Problemas de Física con C

1. Dadas las coordenadas X, Y y Z y las cargas Q de un conjunto de cargas puntuales, calcular la energía potencial electrostática del conjunto debida a la interacción entre cargas. Recuérdese que la energía potencial electrostática de un par de cargas, q_1 y q_2, a una distancia d una de otra, vale

$$U = K_e \frac{q_1 q_2}{d}$$

donde $K_e = 9 \times 10^9$ J mC^{-2}, y que la energía total es la suma de todos los pares de cargas.

Se propone, declarar:

```
double epotencial(int ncargas, double q[], double x[], double y[], double
    z[]);
```

Solución:

```
#define Ke 9e+9
double epotencial(int ncargas, double q[], double x[], double y[],
    double z[]) {
     int n, m;
     double d, Ep=0;
     for(n=0; n<ncargas; n++) {
          for(m=0; m<n; m++) {
               d=sqrt((x[m]-x[n])*(x[m]-x[n])
                    +(y[m]-y[n])*(y[m]-y[n])
                    +(z[m]-z[n])*(z[m]-z[n]));
               Ep+=Ke*q[n]*q[m]/d;
          }
     }
     return Ep;
}
```

2. Construir un programa que escriba en un archivo los valores de la posición de una partícula que se mueve con un movimiento uniformemente acelerado en el eje X. Estos valores deberán ser calculados en n instantes del tiempo distribuidos uniformemente en un cierto intervalo $[0, T]$. Los valores de la posición inicial, la velocidad inicial, la aceleración, el número n de tiempos considerados y el intervalo de tiempo T, deberán ser introducidos por línea de comandos al ejecutar la función.

Solución: Ver código `mua.c`

3. Construir un programa que calcule el campo eléctrico creado en un punto del plano XY por un conjunto de cargas. El programa debe pedir al usuario que introduzca por línea de comandos el número de cargas que se van a considerar (hasta un número máximo). Después deberá pedir que introduzca para cada carga los datos de la misma, esto es, su posición definida por las coordenadas (x, y) y el valor de su carga q. Los datos de cada carga serán almacenados en una estructura, y las cargas en un vector

de estructuras. Finalmente, deberá pedir el punto del espacio donde se quiere obtener el campo y calculará dicho campo.

Recordamos que el campo generado por una carga q en un punto del espacio tiene la forma

$$\mathbf{E} = k\frac{q}{r^3}\mathbf{r}$$

siendo $\mathbf{r}$ el vector que va de la carga al punto y r su módulo, y $k = 9 \times 10^9$ N m^2/C^2.

Solución: Ver código `campo_electrico.c`

4. El juego de la vida de Conway consiste en un tablero en cuyas casillas se hallan colocadas "células" que se comportan según las siguientes reglas:

   (a) una célula que tiene menos de dos vecinas en sus 8 celdas cercanas, desaparece (muerte por subpoblación)

   (b) una célula con más de tres vecinas en sus 8 celdas cercanas, desaparece (muerte por sobrepoblación)

   (c) una célula que tiene dos o más vecinas en sus 8 celdas cercanas, permanece como está

   (d) en una casilla vacía cuyas 8 celdas cercanas contienen en total tres células (exactamente), aparece una nueva célula (reproducción)

   Se pide escribir en C una función

```
#define nF 1024
#define nC 1024
int conway(char A[nF][], char B[nF][]);
```

   que aplique estas reglas a una matriz `A[][]` (que contiene en sus celdas `0`, si no hay "célula", o `1` si hay "célula") y guarde el resultado de su evolución (un único paso) en la matriz `B[][]` [ambas matrices tienen los mismos números de filas (`nF`) y de columnas (`nC`)]. Debe tenerse en cuenta que las celdas en los bordes de la matriz (primera y última filas o columnas) no tienen vecinos. La función devolverá el número de celdas modificadas (número de células que han "muerto" o que han "nacido" en el paso).

   Recomendación: Escriba una función

```
int vecinos(char A[nF][], int i, int j);
```

   que devuelva el número de "células vivas" en las celdas vecinas a la (i,j) de la matriz `A`.

   Solución:

```
#define nF 1024
#define nC 1024

int vecinos(char A[nF][], int i, int j) {
    int n=0, ii, jj;
    for(ii=i-1; ii<=i+1; ii++)
        for(jj=j-1; jj<=j+1; jj++)
            if( 0<ii && ii<nF && 0<jj && jj<nC ) n+=A[ii][jj];
    return n-A[i][j];
}

int conway(char A[nF][], char B[nF][]) {
    int nc=0, v;
```

```
    int i, j;
    for(i=0; i<nF; i++) {
        for(j=0; j<nC; j++) {
            v=vecinos(A, i,j);
            switch(v) {
                case 0:
                case 1:
                    B[i][j]=0;
                    break;
                case 2:
                    B[i][j]=A[i][j];
                    break;
                case 3:
                    B[i][j]=1;
                    break;
                default:
                    B[i][j]=0;
            }
            if( B[i][j] != A[i][j] ) nc++;
        }
    }
    return nc;
}
```

5. En su artículo de 1910, Millikan estimó la carga del electrón a partir de medidas indirectas de la carga contenida en una gota de aceite. En la siguiente tabla se encuentran numerados los valores medios de esta carga; cada número k representa lo que Millikan interpretó como "número de electrones" y las cargas q_k están divididas por 10^{-19} Coulombios.

k	4	5	6	7	8	9	10	11
q_k (10^{-19} C)	6.558	8.206	9.880	11.50	13.14	14.81	16.40	18.04
k	12	13	14	15	16	17	18	
q_k (10^{-19} C)	19.68	21.32	22.96	24.60	26.24	27.88	29.52	

