (argumento) un numero aleatorio dado por
    la distribucion exponencial, es decir, un valor de -log(random entre 0 y 1)
 */
void inicializa_tiempos(double matriz[N][M]){
    for(int i=0; i<N; i++){
        for(int j=0; j<M; j++){
            matriz[i][j]=-log(random_01());
        }
    }
```

```
}

/**
 * Se lee una matriz y se devuelve la fila y columna del elemento de minimo valor
     positivo, asi como este valor
 */
double busca_minimo_matriz(double matriz[N][M], int *fila, int *columna){
    double minimo=matriz[0][0];
    *fila=0;
    *columna=0;
    for(int i=0; i<N; i++){
        for(int j=0; j<M; j++){
            if(minimo<=0){
                minimo=matriz[i][j];  //Buscamos primero el tiempo de un elemento no
                    roto, para comparar
                *fila=i;
                *columna=j;
            }
            if(matriz[i][j]<minimo && matriz[i][j]>0){
                minimo=matriz[i][j];  //Buscamos el minimo tiempo de un elemento no roto
                *fila=i;
                *columna=j;
            }
        }
    }
    return minimo;
}

/**
 * Se actualizan los tiempos transcurridos para cada elemento (argumento 1), a partir
     de las cargas de red (argumento 2) y del intervalo en que las han soportado
     (argumento 3).
 */
void incrementa_tiempos(double tiempos_transcurridos[N][M], double cargas[N][M],
    double delta){
    for(int i=0; i<N; i++){
        for(int j=0; j<M; j++){
            tiempos_transcurridos[i][j]+=delta*pow(cargas[i][j], rho);
        }
    }
}

/**
 * Se devuelven los nuevos tiempos de ruptura para cada elemento (argumento 1),
     calculados a partir de los tiempos iniciales (argumento 2), de los tiempos
     transcurridos (argumento 3) y de las cargas de red (argumento 4).
```

```
 */
void recalcula_tiempos(double tiempos_nuevos[N][M], double tiempos_iniciales[N][M],
                    double tiempos_transcurridos[N][M], double cargas[N][M]){
   for(int i=0; i<N; i++){
       for(int j=0; j<M; j++){
           if(cargas[i][j]>0){
               tiempos_nuevos[i][j]=
  (tiempos_iniciales[i][j]-tiempos_transcurridos[i][j])/(pow(cargas[i][j], rho));
           }else{     //Para elementos rotos, asignamos el valor de 0, que no esta
               contemplado a la hora de buscar el minimo tiempo
               tiempos_nuevos[i][j]=0;
           }
       }
   }
}
```

Funciones_radiactivo_2D.cpp:

```
/**
 * Calcula la probabilidad de ruptura de cada elemento a partir del vector de
 * cargas de red (primer parametro) y la almacena en un vector (segundo parametro).
 * Ademas, devuelve la delta de la configuracion, necesaria para calcular T
 */
double probabilidades_ruptura(double red[N][M], double prob[N][M]){
   double gamma_total=0;    // Tiempo de vida de la configuracion actual
   double gamma[N][M];      // Matriz para el calculo de las probabilidades

   for(int i=0; i<N; i++){ // Calculo de las gammas
       for(int j=0; j<M; j++){
           gamma[i][j]=pow((red[i][j]),rho);
           gamma_total+=gamma[i][j];
       }
   }
   for(int i=0; i<N; i++){ // Calculo de las probabilidades
       for(int j=0; j<M; j++){
           prob[i][j]=(gamma[i][j])/(gamma_total);
       }
   }

   return 1/gamma_total;   // Se devuelve el valor de delta
}

/**
 * A partir del vector de probabilidades, lanza un numero entre 0 y 1 y determina
 * el siguiente elemento en romperse, que develve mediante sus coordenadas (x,y).
```

```
 * La matriz se recorre de izquierda a derecha y de arriba a abajo
 */
void calcula_elemento_roto(double probabilidades[N][M], int *fila, int *columna){
    double numero=random_01();
    double acum_prob=0;
    int j, i;

