Prueba de Evaluación Continua

Adolfo Hilario Garcia

24 de mayo de 2020

Ejercicio 1

El objetivo del ejercicio 1 es comprobar que el número de desintegraciones en cualquier paso j de la simulación tiene una distribución que se aproxima a una Poisson de media $N(j-1)\,p = \lambda_0\,N(j-1)\,dt$ cuando $N(j-1) \to \infty$ y $p \to 0$. Para ello, vamos a simular el primer paso de la simulación $j = 1$ para cuatro casos diferentes con cantidades de núcleos iniciales y probabilidades dadas por:

$$N_0 = [20, 50, 10^2, 10^3] \qquad p = [0.5, 0.2, 0.1, 0.01]$$

Para cada caso repetiremos este proceso $M = 10^6$ veces. Después se representará gráficamente, junto a la función de distribución de la probabilidad de los procesos de Poisson para $\lambda = N_0\,p = 10$, el histograma del número de desintegraciones observadas en cada caso, en el que se muestre la frecuencia $p(x)$ frente a x calculada de este modo:

$$p(x) = \frac{P(x)}{M} \qquad x = 0, 1, 2, 3, ...$$

donde $P(x)$ es el número de veces que se ha obtenido el resultado x. A continuación se especificará el código de C usado para realizar la simulación pedida.

1.1 Código

Para realizar la simulación descrita en la sección 8 de la guía de enunciados, necesitamos que el programa:

1. Ejecute bucle principal que recorra cada caso `N0[k]`, y cada elemento de la matriz de unos de longitud `N0[k]`. Como cada elemento del vector `N0` tiene un valor distinto, es decir, cada caso almacenará cantidades diferentes de memoria, debemos crear y liberar espacio de memoria a medida que la vayamos necesitando. Para manejar dinámicamente el tamaño del vector `N` utilizamos la función `calloc`

(seccion 7.8.5 Kernighan, Ritchie), que asigna memoria a un array de un de determinado tamaño e inicializa todos los bytes del almacenamiento asignado a cero. Antes de cerrar el bucle, liberamos la memoria que hemos utilizado para poder utilizarla en el siguiente caso (función `free`). De esta forma nos aseguramos un uso eficiente y controlado de la memoria a lo largo de la simulación.

2. Simule el proceso de desintegración de los núcleos a lo largo de j-iteraciones, repitiendo un millón de veces cada caso, rellenando previamente el vector N de unos. A continuación, debe introducir un `if-else` donde, si el número encontrado al recorrer el vector es igual a 1, se genera un número pseudo-aleatorio `U` que se encuentra en el intervalo (0,1) mediante `drand48()` [1], y según el valor de `P0[k]` desintegrará o no el núcleo en cuestión (listado 1). Después de actualizar el vector, la función `cuentanumeros` (expuesta en Funciones creadas), contará los ceros y los guardará en el vector `desintegraciones`, de longitud máxima M.

3. Por último, el programa debe crear las matrices `P` y `p`, donde `P` es el número de veces que se repite cada desintegración en el vector `desintegraciones` de longitud `M` a lo largo de j iteraciones. Entonces, `p` equivale a `P` dividido por el número de muestras. A continuación se muestra el listado del código utilizado para el proceso explicado en estos párrafos.

Una vez realizado el bucle, creamos un archivo llamado `ejercicio1.txt` que guarde los datos de la matriz p como puntos a representar en una gráfica (junto a la solución teórica) mediante GNUplot, donde las filas corresponden al número de desintegraciones y las columnas a los casos correspondientes. A continuación, llamamos a la función `histograma` que genera una gráfica con un valor máximo de x en 25 (a partir del cual apenas hay desintegraciones) de la matriz `p`.