Escriba un programa en C que estime la carga del electrón (e) y el error de esta estimación (Δe), relacionados por $q_k = ke + \Delta e$, a partir de estas $N = 15$ medidas, usando las siguientes ecuaciones de regresión lineal:

$$e = \frac{\sum x_k \sum x_k q_k - \sum x_k^2 \sum q_k}{\left(\sum x_k\right)^2 - N \sum x_k^2}$$

$$\Delta e = \frac{\sum x_k \sum q_k - N \sum x_k q_k}{\left(\sum x_k\right)^2 - N \sum x_k^2}$$

Solución: Ver código `Millikan.c`

6. Una partícula subatómica se mueve en línea recta y se encuentra una barrera A. En esa barrera, la partícula tiene cierta probabilidad P_a de ser absorbida, otra probabilidad de continuar su movimiento P_f y una probabilidad P_b de rebotar hacia atrás. Considere ahora N barreras de este tipo, A_1, A_2 ... A_N. Escriba una función que simule por Monte Carlo el comportamiento de una partícula lanzada inicialmente hacia A_1 y que puede ser absorbida, rebotar o traspasar esta barrera y llegar a A_2, donde puede ser absorbida, rebotar de nuevo hacia A_1 o continuar hacia A_3, etc. hasta que o bien la partícula

sea absorbida en alguna barrera, o bien rebote hacia su origen inicial, o bien atraviese todas las barreras: la función devolverá, respectivamente, 0, -1 y $+1$ para indicar el resultado final de la simulación.

Nota: Para tomar la decisión en cada barrera, considere una variable aleatoria uniformemente distribuida entre 0 y 1 (calculada con cierta función `randomf()`) y compárela con las probabilidades acumuladas como

```
double eta=randomf();
if(eta < Pa)     { /* se absorbe */ }
else
if(eta < Pa+Pb) {  /* rebota = invierte su direccion */ }
else               { /* atraviesa la barrera */ }
```

Solución:

```
int mcparticula(int N, float Pa, float Pb, float Pf) {
    int n, d=1;
    while( d!=0 && 0<=n && n<N ) {
        double eta=randomf();
        if( eta < Pa ) {
            d=0; /* es absorbida */
        } else
        if( eta < Pa+Pb ) {
            d=-d; /* rebota = invierte su direccion */
        }
        n+=d; /* siguiente barrera en la direccion del movimiento */
    }
    return d;
}
```

7. El modelo de Ising consiste en un conjunto de N partículas cuánticas ("espines") cuyo número cuántico s puede tomar sólo los valores $+\frac{1}{2}$ o $-\frac{1}{2}$. Los espines interactúan entre sí, de forma que la energía del sistema aumenta cuando un par de espines en interacción son paralelos y disminuye cuando son antiparalelos; usualmente se dice que cada par de espines s_i y s_j suma a la energía del sistema un valor $s_i s_j$. Los pares de espines en interacción pueden ser todos los pares que se pueden formar (en total, $N(N-1)/2$ pares), o sólo los pares consecutivos en la línea de espines (N pares).

   Escríbase una función

   float ising_U(**int** N, **float** s[], **int** local);

   que, al pasarle el número de espines, el array con los valores de los espines y un valor que indique si la interacción es global (0: todos los pares posibles) o local (1: sólo los vecinos próximos) calcule la energía del modelo de Ising.

   Solución:

```
float ising_U(int N, float s[], int local) {
    int i, j;
    float egy=0.0;
    if(local) {
        for(i=1; i < N; i++)
            egy+=s[i]*s[i-1];
        egy+=s[0]*s[N-1];
    } else {
        for(i=0; i < N; i++)
```

```
            for(j=0; j < i; j++)
                egy+=s[i]*s[j];
        }
        return egy;
    }
```

8. El objetivo de este ejercicio es diseñar un programa que calcule el centro de masas de un sistema discreto de partículas. Supondremos que los datos de las partículas se encuentran en un archivo con cuatro columnas: las tres primeras columnas corresponden a las coordenadas (x_i, y_i, z_i) de cada partícula i, mientras que la cuarta columna corresponde a la masa m_i. Aunque no conocemos exactamente el número de partículas que componen el sistema, sabemos que es menor que 1000. El nombre del archivo que contiene los datos deberá ser introducido por línea de comandos (como argumento de la función `main()`) cuando se ejecute el programa. Para leer y almacenar los datos de las partículas, se deberá definir una estructura denominada "`Punto`" que representa una partícula con su posición y masa. Después, en la función `main()` de deberá declarar un array de "`Puntos`", es decir, un array de estas estructuras, para almacenar en él la información de todo el sistema de partículas. El programa debe escribir en pantalla la posición del centro de masas del sistema.

   La posición del centro de masas de un conjunto de partículas con posición $\mathbf{r}_i$ y masa m_i es:

$$\mathbf{r}_{CM} = \frac{\sum_{i=1}^{n} m_i \mathbf{r}_i}{\sum_{i=1}^{n} m_i}$$

   Solución: Ver código `centro_de_masas.c`

9. Supongamos una red cuadrada discreta en la que cada nodo de la red (i, j) con $i = 0, 1, 2, \ldots$ y $j = 0, 1, 2, \ldots$ ha sido definido mediante la estructura:

```
struct punto {
       int i,j;
};
typedef struct punto Punto;
```

   Consideremos ahora que nuestro programa principal calcula dos trayectorias sobre la red descritas mediante sendos vectores de `Puntos` que han sido declarados dentro de `main` como:

```
Punto camino1[long1];
Punto camino2[long2];
```

   Escriba una función que reciba de `main` estas dos trayectorias con sus respectivas longitudes, y que devuelva el número de puntos de la red que tienen en común.