    /* El intervalo [0,1) se divide en segmentos, cada uno de una longitud
     * igual a la probabilidad de ruptura del elemento. El numero entre 0 y 1
     * determina el segmento en el que cae, al cual se le asocian las coordenadas,
     * que son los parametros de salida
     */
    for(i=0; i<N; i++){
        for(j=0; j<M; j++){
            acum_prob+=probabilidades[i][j];
            if(acum_prob>numero){ // El indice se devuelve cuando el numero esta entre
                dos separaciones de segmentos
                *fila=i;
                *columna=j;
                return;  // Para salir del bucle
            }
        }
    }
}
```

Otras librerías que son comunes a ambos códigos:

Random.cpp:

```
#define NormParisi (2.3283063671E-10F) //Para normalizar el valor generado en la
    rueda de Parisi-Rapuano
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/**
  * Devuelve un numero aleatorio en intervalo [0,1) (distribucion PLANA)
  */
double random_01()
{

    int i;
    unsigned int rueda[256], aleatorio;
    unsigned char indice_ran, indice1, indice2, indice3;
    //Inicializar rueda
    for(i = 0; i < 256; i++)
```

```
        rueda[i] = (rand()<<16) + rand();
    //Inicializar indices
    indice_ran = 0; indice1 = 0; indice2 = 0; indice3 = 0;
    //Modificamos los indices
    indice1 = indice_ran - 24;
    indice2 = indice_ran - 55;
    indice3 = indice_ran - 61;
    //Modificamos la rueda
    rueda[indice_ran] = rueda[indice1] + rueda[indice2];
    //Generamos un numero aleatorio entre 0 y 2^32-1
    aleatorio = (rueda[indice_ran]^rueda[indice3]);
    //Cambiamos la posicion base para el siguiente numero aleatorio
    indice_ran++;
    //Devolvemos el numero aleatorio normalizado, entre 0 y 1
    return aleatorio * NormParisi;
}

double random_ab(double a, double b){
    return a+random_01()*b;
}
```

Estadisticas_ficheros.cpp:

```
/**
 * Calcula un histograma a partir de un conjunto de datos.
 * Primer argumento: ruta a un archivo con los datos en bruto, o bien un vector con
     los datos
 * Number: numero de datos (tamano del vector numeros)
 * Interv: numero de intervalos deseado
 * f_salida: ruta a un archivo para la escritura de resultados
 * Modo: a donde apunta el primer argumento: 0 para archivo de texto, 1 para archivo
     binario, 2 para vector
 */
void histograma(char *f_entrada, int Number, int Interv, char *f_salida, int modo){
    int i;
    int posicion;
    double d, Max, Min;
    double posiciones[Interv];
    double numero_leido;
    FILE *fich_entrada;

/*----Inicializar las frecuencias----*/

    for (i=0; i<Interv; i++){
        posiciones[i]=0;
    }
```

```
/*-------------Calcular el maximo y el minimo-----------*/
   if(modo==1){                 //Para archivos binarios
       fich_entrada=fopen(f_entrada, "rb");
       fread(&numero_leido, sizeof(double), 1, fich_entrada);
       Max=numero_leido;
       Min=numero_leido;

       for (i=1; i<Number; i++){
       fread(&numero_leido, sizeof(double), 1, fich_entrada);
           if (numero_leido>Max){
               Max=numero_leido;
           }
           if (numero_leido<Min){
               Min=numero_leido;
           }
       }
       fclose(fich_entrada);
   }else{
       if(modo==0){             //Para archivos de texto
           fich_entrada=fopen(f_entrada, "r");
           fscanf(fich_entrada, "%lf", &numero_leido);
           Max=numero_leido;
           Min=numero_leido;

           for (i=1; i<Number; i++){
           fscanf(fich_entrada, "%lf", &numero_leido);
               if (numero_leido>Max){
                   Max=numero_leido;
               }
               if (numero_leido<Min){
                   Min=numero_leido;
               }
           }
           fclose(fich_entrada);
       }else{                   //Para vectores
           Max=(*f_entrada);
           Min=(*f_entrada);

           for (i=1; i<Number; i++){
               if (*(f_entrada+i)>Max){
                   Max=(*(f_entrada+i));
               }
               if (*(f_entrada+i)<Min){
                   Min=(*(f_entrada+i));
               }
```

```
            }
        }
    }

/*-------Calcular la separacion entre intervalos------*/

    d=(Max-Min)/Interv;
    if (d==0){
        printf("Error: no se han podido calcular los intervalos.");
        exit(1);
    }