1.2 Resultados

La Figura 1 muestra la distribución de la probabilidad en función del número de desintegraciones para cada caso $(N_0[k], p[k])$. Cómo se puede observar, cuanto mayor es la cantidad inicial de núcleos y menor es la probabilidad de que éstos se desintegren, más se parece la distribución de los puntos a una distribución de Poisson con ecuación:

$$p(x) = \frac{\lambda^x}{x!}\, e^{-\lambda}$$

De esta forma, queda demostrado que cuando $N(j-1) \to \infty$ y $p \to 0$, la distribución de la probabilidad de desintegración se aproxima a una Poisson de media $N_0\, p = 10$.

[1] Previamente se ha tenido que introducir la función `srand48(time(NULL))` que introduce una semilla, en este caso, la hora del ordenador, que cambia cada segundo, afianzando la aleatoriedad del proceso.

Listado 1: *Bucle del ejercicio 1*

```
for (k=0; k<ncasos; k++)
{
    longitud_N = N0[k];

    N = (int *)calloc(longitud_N, sizeof(int));

    for(m=0; m<M; m++)
    {
        for(i=0; i<longitud_N; i++)
        {
            N[i] = 1;   //cada elemento del vector es igual a 1
        }

        for(j=0; j<niteraciones; j++)
        {
            for(i=0; i<longitud_N; i++)
            {
                if(N[i]==1) //si no se ha desintegrado
                {
                    U = drand48();
                    if(U<=P0[k]){N[i] = 0;}
                }
            }
        }
        sucesos[m] = cuentanumeros(N, longitud_N, 0);
    }

    for(j=0; j<=max_sucesos; j++)
    {
        P[k][j] = cuentanumeros(sucesos, M, j);
        p[k][j] = (double)P[k][j]/M;
    }
        free(N);
}
```

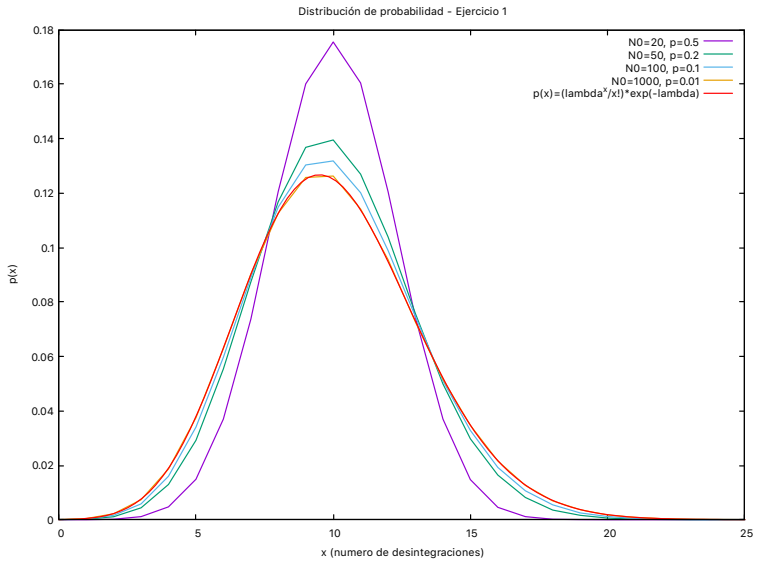

Figura 1: *Función de distribución de la probabilidad para cada caso*

Listado 2: *Creación del archivo de datos* `ejercicio1.txt`

```
FILE * archivodatos = fopen("ejercicio1.txt", "w");

for (i=0;i<=25;i++)
{
    fprintf(archivodatos, " %d %f %f %f %f \n",
         i, p[0][i], p[1][i], p[2][i], p[3][i]);
}
fclose(archivodatos);
histograma();
```

1.3 Funciones creadas

A continuación se muestran las funciones creadas para la ejecución del programa:

1. `cuentanumeros(int V[], int ncomponentes, int numero)`: devuelve el número de un determinado número a lo largo de un vector de una determinada extensión.
2. `max(int V[], int ncomponentes)`: calcula el valor máximo de un vector de una determinada longitud.
3. `histograma()`: genera un histograma a partir de un archivo de datos. Configura las leyendas, el color, y otros comandos de GNUplot y los ejecuta en una misma gráfica imprimiéndolos en un canal creado previamente por la orden `popen`.A continuación se muestra el código de ésta función, que iremos modificando según las demandas de cada ejercicio.