   Solución:

```
int compara_caminos (int long_a, Punto *a, int long_b, Punto *b) {
    int k, l, cont;
    cont=0;
    for (k=0; k<long_a; k++) {
        for (l=0; l<long_b; l++) {
            if (a[k].i==b[l].i && a[k].j==b[l].j) {
                cont++; break;
            }
        }
    }
```

```
        return cont;
    }
```

10. En este ejercicio vamos a simular de forma muy simplicada el movimiento browniano de una partícula en $2D$. Para ello contamos con dos funciones definidas del siguiente modo:

```
double calcula_angulo() {
    double Pi=3.1416;
    return ((double)2*Pi*rand()/(RAND_MAX+1));
}

double calcula_distancia() {
    return(1-(double)rand()/(RAND_MAX+1));
}
```

La primera función devuelve un ángulo aleatorio θ distribuido uniformemente en el intervalo $[0, 2\pi)$. La segunda devuelve una distancia aleatoria d distribuida uniformemente en el intervalo $(0, 1]$. En nuestro modelo suponemos que la partícula va experimentando choques aleatorios con el medio, de modo que después de cada choque la partícula sale con una dirección aleatoria determinada por el ángulo θ que forma el vector con el eje X, y se mueve en línea recta recorriendo una distancia aleatoria d hasta el siguiente choque. Por lo tanto, si llamamos $\mathbf{x}_i$ al vector posición de la partícula en el choque i, después del cual sale con dirección θ_i recorriendo una distancia lineal d_i, el siguiente choque se producirá en el punto:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{d}_i$$

donde el vector $\mathbf{d}_i$ tiene la forma:

$$\mathbf{d}_i = d_i \begin{pmatrix} \cos\theta_i \\ \sin\theta_i \end{pmatrix}$$

Escribir un programa en C que simule este movimiento browniano y que imprima en un archivo de datos los puntos en los que la partícula ha experimentado un choque con el medio (si después representamos gráficamente estos puntos uniéndolos con segmentos obtendremos la trayectoria de la partícula). La posición de la partícula $\mathbf{x} = (x, y)$ deberá ser definida como una estructura, y $\mathbf{d}$ como un vector. El archivo de datos deberá tener dos columnas: la primera con la coordenada x y la segunda con la coordenada y de los sucesivos puntos. La posición inicial de la partícula $\mathbf{x}_0 = (x_0, y_0)$, el número de pasos que queremos simular `npasos`, y el nombre del archivo al que se exportarán los resultados `datos.dat` deberán ser introducidos por línea de comandos como argumentos de la función `main`.

Solución: Ver código `movimiento_browniano_2D.c`

11. El objetivo de este ejercicio es simular el "paseo aleatorio" de una partícula que se mueve por un medio bidimensional experimentando choques aleatorios con las partículas del mismo. Después de cada choque la partícula sale con una dirección aleatoria determinada por el ángulo polar ϕ que forma el vector director con respecto al eje X, y se mueve en línea recta recorriendo una distancia s, que también es aleatoria, hasta el siguiente choque. Por lo tanto, si llamamos $\mathbf{r}_j$ al vector posición de la partícula en el choque j, después del cual sale con dirección ϕ_j recorriendo una distancia lineal s_j, el siguiente choque se producirá en el punto $\mathbf{r}_{j+1} = \mathbf{r}_j + \mathbf{s}_j$, donde el vector $\mathbf{s}_j$ tiene la forma $\mathbf{s}_j = s_j\,(\cos\phi_j, \sin\phi_j)$.

En este ejercicio deberá escribir un programa en C que simule este paseo aleatorio y que imprima en un archivo de datos los puntos en los que la partícula ha experimentado un choque con el medio. La posición de la partícula $\mathbf{r} = (x, y)$ deberá ser definida como un vector, mientras que el vector $\mathbf{s}_i$ deberá ser definido como una estructura. El archivo de datos deberá tener dos columnas: la primera con la coordenada x y la segunda con la coordenada y de los sucesivos puntos. La posición inicial de la partícula $\mathbf{r}_0 = (x_0, y_0)$, el número de pasos que queremos simular `npasos`, y el nombre del archivo

al que se exportarán los resultados `datos.dat` deberán ser introducidos por línea de comandos como argumentos de la función `main`.

Para ello contamos con dos funciones definidas del siguiente modo, cuyos cálculos internos son irrelevantes para el ejercicio. Cada vez que son llamadas estas funciones retornan valores aleatorios.

```
double calcula_phi() {
    double phi;
    ...  /* la funcion calcula el angulo aleatorio phi con el que sale la
        particula
          despues de cada choque */
    return(phi);
}

double calcula_s() {
    double s;
    ...  /* la funcion calcula la distancia aleatoria s que recorre la
        particula
          entre dos choques consecutivos */
    return(s);
}
```

Solución: Ver código `paseo_aleatorio_2D.c`

12. La simulación de Monte Carlo de la interacción radiación-materia consiste básicamente en simular mediante colisiones los diferentes tipos de interacción que pueden experimentan las partículas que componen la radiación (fotones, electrones, partículas α, neutrones, etc.) con las partículas del medio (electrones libres o ligados y núcleos atómicos). Supongamos un haz de luz muy energética (rayos X o radiación gamma) que atraviesa un medio. A medida que los fotones penetran en el material experimentan colisiones que modifican la dirección del movimiento y que reducen su energía. Esta energía perdida por el fotón se deposita en el punto del material donde se ha producido la colisión. Entre dos colisiones consecutivas el fotón se moverá en línea recta.