/*------Calcular los valores de la frecuencia en cada intervalo------*/

    if(modo==1){                            //Para archivos binarios
        fich_entrada=fopen(f_entrada, "rb");
        for (i=0; i<Number; i++){
            fread(&numero_leido, sizeof(double), 1, fich_entrada);
            posicion=(numero_leido-Min)/(d);
            posiciones[posicion]++;
        }
        fclose(fich_entrada);
    }else{
        if(modo==0){                        //Para archivos de texto
            fich_entrada=fopen(f_entrada, "r");
            for (i=0; i<Number; i++){
                fscanf(fich_entrada, "%lf", &numero_leido);
                posicion=(numero_leido-Min)/(d);
                posiciones[posicion]++;
            }
            fclose(fich_entrada);
        }else{                              //Para vectores
            for (i=0; i<Number; i++){
                posicion=(*(f_entrada+i)-Min)/(d);
                posiciones[posicion]++;
            }
        }
    }
//Normalizar el histograma

    double Norm;
    Norm=1.0/(Number);
    for(i=0; i<Interv; i++){
        posiciones[i]=posiciones[i]*Norm;
    }
```

```
//Guardar el histograma con el formato adecuado para representar con barras, en el
    fichero indicado por f_salida

    FILE *fich_salida;
    fich_salida=fopen(f_salida, "wt");
    double punto;
    punto=Min+0.5*d;
    for (i=0; i<Interv; i++){
        fprintf(fich_salida, "%lf \t %lf \n", punto, posiciones[i]);
        punto+=d;
    }
    fclose(fich_salida);
}

/**
 * Calcula la media y la varianza de un conjunto de datos
 * Primer argumento: ruta del conjunto de datos
 * Segundo argumento: numero de datos
 * Tercer y cuarto argumentos: media y varianza (resultados de salida)
 * Quinto argumento: tipo de conjunto de datos (0: ruta a archivo de texto, 1: ruta a
     archivo binario, 2: vector)
 */
void med_var(char *f_entrada, int Cantidad, double *media, double *varianza, int
    modo){
    double suma, suma2;
    int j, k;
    suma=0;
    suma2=0;

    if(modo==0){              //Para archivos de texto
        double numero_leido;
        FILE *fich_entrada;

        /*-------------Calculo de la media------------*/
        fich_entrada=fopen(f_entrada, "rt");

        for (j=0; j<Cantidad; j++){
            fscanf(fich_entrada, "%d\n", &numero_leido);
            suma+=numero_leido;
        }
        fclose(fich_entrada);

        /*-----------------Calculo de la varianza--------*/

        fich_entrada=fopen(f_entrada, "rt");
        for (k=0; k<Cantidad; k++){
```

```
            fscanf(fich_entrada, "%d\n", &numero_leido);
            suma2+=(numero_leido)*(numero_leido);
        }
        fclose(fich_entrada);
    }else{
        if(modo==1){          //Para archivos binarios
            double numero_leido;
            FILE *fich_entrada;

            /*-------------Calculo de la media------------*/
            fich_entrada=fopen(f_entrada, "rb");

            for (j=0; j<Cantidad; j++){
                fread(&numero_leido, sizeof(double), 1, fich_entrada);
                suma+=numero_leido;
            }
            fclose(fich_entrada);

            /*-----------------Calculo de la varianza--------*/

            fich_entrada=fopen(f_entrada, "rb");
            for (k=0; k<Cantidad; k++){
                fread(&numero_leido, sizeof(double), 1, fich_entrada);
                suma2+=(numero_leido)*(numero_leido);
            }
            fclose(fich_entrada);
        }else{            //Para vectores
            /*-------------Calculo de la media------------*/

            for (j=0; j<Cantidad; j++){
                suma+=*(f_entrada+j);
            }

            /*-----------------Calculo de la varianza--------*/

            for (k=0; k<Cantidad; k++){
                suma2+=(*(f_entrada+k))*(*(f_entrada+k));
            }
        }
    }
    *media=suma/Cantidad;
    *varianza=suma2/Cantidad-(*media)*(*media);
}
```

Inicializacion_arrays.cpp:

```
/**
 * Asocia a todos los elementos de la matriz (primer argumento) el valor del segundo
    argumento
 */
void inicializa_matriz(int matriz[N][M], int valor){
   for(int i=0; i<N; i++){
      for(int j=0; j<M; j++){
         matriz[i][j]=valor;
      }
   }
}