Listado 3: *Función* `histograma()`

```
int histograma()
{
    #define NUM_COMANDOS 8
    int i;
    char * comandosGNUplot[] =
    {
        "set ylabel 'p(x)'",
        "set xlabel 'x (numero de desintegraciones)'",
        "plot 'ejercicio1.txt' using 1:2 with lines title 'N0=20, p=0.5'",
        "replot 'ejercicio1.txt' using 1:3 with lines title 'N0=50, p=0.2'",
        "replot 'ejercicio1.txt' using 1:4 with lines title 'N0=100, p=0.1'",
        "replot 'ejercicio1.txt' using 1:5 with lines title
            'N0=1000, p=0.01'",
        "replot [0:25] (10**x/gamma(x+1))*exp(-10) with lines lc rgb 'red'
            title 'p(x)=(lambda^x/x!)*exp(-lambda)'"
    };

    FILE * p = popen("gnuplot -persist", "w");

    for (i=0;i<NUM_COMANDOS;i++)
    {
        fprintf(p, "%s\n", comandosGNUplot[i]);
    }
    return 0;
}
```

Ejercicio 2

El objetivo de este ejercicio es comprobar que el comportamiento obtenido en la simulación de desintegración completa coincide con la solución analítica $N(t) = N_0\, e^{-\lambda_0 t}$, donde N es el número de núcleos restantes en el tiempo t. Partiremos de un conjunto inicial de 10 000 núcleos y utilizaremos diferentes probabilidades `p[k]`:

$$p = [0.5, 0.1, 0.01, 0.001]$$

para realizar una simulación completa donde el paso de tiempo es $dt = p/\lambda_0$. Se representará gráficamente las parejas de puntos $(t(j), N(j))$ junto a la solución teórica.

2.1 Código

Los valores de `N0` y `p` se introducirán por la línea de comandos mediante las funciones `atoi` y `atof`, que leerán los valores que introduzcamos y los asignará a las variables correspondientes, bien como enteros o bien como *floats* o *doubles*.

Para cada caso `P0[k]`, configuramos el programa de forma que:

1. La primera fila de N, con tantas filas como casos y tantas columnas como núcleos iniciales haya, incluya el número inicial de núcleos N_0.

2. Para cada caso, genere un vector de unos `n` de longitud `N0` y realize el algoritmo ya explicado en el Ejercicio 1, repitiendo el proceso mientras que el resultado de la iteración realizada sea mayor que cero, guardando el número de núcleos restantes en cada vuelta mediante la función `cuentanumeros` en la matriz `N`.

3. Anote el máximo número de iteraciones `nmax` y lo guarde en un archivo para usarlo más tarde en el Ejercicio 3.

4. Se creen cuatro archivos `ejercicio2_n.txt` (donde $n = k + 1$) para cada k-caso en que solo se guarden los valores de `t[k][j]` y `N[k][j]` si, al dividirlas por el paso a qué estén sometidas, el resto es cero. De esta forma podemos asignar un paso a cada probabilidad, escribiéndose en los archivos solamente el número de datos que necesitamos. Para ello, se ha utilizado la función `sprintf`, que asignará la extensión `.txt` y el índice para cada archivo, y un `if` dentro de un bucle `for` (ver listado 4). También se creará un archivo para el cuarto caso a parte, con el objetivo de que se representen solo los núcleos restantes para $t > 50$ años.

5. Represente gráficamente los resultados obtenidos mediante funciones muy similares a la función `histograma` del ejercicio anterior: `graficaplot` y `graficaplot50`.