    En este ejercicio vamos a simular de forma muy simplicada el movimiento de un fotón en un medio 3D y calcularemos los puntos en los que va depositando su energía. Supondremos que nuestro fotón parte del origen de coordenadas con una energía inicial E_0. Después de recorrer un desplazamiento aleatorio $\mathbf{d}_1$ se producirá la primera interacción con el medio (colisión 1), en la que cederá una parte de su energía ΔE_1, que es una variable aleatoria. Después de esa colisión el fotón saldrá con una energía $E = E_0 - \Delta E_1$ y realizará de nuevo un desplazamiento aleatorio $\mathbf{d}_2$ hasta la segunda colisión, donde vuelve a ceder una energía ΔE_2, saliendo de ella con energía $E = E_0 - \Delta E_1 - \Delta E_2$. Este proceso se repite hasta que la energía del fotón desciende por debajo de una cierta energía denominada energía de absorción del medio, E_{abs}. Cuando esto ocurre el fotón es absorbido en ese mismo punto cediendo al medio el resto de su energía y finaliza la simulación.

    Para simular este proceso disponemos de la siguiente función, que calcula el vector desplazamiento (aleatorio) entre dos interacciones consecutivas, y cuyos cálculos internos son irrelevantes para el ejercicio:

```
void desplazamiento_aleatorio(double d[]) {
    ... /* la funcion calcula el vector desplazamiento
        entre dos interacciones consecutivas */
    d[1]= ...;
    d[2]= ...;
    d[3]= ...;
```

```
    return;
}
```

Y también de la siguiente función que calcula la energía perdida en cada interacción (que también es una variable aleatoria), y que depende de la energía del fotón justo antes de la interacción.

```
double energia_perdida(double E) {
    double Delta_E;
    .... /* la funcion calcula la energia perdida en la interaccion (
        Delta_E)
            en funcion de la energia del foton antes de la interaccion (E)
                */
    return(Delta_E);
}
```

Cada vez que son llamadas estas funciones retornan valores aleatorios.

Escribir un programa en C que simule la vida del fotón en el medio y el proceso de deposición de su energía, y que imprima en un archivo de datos los puntos del medio en los que se han producido las interacciones junto con la energía depositada en cada punto. El archivo de datos deberá tener cuatro columnas: la tres primeras columnas deben ser paras las coordenadas x, y, z de los puntos donde se han producido las interacciones, mientras que la cuarta columna debe mostrar la energía perdida por el fotón y por tanto depositada en ese punto. La energía inicial del fotón E_0, la energía de absorción del medio E_{abs}, y el nombre del archivo al que se exportarán los resultados `datos.dat` deberán ser introducidos por línea de comandos como argumentos de la función `main`.

Solución: Ver código `simulacion_MC_fotones.c`

13. Cuando una partícula de radiación ionizante como una partícula α (núcleo de $^4_2$He) penetra en un medio material, experimenta colisiones con las partículas del medio que modifican la dirección de su movimiento y reducen su energía. Entre dos colisiones consecutivas la partícula se moverá en línea recta y la energía perdida por la partícula en cada colisión se deposita en el punto del medio donde se ha producido la colisión.

    El objetivo de este ejercicio es simular el "viaje" que experimenta una partícula α por un medio material tridimensional hasta que pierde toda su energía y es absorbida por el medio. Para simular este proceso disponemos de la siguientes funciones.

    La función `desplazamiento_entre_colisiones` calcula el vector desplazamiento (aleatorio) entre dos colisiones consecutivas. Por lo tanto, si llamamos $\mathbf{r}_i$ al vector posición de la partícula en el choque i, el siguiente choque se producirá en el punto:

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{d}_i$$

    La función `desplazamiento_entre_colisiones` está definida de esta forma (los cálculos internos son irrelevantes para el ejercicio):

```
void desplazamiento_entre_colisiones(double *d) {
    ... /* la funcion calcula el vector desplazamiento aleatorio
        entre dos interacciones consecutivas */
    d[1]= ...;
    d[2]= ...;
    d[3]= ...;
    return;
}
```

Por otro lado tenemos la función `energia_depositada`, que devuelve la energía perdida por la partícula α en cada colisión y que se deposita en el medio en el punto en el que se ha producido la colisión. Esta energía cedida es una variable aleatoria que depende de la energía de la partícula justo antes de la interacción.

```
double energia_depositada(double E) {
    double E_perdida;
    .... /* la funcion calcula la energia perdida en la interaccion (E_perdida)
            en funcion de la energia de la particula antes de la interaccion (E) */
    return(E_perdida);
}
```

Cada vez que son llamadas estas funciones retornan valores aleatorios.

Escribir un programa en C que simule el transporte de una partícula α en un medio material. La partícula comenzará su viaje en el punto inicial $\mathbf{r_0} = (x_0, y_0, z_0)$ con una energía inicial E_0. La partícula experimentrá un desplazamiento aleatorio obtenido de la función `desplazamiento_entre_colisiones` hasta la primera colisión, donde cederá una parte de su energía obtenida de la función `energia_depositada`. A continuación realizará un nuevo desplazamiento hasta la segunda colisión, y así sucesivamente hasta que su energía descienda por debajo de una cierta energía denominada energía de absorción del medio, E_{abs}. Cuando esto ocurre consideraremos que la partícula ha sido absorbida en ese mismo punto, cediendo al medio el resto de su energía, y finalizará la simulación. El programa debe imprimir en un archivo de datos los puntos del medio en los que se han producido las interacciones junto con la energía depositada en cada punto. El archivo de datos deberá tener cuatro columnas: la tres primeras columnas deben ser paras las coordenadas x, y, z de los puntos donde se han producido las interacciones, mientras que la cuarta columna debe mostrar la energía perdida por la partícula y por tanto depositada en ese punto. La posición inicial de la partícula $\mathbf{r_0} = (x_0, y_0, z_0)$, su energía inicial E_0, la energía de absorción del medio E_{abs}, y el nombre del archivo al que se exportarán los resultados `datos.dat` deberán ser introducidos por línea de comandos como argumentos de la función `main`.