/**
 * Asocia a todos los elementos de la matriz (primer argumento) el valor del segundo
    argumento
 */
void inicializa_matriz_double(double matriz[N][M], double valor){
   for(int i=0; i<N; i++){
      for(int j=0; j<M; j++){
         matriz[i][j]=valor;
      }
   }
}

/**
 * Asocia a todos los elementos del vector (primer argumento) el valor del segundo
    argumento
 */
void inicializa_vector(int *vector, int valor, int tamano){
   for(int i=0; i<tamano; i++){
      vector[i]=valor;
   }
}

/**
 * Asocia a todos los elementos del vector (primer argumento) el valor del segundo
    argumento
 */
void inicializa_vector_double(double *vector, double valor, int tamano){
   for(int i=0; i<tamano; i++){
      vector[i]=valor;
   }
}
```

2. Apéndice B

Para realizar el reparto de cargas, se utilizan las siguientes variables auxiliares:

- Un vector (de tamaño N) o matriz (de tamaño NxM) cuyos elementos son los índices de las grietas. Los elementos no rotos llevan asignado el valor de 0. En adelante, denotaremos por I a este vector.
- Un vector (de tamaño N) o matriz (de tamaño NxM) cuyos elementos a su vez tienen 2 componentes (4 en el caso bidimensional). Cada una de las componentes de estos elementos constituye el índice de las grietas adyacentes. Por ejemplo, en el caso unidimensional, si un elemento es vecino de las grietas indexadas por 2 y 4, se almacenan los valores de 2 y 4 en la componente correspondiente del vector. Si sólo es vecino de la grieta 3, se almacenan los valores de 3 y 0. Para el caso bidimensional, por ejemplo, si un elemento es vecino de la grieta 1 (por la izquierda y por arriba, es decir, está rodeado) y de la grieta 4 por la derecha, se almacenan los valores de 1, 0, 4 y 0 en la componente correspondiente del vector (en realidad sería 1, 1, 4 y 0, pero los índices que se repiten sólo se almacenan una vez para facilitar el cálculo). Llamaremos a este vector V.
- Un vector, de tamaño N/2+1 en el caso 1D, y de tamaño N·M/2+1 para 2D (en ambos casos se trata del número máximo de grietas distintas posibles, es decir, configuración alternada en 1D y configuración "de ajedrez" en 2D). Este vector almacena el número de elementos rotos de cada grieta (tamaño). En la primera componente de este vector se guarda el número de elementos supervivientes, y en la componente i-ésima del vector, el número de elementos de la grieta indexada por i (la suma de todas las componentes del vector debe dar el número total de elementos del haz). Llamaremos a este vector R, pues está relacionado con las variables tipo r que aparecen en las *Ecs. (1) y (2)*.
- Un vector, del mismo tamaño que el anterior, que guarda en la componente i-ésima el número de elementos sanos que son vecinos de la grieta i-ésima (perímetro). Llamaremos a este vector S, ya que sus componentes son las variables tipo s de las *Ecs. (1) y (2)*.

Y el procedimiento es el siguiente (ver *Figuras 1, 2, 3, 4, 5*):