Listado 4: *Creación de los archivos con paso*

```
FILE *archivodatos;

char nombrebase[100] = "ejercicio2";
char nombrearchivo[100] = "";

sprintf(nombrearchivo,"%s",nombrebase);
sprintf(nombrearchivo,"%s_%d",nombrearchivo,k+1);
sprintf(nombrearchivo,"%s.txt",nombrearchivo);

archivodatos = fopen(nombrearchivo, "w");
for (i=0;i<nmax[k];i++)
{
    if(i % paso[k]==0){fprintf(archivodatos,"%f %d \n",t[k][i], N[k][i]);}
}
fclose(archivodatos);
```

Resultados

Como se puede apreciar en la Figura 2, la variación del número de núcleos sin desintegrar sufre un proceso de decrecimiento exponencial similar al del caso teórico, también representado en la gráfica. Cuánto menor es la probabilidad de desintegración, mayor es la exactitud de la representación. En definitiva, el comportamiento obtenido en la simulación de desintegración coincide con la expresión matemática.

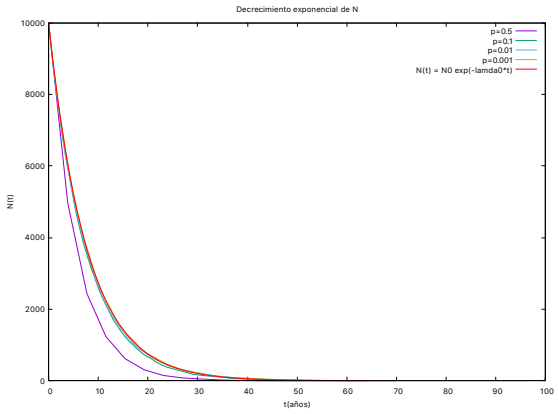

Figura 2: *Representación gráfica de la evolución del número de núcleos sin desintegrar a lo largo del tiempo para cada caso*

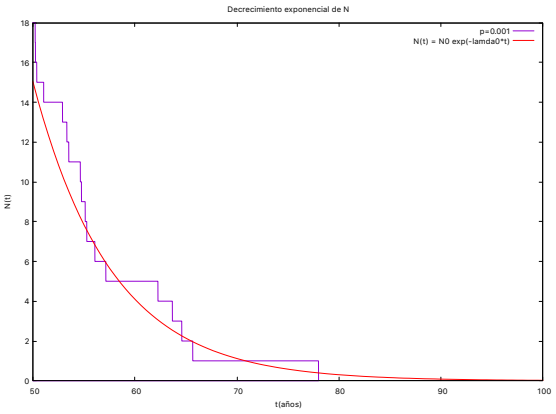

Figura 3: *Representación gráfica de la evolución de los núcleos restantes para tiempos mayores que 50 años.*

Para tiempos mayores que 50 años, se obtiene la representación mostrada en la Figura 3. Como se puede apreciar, los datos simulados oscilan alrededor de la solución analítica. Cabe recordar que en este ejercicio $M = 1$, si aumentásemos el número de muestras y cada punto fuese la media obtenida de núcleos restantes para una determinada j-iteración (como ya sucede en los ejercicios Ejercicio 3 y Ejercicio 4), los datos obtenidos por simulación se ajustarían más al modelo matemático ideal.

Ejercicio 3

El objetivo de este ejercicio es comprobar que los intervalos de tiempo transcurridos entre eventos en los procesos de Poisson tienen distribuciones exponenciales. Para ello utilizaremos la función de distribución acumulada (FDA), que viene dada por la expresión:

$$F(t) = \text{prob}(X \leq t) = \frac{N_0 - \frac{1}{M}\sum_{k=1}^{M} N_k(j)}{N_0}$$

donde X es el tiempo que tarda un núcleo particular en desintegrarse, $t = j\,dt$ y el numerador del cociente situado en la parte derecha de la igualdad representa el número de núcleos que, de los N_0 iniciales, se ha desintegrado en el tiempo t. Dividiendo este valor por N_0 se obtiene la probabilidad de desintegración en un tiempo X menor que t, siendo M el número de k-simulaciones. Representaremos en gráficas separadas un primer caso con 1 simulación y un segundo con $M = 100$, ambas junto a la FDA teórica $F(t) = 1 - e^{-\lambda_0 t}$, para $N_0 = 10^3$ y $p = 10^{-3}$.