Solución: Ver código `simulacion_MC_particulas_alpha.c`

14. La teoría cinética de los gases es un modelo simple del comportamiento de un gas diluido. Según esta teoría un gas está constituido por un gran número de partículas puntuales (átomos o moléculas) que se desplazan rápidamente experimentando colisiones elásticas entre sí y con las paredes del recipiente. Estas colisiones cambian su dirección de movimiento y su velocidad. El resultado es que las moléculas del gas tienen distintas velocidades, y estas velocidades están distribuidas según la conocida función de distribución de velocidades de Maxwell-Boltzmann.

    El objetivo de este ejercicio es simular de forma muy simplicada el movimiento de una de esas moléculas del gas en tres dimensiones. Para ello **sólo** nos fijaremos en los puntos y en los instantes de tiempo en los que la partícula experimenta una colisión. Supongamos que en el instante de tiempo t la molécula experimenta una colisión en el punto $\mathbf{r}(t)$ (recordamos que estamos trabajando con tres dimensiones). Después de esa colisión la partícula sale con una velocidad aleatoria $\mathbf{v}$ y el tiempo que transcurre hasta la siguiente colisión es Δt, que también es una variable aleatoria. De este modo la siguiente colisión tendrá lugar en el tiempo $t + \Delta t$ y ocurrirá en el punto

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \Delta t\, \mathbf{v}$$

    Escribir un programa en C que simule el movimiento de la molécula del gas y que imprima en un archivo de datos los siguientes datos para cada colisión. El archivo de datos deberá tener seis columnas: en la primera columna deben ir los tiempos (t) en los que se producen las colisiones, las tres siguientes son para las coordenadas x, y y z de los puntos $\mathbf{r}(t)$ en los que se han producido las colisiones, la quinta columna deberá mostrar el desplazamiento de la molécula con respecto a la posición inicial $\|\mathbf{r}(t) - \mathbf{r}_0\|$ y la última columna el módulo del incremento de momento que ha sufrido la partícula en el choque

$\|\Delta\mathbf{p}\| = m\|\Delta\mathbf{v}\|$. La masa m de la molécula, la posición inicial $\mathbf{r}_0 = (x_0, y_0, z_0)$ donde se produce la primera colisión, el número de colisiones que queremos simular `n_colisiones`, y el nombre del archivo al que se exportarán los resultados `datos.dat` deberán ser introducidos por línea de comandos como argumentos de la función `main`.

Para este programa supondremos que contamos con dos funciones definidas del siguiente modo (los cálculos internos son irrelevantes para el ejercicio):

```
void distribucion_Maxwell_Boltzmann(double v[]){
    ... /* la funcion calcula las componentes del nuevo vector velocidad
        despues de la colision */
    v[1]= ...;
    v[2]= ...;
    v[3]=....;
    return;
}

double tiempo() {
    double t;
    .... // la funcion devuelve el tiempo t que transcurre hasta la
        siguiente colision
    return t;
}
```

La primera función calcula el vector velocidad de la molécula de gas después de una colisión, de acuerdo son la distribución de Maxwell-Boltzmann. La segunda función devuelve el intervalo de tiempo aleatorio Δt que transcurre hasta la siguiente colisión. Cada vez que son llamadas estas funciones retornan valores aleatorios.

Solución: Ver código `teoria_cinetica_gases.c`

15. Consideremos un perfil de alturas h_i (con $i = 0, \ldots, L-1$) dado por una secuencia de L valores enteros que indican la altura h en el punto i. Modelamos la caída de una partícula sobre este perfil en un punto aleatorio p haciendo que la altura del perfil en ese punto, h_p, se incremente en una unidad. Repitiendo esto para un número N de partículas depositadas, se obtiene un perfil de alturas aleatorias. El objetivo de este ejercicio es escribir una función externa a `main` que reciba la longitud L del perfil, un array de enteros de esa longitud (h_i) y un valor N con el número de partículas a depositar. La función deberá inicializar la altura de todos los puntos a 0 y después actualizar los valores de h_i simulando el depósito de esas N partículas en posiciones aleatorias del perfil. Utilícese, para calcular estas posiciones, la siguiente instrucción:

```
p=(int)(L*rand()/(double)(RAND_MAX+1));
```

Esta instrucción devuelve un número entero aleatorio distribuido uniformemente en el intervalo `[0,L-1]`.

Solución:

```
void deposicion_aleatoria(int L, int h[], int N) {
    int i, p;
    srand(time(NULL)); /* esto es opcional, de este modo la semilla del
    generador de numeros aleatorios esta referida al reloj de la CPU y
    por tanto cambia de forma continua, nunca es la misma */

    for(i=0; i<L; i++) h[i]=0;
    for(i=1; i<=N; i++) {
```

```
            p=(int)(L*rand()/(double)(RAND_MAX+1));
            h[p]++;
        }
        return;
    }
```

16. Consideremos un perfil de alturas h_i (con $i = 0, \ldots, L-1$) dado por una secuencia de L valores enteros que indican la altura h en el punto i. Escribir una función externa a `main` que reciba la longitud L del perfil y el array de enteros h_i de dicho perfil, y calcule la rugosidad w del perfil, definida como la desviación típica de los valores en h_i, esto es:

$$w = \sqrt{\frac{1}{L}\sum_{i=0}^{L-1}(h_i - \bar{h})^2}$$

donde

$$\bar{h} = \frac{1}{L}\sum_{i=0}^{L-1} h_i$$

Solución:

```
double calcula_rugosidad(int L, int h[]) {
    int i;
    double hmedia, w;
    hmedia=w=0.;
    for(i=0; i<L; i++) hmedia+=h[i];
    hmedia=hmedia/L;
    for(i=0; i<L; i++) w+=pow(h[i]-hmedia,2);
    w=sqrt(w/L);
    return w;
}
```

17. Para un estudio determinado estamos trabajando con datos sobre planetas. Para ello definimos una estructura llamada `Planeta` del siguiente modo:

```
struct Planeta {
    char *nombre;
    double radio;
    double masa;
    double volumen;
    double densidad;
};
```

Todos los campos están completados mediante la recogida de datos excepto `volumen` y `densidad`, que están inicializados con un valor nulo. Escriba una función tipo `void` llamada `actualizar_planeta` que reciba como argumento una estructura `Planeta` pasada por referencia, y actualice los campos `volumen` y `densidad` de ese planeta calculándolos a partir de su radio y de la masa.