1. Romper un elemento. Si no es vecino de ninguna grieta (lo cual se consulta mediante el vector I, comprobando que las componentes adyacentes valgan 0), se le asigna una nueva etiqueta (vector o matriz I). En caso contrario, se le asigna la etiqueta correspondiente a una de ellas, con el siguiente orden de prioridad: en 1D, izquierda-derecha; en 2D, arriba-izquierda-derecha-abajo. Por lo tanto, en primer lugar se ha obtenido la componente del vector I que se le asocia al nuevo elemento roto, que denotaremos por I_0.
2. Con esta etiqueta, se incrementa en 1 el valor de la componente correspondiente del vector R, ya que ahora la grieta indexada por I_0 cuenta con un elemento más. Es decir, $R(I_0)$ se incrementa en 1.

3. Si el elemento que se ha roto era vecino de dos o más grietas diferentes, se lee I y se modifican los índices de la grieta o grietas que se han unido, de manera que todos los elementos de la nueva grieta tengan los mismos índices. Esta es la principal diferencia respecto al algoritmo de Hoshen–Kopelman: la lectura de I sólo debe realizarse en caso de que se unan varias grietas, mientras que se omite cuando se forman grietas nuevas. Paralelamente, se modifica apropiadamente el vector R, incrementándose el valor de $R(I_0)$ (en una cantidad igual al número de elementos de las grietas que se han unido), y anulándose las componentes que dan cuenta de la grieta o grietas cuyos índices se han perdido. De esta forma, ya conocemos el valor de las variables tipo r que intervienen en el cálculo de las cargas.

4. Una vez roto el elemento y reindexado correctamente las grietas afectadas, se lee I para encontrar el perímetro, asignando los valores correspondientes al vector S. Por lo tanto, ya contamos con el valor de las variables tipo s, necesarias para calcular las cargas. Paralelamente, se obtienen las componentes de V, lo cual permite efectuar fácilmente ese cálculo, pues para cada elemento i del haz, se conocen los índices de las grietas vecinas, denotados por $V_r(i)$ y $V_l(i)$. En el caso 2D, para cada elemento *(ij)* del haz se tiene $V_l(ij)$, $V_r(ij)$, $V_u(ij)$ y $V_d(ij)$. Para mayor claridad:

$$\begin{cases} \text{1D: } V_r(i) = I(i+1);\ V_l(i) = I(i-1) \\ \text{2D: } V_r(ij) = I(i,j+1);\ V_l(ij) = I(i,j-1);\ V_u(ij) = I(i+1,j);\ V_d(ij) = I(i-1,j) \end{cases}$$

5. Se recorre toda la red, calculando la carga de cada elemento con las ecuaciones vistas anteriormente. En términos del algoritmo, este cálculo se reduce simplemente a:

$$\left.\begin{array}{r} r_r(i) = R[V_r(i)] \\ r_l(i) = R[V_l(i)] \\ s_r(i) = S[V_r(i)] \\ s_l(i) = S[V_l(i)] \end{array}\right\} \Rightarrow \textit{Ec. (1)} \tag{1}$$

   Y en 2D:

$$\left.\begin{array}{r} r_r(ij) = R[V_r(ij)] \\ r_l(ij) = R[V_l(ij)] \\ r_u(ij) = R[V_u(ij)] \\ r_d(ij) = R[V_d(ij)] \\ s_r(ij) = S[V_r(ij)] \\ s_l(ij) = S[V_l(ij)] \\ s_u(ij) = S[V_u(ij)] \\ s_d(ij) = S[V_d(ij)] \end{array}\right\} \Rightarrow \textit{Ec. (2)} \tag{2}$$

Por lo tanto, a diferencia del algoritmo de Hoshen–Kopelman (en el que el vector o matriz I se lee 3 veces por cada iteración), con este método la lectura de I se realiza 2 ó 3 veces, dependiendo de que el nuevo elemento roto unifique o no algunas grietas preexistentes.

Otra ventaja de utilizar estas variables auxiliares es la facilidad de obtener otros resultados,