3.1 Código

Los valores de N_0 y p, junto al paso y el número de muestras se introducirán por la línea de comandos de forma similar al Ejercicio 2. Antes de iniciar los bucles, leemos el archivo de datos en el cual se encuentra el número máximo de iteraciones y se asigna el valor a la variable `nmax`.

Para cada caso `M[m]`, configuramos el programa de forma que:

1. Defina la matriz `N` con `nmax` filas y `M[m]` columnas (esto es, un máximo de 100), donde el número de núcleos de la primera fila ($j = 0$) es N_0.

2. Para cada simulación k, esto es, para cada columna de la matriz N, realize el algoritmo descrito en el Ejercicio 2 hasta que todos los elementos del vector `n` queden en cero, pasando entonces a la siguiente columna hasta que rellene por completo la matriz (veáse listado 5).

3. Para cada fila, se define la matriz F con `nmax` filas y tantas columnas como casos de simulaciones. Se calcula cada `sumatorio` para cada caso y se realiza la operación descrita en la ecuación de la FDA. Definimos después la matriz del tiempo para cada iteración (filas) y cada caso (columnas) como $t = j\,dt$ en el mismo bucle `for`.

4. Definimos un nuevo contador j_2 para que las matrices F y t solamente guarden los datos de j-iteraciones cuyo resto respecto al paso definido previamente sea cero. Recomiendo que el paso sea de 50 para obtener resultados nítidos.

5. Creamos un archivo de datos con los datos de F y t para el primer caso, esto es, `t[j2][0]` y `F[j2][0]`, y otro para el segundo caso: `t[j2][1]` y `F[j2][1]`.

6. LLamamos a la función `graficaplot` para que nos represente cada caso en una gráfica duplicando cerrando y volviendo a abrir el puntero que apunta al canal creado por `popen`.

Listado 5: *Matriz de la FDA en C*

```
do{
    for(i=0; i<N0; i++)
    {
        if(n[i]==1)
        {
            U = drand48();
            if(U<=P0){n[i] = 0;}
        }
    }
    N[j][k] = cuentanumeros(n, N0, 1);

    j++;
} while(j<=nmax);

j2 = 0;
for(j=0;j<nmax;j++)
{
    if(j % paso==0) .
    {
        F[j2][m] = (double)((N0-(double)(sumatorio(N[j], M[m])/M[m]))/N0);
        t[j2][m] = (double)dt*j2*paso;
        j2++;
    }
}
```

3.2 Resultados

A continuación se muestran las dos gráficas generadas por el programa. Cómo se puede observar, para $M = 1$ (Figura 4) la distribución de puntos es menos exacta que en la Figura 5, en la que se realizan 100 simulaciones.

En conclusión, para los valores dados de N_0, p, λ_0 y para los distintos valores de M, la probabilidad de que ocurra una desintegración en un tiempo $(X \leq t)$ se distribuyen en el tiempo siguiendo la FDA de la forma $F(t) = 1 - e^{-\lambda_0 t}$.

3.3 Función `sumatorio()`

Para este ejercicio, se ha creado la función `sumatorio(int V[], int ncomponentes)`, que devuelve el valor la suma de todos los elementos de un vector de *integers*.