Nota: Suponga que el valor de π está definido como una variable global con el nombre M_PI.

Solución:

```
void actualizar_planeta(struct Planeta *P1) {
    double volumen, densidad;
    volumen=4*M_PI*pow(P1->radio, 3)/3;
```

```
        densidad=(P1->masa)/volumen;
        P1->volumen=volumen;
        P1->densidad=densidad;
        return;
    }
```

18. Suponga un recinto circular de radio unidad centrado en el origen de coordenadas y una partícula que inicialmente se encuentra en el centro de dicho recinto. La partícula se mueve sobre el plano describiendo una cierta trayectoria y muestreamos su posición en intervalos de tiempo de $\delta t = 1$ segundo, guardando la posición (x, y) en la siguiente estructura denominada `Vector2d`:

```
struct Vector2d {
    double x, y;
};
```

La diferencia entre la posición a tiempo t y la posición en $t+\delta t$, $v = (x(t+\delta t) - x(t), y(t+\delta t) - y(t))$, está dada por el `Vector2d v` resultante de la función:

```
struct Vector2d incremento_posicion() {
    struct Vector2d v;
    ...
    return v;
}
```

Los puntos suspensivos representan los cálculos necesarios para obtener `v`. Esos cálculos no nos interesan en este problema (no es necesario completar la función) y asumiremos que `v` se obtiene de una forma que se conoce.

Supongamos que el movimiento de la partícula finaliza cuando dicha partícula consigue salir del círculo. Escriba un programa que simule el movimiento de la partícula desde $t = 0$ hasta que abandona el círculo. El programa deberá repetir esto un número `N` de veces, simulando así `N` partículas diferentes (todas partiendo del centro de círculo) y deberá guardar en un archivo de texto el tiempo que tarda cada partícula en abandonar el círculo. Tanto `N` como el nombre del archivo de datos deberán ser pasados por línea de comandos como argumentos de la función `main`.

Solución: Ver código `escape_circulo.c`

19. Para un determinado estudio estamos trabajando con datos sobre estrellas. Para ello definimos una estructura llamada `Estrella` del siguiente modo:

```
struct Estrella{
    char name[10];
    char tipo[10];
    int brillo;
    double masa;
};
```

Debido al interés sobre las estrellas dobles, definimos una nueva estructura denominada `EstrellaDoble` del siguiente modo:

```
struct EstrellaDoble{
        struct Estrella E1;
        struct Estrella E2;
        double masa;
};
```

Escriba una función denominada `estrella_doble()` que reciba como argumentos de entrada dos estructuras `Estrella`, y retorne como salida una estructura `EstrellaDoble`. La estructura retornada por la función debe tener como primeras dos componentes las estrellas que se pasan como argumentos de entrada, y como campo masa, la suma de las masas individuales de dichas estrellas.

Solución:

```
struct EstrellaDoble estrella_doble(struct Estrella E1, struct Estrella E2){
    struct EstrellaDoble Edoble;
    Edoble.E1=E1;
    Edoble.E2=E2;
    Edoble.masa=E1.masa+E2.masa;
    return Edoble;
}
```

20. Supongamos un sistema formado por una cadena de longitud L en la que para cada posición i (con $i = 1, \ldots, L$) se ha definido una cierta variable S_i. Supongamos ahora que para el sitio i calculamos un valor de "energía" dado por $A_i = S_{i-1} \cdot S_i \cdot S_{i+1}$, representando la interacción del sitio S_i con sus vecinos. En este tipo de sistemas lo usual es imponer condiciones de contorno periódicas, haciendo que el vecino izquierdo del primer elemento sea el último, y que el vecino derecho del último elemento sea el primero, es decir, $A_1 = S_L \cdot S_1 \cdot S_2$ y $A_L = S_{L-1} \cdot S_L \cdot S_1$. Escriba una función que, dado el elemento i del sistema, devuelva el valor A_i calculado a partir de los S_i. Los argumentos de entrada de la función deberán ser el array `S[L]` que describe el sistema y la posición i (con $i = 1, \ldots, L$) sobre la cual calcularemos el valor de A_i. Para escribir la función, puede suponer que el tamaño del sistema L está definido como una constante global.

Nota: Usar la instrucción condicional **switch**(){ **case**: **default**: } a la hora de imponer las condiciones periódicas.

Solución:

```
double a_comp(double A[],int i){
    double Ai;
    switch(i){
        case 1:
        Ai=A[L-1]*A[0]*A[1];
        break;
        case L:
        Ai=A[0]*A[L-1]*A[L-2];
        break;
        default:
        Ai=A[i-2]*A[i-1]*A[i];
    }
    return Ai;
}
```

21. Escriba la implementación de las funciones

```
float resSeries(int N, float R[]);
float resParall(int N, float R[]);
```

que devuelven los valores de resistencia total de configuraciones de N resistencias (con valores individuales dados por los R_n) en serie o en paralelo, respectivamente.