como la evolución temporal del tamaño medio de las grietas (que se obtiene como el promedio de todas las componentes del vector R), o animaciones de la secuencia de rotura, en las que se visualizan los elementos rotos e intactos (gracias al vector I), así como los elementos del perímetro de las grietas (mediante V).

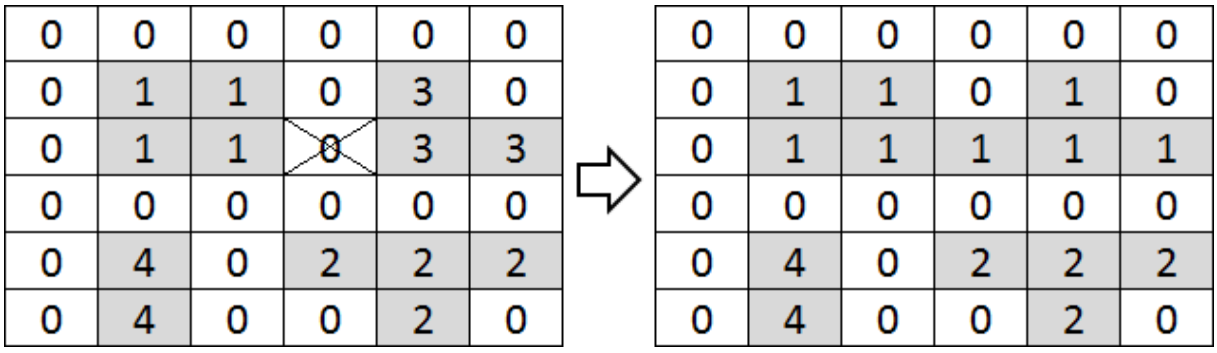

Figura 1: Ejemplo de rotura de un elemento (fila 2, columna 3) y reindexación de las grietas que se unen. Se muestra el cambio en la matriz *I*. Al elemento roto se le asigna el índice de la grieta de su izquierda, por el orden de prioridad establecido, y por tanto la grieta 3 es reindexada con el índice 1. El valor de 0 se asigna a los elementos sanos.

Figura 2: Mismo ejemplo que en la *Figura 1*, pero desde el punto de vista del vector R. La primera componente muestra el número de elementos sanos, y las sucesivas componentes, el número de elementos de cada grieta. Como las grietas 1 y 3 se han unificado en una nueva grieta mayor (indexada por 1), se incrementa el valor de *R(1)* y se anula *R(3)*

.

Figura 3: Mismo ejemplo que en la *Figura 1*, pero desde el punto de vista del vector S. La primera componente muestra el número de elementos sanos (aunque es irrelevante para el cálculo), y las sucesivas componentes, el número de elementos vecinos no rotos de cada grieta (perímetro). Como las grietas 1 y 3 se han unificado en una nueva grieta mayor (indexada por 1), se incrementa el valor de *S(1)* y se anula *S(3)*, aunque en este caso no se conserva la suma de las componentes de S, ya que es más complicado.

0,0,0,0	4,0,0,1	0,0,0,1	0,0,0,0	2,0,0,3	0,0,0,0
0,0,1,0	0,0,0,0	0,0,0,0	0,1,3,0	0,0,0,0	0,3,0,0*
0,3,1,0	0,0,0,0	0,0,0,0	0,1,3,0	0,0,0,0	0,0,0,0
0,0,0,0	1,0,0,4	1,0,0,0	0,0,0,2	3,0,0,2	3,0,0,2
0,2,4,0	0,0,0,0	0,4,2,0	0,0,0,0	0,0,0,0	0,0,0,0
0,0,4,0	0,0,0,0	0,4,0,0	2,0,0,0*	0,0,0,0	2,0,0,0*

⇨

0,0,0,0	4,0,0,1	0,0,0,1	0,0,0,0	2,0,0,1	0,0,0,0
0,0,1,0	0,0,0,0	0,0,0,0	0,1,0,0*	0,0,0,0	0,1,0,0*
0,1,0,0*	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0
0,0,0,0	1,0,0,4	1,0,0,0	1,0,0,2	1,0,0,2	1,0,0,2
0,2,4,0	0,0,0,0	0,4,2,0	0,0,0,0	0,0,0,0	0,0,0,0
0,0,4,0	0,0,0,0	0,4,0,0	2,0,0,0*	0,0,0,0	2,0,0,0*

Figura 4: Mismo ejemplo que en la *Figura 1*, pero desde el punto de vista de la matriz V. En cada elemento, las cuatro componentes indican los índices de las grietas superior, izquierda, derecha e inferior. Los elementos señalados con (*) son aquellos en los que alguno de los índices se repetiría. En estos casos, sólo se tienen en cuenta una vez, ya que el cálculo de σ requiere que las grietas que limitan con el elemento sean distintas.

1	11/6	3/2	1	27/14	1
3/2	0	0	27/14	0	10/7
27/14	0	0	27/14	0	0
1	11/6	3/2	3/2	27/14	27/14
11/6	0	11/6	0	0	0
4/3	0	4/3	3/2	0	3/2

⇨

1	2	5/3	1	13/6	1
5/3	0	0	5/3	0	5/3
5/3	0	0	0	0	0
1	2	5/3	13/6	13/6	13/6
11/6	0	11/6	0	0	0
4/3	0	4/3	3/2	0	3/2

Figura 5: Mismo ejemplo que en la *Figura 1*, pero desde el punto de vista de la matriz de cargas, que se calculan aplicando las *Ecs. (1) ó (2)*, con los valores de R, S y V obtenidos en las *Figuras 2, 3 y 4*.