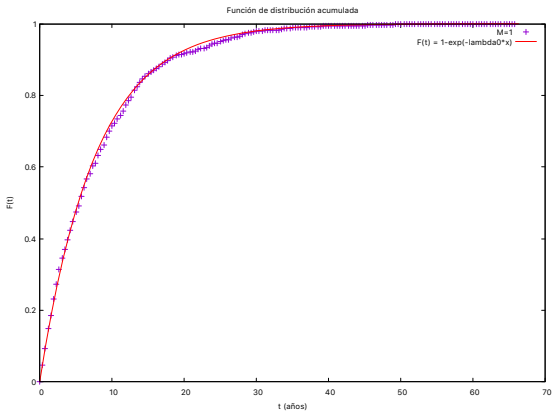

Figura 4: *Representación gráfica de la FDA para M=1 en función del tiempo*

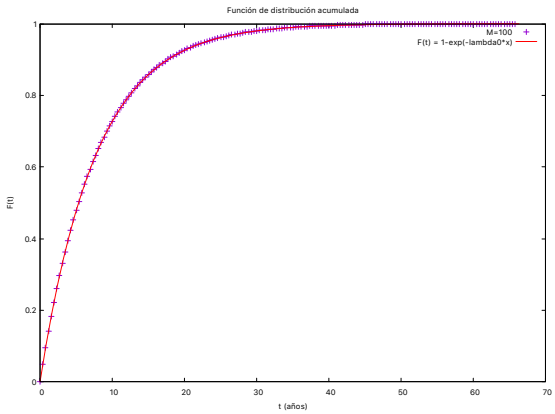

Figura 5: *Representación gráfica de la FDA para M=100 en función del tiempo*

Ejercicio 4

El objetivo de este ejercicio es demostrar que el proceso es *markoviano*. Para ello, verificaremos que la probabilidad de que un núcleo que no se ha desintegrado en un tiempo t tampoco lo haga en el próximo intervalo Δt, es independiente de t e igual a $e^{-\lambda_0 t}$. Este probabilidad puede ser demostrada del siguiente modo:

$$\text{prob}(X > t + \Delta t \,|\, X > t) = \frac{\frac{1}{M}\sum_{k=1}^{M} N_k(j+l)}{\frac{1}{M}\sum_{k=1}^{M} N_k(j)} = \frac{\sum_{k=1}^{M} N_k(j+l)}{\sum_{k=1}^{M} N_k(j)}$$

donde $l = 130$, $\Delta t = l\,dt = 1$, $N_0 = 10^4$, $p = 10^{-3}$ y los tres casos a tratar son: $M = [10, 100, 1000]$. Se representarán las gráficas por separado junto a la recta $e^{-\lambda_0 t}$.

4.1 Código

El código de este ejercicio es muy similar al del Ejercicio 3, pero cambiando la función `F[j][m]` por la función presentada en la introducción a este apartado que llamaremos `G[j][m]`, y haciendo que el bucle recorra las j-filas mientras la iteración en cuestión sea menor que `nmax-1` (recordemos que $l = 130$) y el denominador de `G[j][m]` sea distinto de cero, como se muestra en el listado 6.

No obstante, me ha resultado imposible realizar el tecer caso de la simulación $M = 1000$, debido a un problema de *Segmentation fault: 11*. He pasado 2 tardes intentando de todo, pero no encuentro la explicación de por qué para valores de M altos o para tres casos no realiza el código. Al parecer el programa accede a posiciones de memoria que no le corresponde, y no sé cómo solucionarlo. En consecuencia, el programa solamente devuelve las gráficas para los dos primeros casos. Si el equipo docente tiene una

Listado 6: *Creación de la matriz G*

```
j = 0;
j2 = 0;
while((j<nmax-1) && (sumatorio(N[j], M[m])!=0))
{
    if(j % paso==0)
    {
        G[j2][m] = (double)(sumatorio(N[j+1], M[m]))/(sumatorio(N[j], M[m]));
        t[j2][m] = (double)dt*j2*paso;

        j2++;
    }
    j++;
}
```

solución a este problema, por favor, hagánmelo saber mediante la plataforma virtual o enviándome un correo a ahilario4@alumno.uned.es. A lo largo de la PEC he tenido problemas con el *Segmentation fault: 11* en otros ejercicios, logrando resolverlos todos excepto éste.