Solución:

```
float resSeries(int N, float R[]) {
    int n;
    float Rt=0;
    for(n=0; n<N; n++) {
        Rr+=R[n];
    }
    return R;
}

float resParall(int N, float R[]) {
    int n;
    float iR=0;
    for(n=0; n<N; n++) {
        iR+=1/R[n];
    }
    return 1/iR;
}
```

22. Explique brevemente el funcionamiento de la función

```
float axon(int x[], int N, int n[], float w[]) {
        float y=0;
        int k;
        for(k=0; k < N; k++)
                y+=w[k]*X[n[k]];
        return sigma(y);
}
```

¿Qué valores posibles pueden albergar los elementos del vector `n[]`? ¿Cuál debería ser la declaración de la función `sigma()` para que funcione todo correctamente?

Solución: La función recibe los vectores $\boldsymbol{x}$ (entero), $\boldsymbol{n}$ (entero) y $\boldsymbol{w}$ (flotante de precisión simple) y el valor entero N. En su interior calcula la sumatoria

$$\sum_{k=0}^{N-1} w_k x_{n_k}$$

Se deduce que N representa la dimensión de los vectores $\boldsymbol{n}$ y $\boldsymbol{w}$, mientras que la del vector $\boldsymbol{x}$ tendrá que ser suficiente para contener al mayor de los n_k. Por último, la función devuelve el resultado de llamar a la función `sigma()` con el valor de la sumatoria como argumento. Esta función conviene declararla como sigue:

```
float sigma(float y);
```

La función `axon()` implementa el modelo matemático de una neurona: $\boldsymbol{x}$ son las señales que llegan a sus dendritas desde las neuronas conectadas con ella, los $\boldsymbol{n}$ representan los "nombres" (numéricos) de estas N neuronas vecinas; los $\boldsymbol{w}$ representan la "fuerza" de las conexiones. La señal emitida por el axón de la neurona, que es lo que devuelve la función, se calcula como una función de respuesta (σ) aplicada a la suma de las $\boldsymbol{x}$ ponderada con los pesos $\boldsymbol{w}$.

23. Suponga que partimos de una matriz de $L \times L$ enteros. Cada celda de la matriz representa una posición sobre una superficie cuadrada discreta y el valor de la celda indica la altura de la superficie en esa posición. Incialmente la supericie es plana, por lo que todos los elementos de la matriz son iguales a

cero. A continuación se depositan sobre esa superficie N partículas de forma sucesiva. Las partículas van cayendo sobre posiciones aleatorias, y después de cada deposición la altura en esa posición se incrementa en una unidad. **Escriba una función** que reciba el tamaño L de la matriz de alturas, la propia matriz H y el número N de partículas que se van a depositar. La función debe simular la deposición sobre la matriz de las N partículas y devolver la altura media h (un número real) de las celdas de la matriz.

Nota: Use la función rand() para generar las posiciones aleatorias. Para generar un número entero entre 0 y $L-1$, puede usar la aproximación del módulo:

```
rand() % L
```

Solución:

```
double deposita(int L, int M[][L], int N) {
  int n;
  int i, j;
  double h=0.0;
  for(n=0; n < N; n++) {
    i=random() % L;
    j=random() % L;
    M[i][j] ++;
  }
  for(i=0; i < L; i++) {
    for(j=0; j < L; j++) {
      h=h+M[i][j];
    }
  }
  h = h/(L*L);
  return h;
}
```

24. Las longitudes de onda λ_r de las transiciones atómicas del átomo de hidrógeno se pueden calcular usando la fórmula de Rydberg:

$$\frac{1}{\lambda_r} = R_\infty \left(\frac{1}{n_1^2} - \frac{1}{n_2^2} \right)$$

donde $R_\infty = 1.097 \times 10^{-2}$ nm^{-1}. Para cada valor de n_1 se obtiene una serie ($n_1 = 1$ da la serie de Lyman, $n_1 = 2$ la serie de Balmer, $n_1 = 3$ la serie de Paschen, etc.). **Escriba un programa** que calcule las 5 primeras longitudes de onda (en nanómetros) de cada una de las 4 primeras series.

Solución:

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

#define Rinf 1.097e-2

int main(int argc, char** argv) {
  int n1, n2;
  double lr;
  printf("n1\tn2\tlambda (nm)\n");
  for(n1=1; n1<=4; n1++) {
    for(n2=1; n2<n1; n2++) {
```

```
            lr= 1.0/( Rinf*( 1.0/(n1*n1) - 1.0/(n2*n2) ) );
            printf("%d\t%d\t%g\n", n1, n2, lr);
        }
    }
    return 0;
}
```

25. La ley de las lentes delgadas

$$\frac{1}{f} = \frac{1}{x_o} + \frac{1}{x_i}$$

relaciona la distancia x_o de un objeto a la lente, la distancia x_i a la que se forma la imagen y la distancia focal f de la lente. **Escriba un programa** al que se pase por línea de comandos la distancia focal f, y que pida sucesivamente al usuario (con scanf()) un valor de x_o y calcule y muestre por pantalla el correspondiente x_i. El programa terminará cuando el usuario introduzca un valor de 0.

Solución:

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

int main(int argc, char** argv) {
  int n;
  double f, xo, xi;
  f=atof(argv[1]);
  do {
    printf("xo = ");
    n=scanf("%lf", &xo);
    if( n && xo!=0 ) {
      xi = 1.0 / ( 1.0/f - 1.0/xo );
      printf("xi = %g\n", xi);
    }
  } while(n && xo!=0);
  return 0;
}
```

26. Una simulación de física estadística consiste en seguir los cambios en el estado discreto de un sistema; esos estados discretos se describen mediante un "número cuántico" $n = 1, 2, 3, 4, \ldots N$. Las transiciones que se producen entre esos estados en cada paso de tiempo, vienen descritas por la matriz A, en cuya celda a_{ij} se guarda la probabilidad de que haya una transición desde el estado i-ésimo al estado j-ésimo. **Escriba una función** que reciba el estado del sistema (un número entero n entre 1 y N) y la matriz A (de $N \times N$) y devuelva un número entero m correspondiente al estado al que saltará el sistema en este paso de simulación. Para ello, genere un número aleatorio r uniformemente distribuido entre [0, 1) y, teniendo en cuenta que la suma de todos los elementos de una fila de A vale 1, determine el estado de destino como el m más pequeño tal que $\sum_{j=0}^{m} a_{nj} > r$.

Nota 1: Use la función rand() para generar los números aleatorios. Recuerde que, para generar un número aleatorio en el intervalo [0, 1), puede usar: **double** r=rand()/((**double**)RAND_MAX+1);

Nota 2: Suponga que el valor de N está dado en el programa; o bien es una variable global entera, o bien se ha #definido, p.ej. #**define** N 100

Solución:

```
int transicion(double A[][N], int n) {
  int m;
  double r, s;
  r=random()/((double)RAND_MAX+1);
  s=0.0;
  for(m=0; m < N && s<r; m++) {
    s=s+A[n-1][m];
  }
  return m+1;
}
```

27. Supongamos que tenemos un agregado bidimimensional compuesto de N partículas de la misma masa. La posición de la partícula i (con $i = 1, \ldots, N$) está determinada por el vector $\mathbf{r}_i$. La distancia media entre dos partículas del clúster, ξ_N, se calcula como:

$$\xi_N = \left(\sum_{i,j=1}^{N} \frac{|\mathbf{r}_i - \mathbf{r}_j|^2}{N^2} \right)^{1/2}$$

donde $i = 1, \ldots, N$ y $j = 1, \ldots, N$. Vemos que la suma incluye todas las parejas que se pueden formar entre las partículas del clúster, incluyendoe las parejas formadas por cada partícula consigo misma.

Supongamos que en nuestro programa hemos definido la posición de una partícula como un nuevo tipo de variable, denominada `particula`, del siguiente modo:

```
struct coordenadas_posicion{
    double x, y;
};
typedef struct coordenadas_posicion particula;
```

El objetivo de este ejercicio es escribir en C una función que calcule y devuelva la distancia media entre dos partículas de un clúster. El número de partículas del clúster, N, y las posiciones de las partículas dadas en forma de vector de elementos de tipo `particula`, deben ser pasados como argumentos de la función.

Solución:

```
double calcula_distancia(int N, particula C[]) {
    int i,j;
    double R=0;
    for(i=0; i<N; i++) {
        for(j=0; j<N; j++) {
            R+=pow(C[i].x-C[j].x,2)+pow(C[i].y-C[j].y,2);
        }
    }
    return sqrt(R/(N*N));
}
```

28. Un polímero es un agregado de partículas (moléculas, macromoléculas, etc.) que puede adoptar formas muy complicadas. Supongamos un polímero en 2 dimensiones compuesto de N partículas de la misma masa. La posición de la partícula i en el polímero (con $i = 1, \ldots, N$) está determinada por el vector $\mathbf{r}_i$. El *radio de giro* del polímero se define como

$$R_N = \left(\sum_{i=1}^{N} \frac{|\mathbf{r}_i - \mathbf{r}_0|^2}{N} \right)^{1/2}$$

donde $\mathbf{r}_0$ es el vector posición del centro de masas del polímero:

$$\mathbf{r}_0 = \sum_{i=1}^{N} \frac{\mathbf{r}_i}{N}$$

Supongamos que en nuestro programa hemos definido la posición de una partícula como un nuevo tipo de variable, denominada `particula`, del siguiente modo:

```
struct coordenadas_posicion{
    double x, y;
};
typedef struct coordenadas_posicion particula;
```

El objetivo de este ejercicio es escribir en C una función que calcule y devuelva el radio de giro de un polímero. El número de partículas del polímero, N, y las posiciones de las partículas dadas en forma de vector de elementos de tipo `particula`, deben ser pasados como argumentos de la función.

Solución:

```
double calcula_radio(int N, particula C[]) {
    int i,j;
    double R=0;
    particula R0;
    R0.x=R0.y=0;
    for(i=0; i<N; i++) {
        R0.x+=C[i].x;
        R0.y+=C[i].y;
    }
    R0.x=R0.x/N;
    R0.y=R0.y/N;
    for(i=0; i<N; i++) {
        R+=pow(C[i].x-R0.x,2)+pow(C[i].y-R0.y,2);
    }
    return sqrt(R/N);
}
```

29. Un polímero es un agregado de partículas (moléculas, macromoléculas, etc.) que puede adoptar formas muy complicadas. Supongamos un polímero en 2 dimensiones compuesto de N partículas de la misma masa. La posición de la partícula i en el polímero (con $i = 1, \ldots, N$) está determinada por el vector $\mathbf{r}_i$. Se define la distancia media entre dos partículas del polímero, ξ_N, como:

$$\xi_N = \left(\frac{2}{N^2} \sum_{i=1}^{N} \sum_{j=i+1}^{N} |\mathbf{r}_i - \mathbf{r}_j|^2 \right)^{1/2}$$

Supongamos que en nuestro programa la posición de una partícula se guarda en forma de estructura del siguiente modo:

```
struct posicion{
    double x, y;
};
```

El objetivo de este ejercicio es escribir en C una función que calcule y devuelva la distancia media entre dos partículas de un polímero. El número de partículas del polímero, N, y las posiciones de las partículas dadas en forma de vector de estructuras `posicion`, deben ser pasados como argumentos de la función.

Solución:

```
double calcula_distancia(int N, struct posicion C[]) {
    int i,j;
    double R=0;
    for(i=0; i<N; i++) {
        for(j=i+1; j<N; j++) {
            R+=pow(C[i].x-C[j].x,2)+pow(C[i].y-C[j].y,2);
        }
    }
    return sqrt(2*R/(N*N));
}
```