4.2 Resultados

Las figuras 6 y 7 muestran que la probabilidad de que un núcleo que no se ha desintegrado en un tiempo t tampoco lo haga en el próximo intervalo Δt, es independiente de t e igual a $e^{-\lambda_0 t}$ para timepos menores que 30 años. Además, la exactitud aumenta con el número de muestras. Puedo predecir con bastante seguridad que para $M = 1000$ la exactitud será aún mayor.

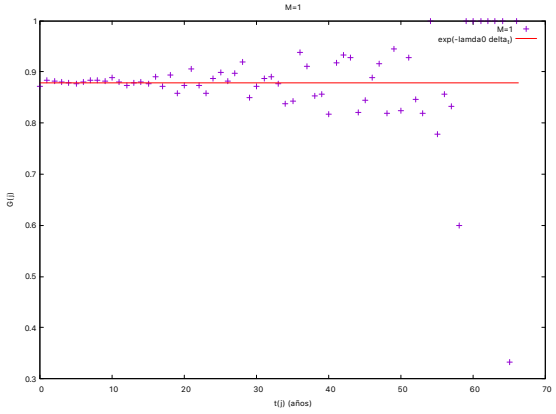

Figura 6: *Representación de la probabilidad dada por la función $G(t)$ junto al valor teórico para $M = 1$*

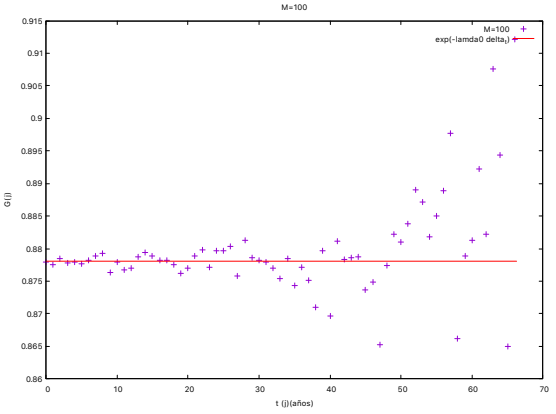

Figura 7: *Representación de la probabilidad dada por la función $G(t)$ junto al valor teórico para $M = 100$*

En conclusión, con esto queda demostrado que el proceso es *markoviano*, es decir, que no tiene memoria. Quedan así demostradas cuatro propiedades fundamentales de la desintegración radiactiva del ^{60}Co.

Sobre las funciones `atof()` y `atoi()`

A continuación se especifica el orden en que se deben introducir los valores de las variables en los ejercicios 2, 3 y 4 por línea de comandos cómo argumentos de la función `main`:

- **Ejercicio 2:** `N0 p1 p2 p3 p4 paso1 paso2 paso3 paso4`
- **Ejercicio 3:** `N0 P0 m1 m2 paso`
- **Ejercicio 4:** `N0 P0 m1 m2 l`

Si por error se introducen menos elementos de los deseados el programa completa las variables que faltan con los valores aportados por el ejercicio (excepto el paso, que creo más conveniente que sea de 50 unidades). El listado 7 muestra cómo se ha realizado esto en el Ejercicio 3. De hecho, si no se asigna ningún valor al archivo *a.out* generado tras compilar, el programa funcionará con los valores asignados por defecto.

Listado 7: *Asignación de valores a las variables en el ejercicio 3*

```
1  if (argc >=2){
2      N0 = atoi(argv[1]);
3  }else{
4      N0 = 1000;
5  }
6
7  if(argc >=3){
8      P0 = atof(argv[2]);
9  }else{
10     P0 = 0.001;
11 }
12
13 if (argc >=4){
14     m1 = atoi(argv[3]);
15 }else{
16     m1 = 1;
17 }
18
19 if (argc >=5){
20     m2 = atoi(argv[4]);
21 }else{
22     m2 = 100;
23 }
24
25 if (argc >=6){
26     paso = atoi(argv[5]);
27 }else{
28     paso = 50;
29 }
```