Problemas de Cálculo Numérico con C

1. Para calcular manualmente la raíz de un número hacemos una primera aproximación a ella, luego dividimos el número por esta estimación y calculamos la media entre la estimación y ese cociente; el resultado nos da una nueva estimación y volvemos a repetir el proceso. Escríbase un programa en C que realice estas iteraciones y calcule el error respecto al valor exacto obtenido con `sqrt()` en función del número de veces que se haya iterado. Considere números entre 1 y 10, y que la primera estimación es siempre 1; deténgase cuando el error sea inferior a 10^{-6}.

   Solución: Ver código `raiz_cuadrada.c`

2. Una manera de eliminar el ruido de una señal adquirida en función del tiempo es sustituir cada valor por la media de los valores en una ventana que comprende los n anteriores y los n posteriores a él. Escriba una función que, recibido un primer array con los datos adquiridos, un segundo array en el que se espera la respuesta (del mismo tamaño que el primero) y el número n de vecinos antes y después, calcule en el segundo array los datos promediados. [**Nota**: téngase en cuenta que los primeros datos, no tienen n anteriores, por lo que el promedio contendrá menos valores; lo mismo para los últimos de la serie]

   Solución:

```
#define LBUFFER 1024
void suaviza(float x[], float y[], int n) {
     int k, m;
     float s, w;

     for(k=0; k < LBUFFER; k++) {
          s=0;
          w=0;
          for(m=-n; m<=n; m++) {
               if( 0<=k+m && k+m<LBUFFER ) {
                    s=s+x[k+m];
                    w++;
               }
          }
          y[k]=s/w;
     }
     return;
}
```

3. El objetivo de este ejercicio es escribir un programa que proporcione una estimación del número π por el método numérico de Monte Carlo. Para ello supondremos que disponemos de la función `rnd()`, declarada del siguiente modo

```
double rnd();
```

que devuelve cada vez que es llamada un número real (double) aleatorio uniformemente distribuido entre -1 y 1. El método consiste en generar parejas de números aleatorios que corresponderán a las coordenadas de puntos (x, y). De este modo, los puntos generados caerán en el cuadrado de lado 2 centrado en el origen de coordenadas (0, 0). La probabilidad P de que un punto caiga en el círculo de radio 1 centrado en el origen vendrá dada por el área total del círculo dividida por el área del cuadrado:

$$P = \frac{\text{área}_{\text{círculo}}}{\text{área}_{\text{cuadrado}}} = \frac{\pi R^2}{(2R)^2} = \frac{\pi}{4}$$

El programa debe aproximar esta probabilidad para N puntos (valor que debe ser introducido por línea de comandos al ejecutar el programa). Si definimos como n el número de esos N puntos que han caído dentro del círculo, tendremos que

$$P \simeq \frac{n}{N}$$

Igualando este valor con el resultado anterior tenemos que nuestra estimación de π es

$$\pi \simeq \frac{4n}{N}$$

Éste es el valor que debe retornar el programa para un N dado.

Solución:

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

int main(int argc, char** argv) {
    int N,n,i;
    double x,y;
    N=atoi(argv[1]);
    n=0;
    for (i=0; i<N; i++) {
        x=rnd();
        y=rnd();
        if (x*x+y*y<=1) n++;
    }
    printf("La estimacion del valor de Pi por el metodo de Monte Carlo
        con %d puntos es %g\n", N, 4.0*n/N);
    return 0;
}
```

4. Cómo se deberían declarar e inicializar las variables `msk`, `n`, `b`, y `x` para que el siguiente código compile sin errores y funcione correctamente.

```
for(n=0; n<32; m++) {
    b= msk & (1<<n);
    if(b) {
        x=x+(double)rand()/RANDOM_MAX;
    }
}
printf("Resultado: %f\n", x);
```

Solución: La variable `msk` deberá ser un entero `int` o `long` (para que, al menos, tenga 32 bits). Las variables `n` y `b` pueden ser cualesquiera enteros (en particular, para `b` se podría usar un `char`). `x` debería ser un `double`, pero también podría ser un `float`.

5. Indique cuál debería ser el resultado de compilar y ejecutar este programa en C. ¿Qué debería contener el archivo "`datos.dat`"?

```
#include <stdio.h>
int main(int argc, char argv) {
    int n=0;
    double x,y, sx=0,sy=0;
    FILE *fIn=fopen("datos.dat", "r");
    while( !feof(fIn) ) {
        fscanf(fIn, "%f%f", &x,&y);
        sx+=x;
        sy+=y;
        n++;
    }
    fclose(fIn);
    printf("xm=%g, ym=%g\n", sx/n, sy/n);
    return 0;
}
```

Solución: El archivo datos.dat debería contener una lista de pares de números (X,Y), separados por espacios (posiblemente un par por línea). El programa lee par a par, acumula la suma de valores de las X y las Y, cuenta el número de pares leídos y, finalmente, imprime el valor medio de las X y el valor medio de la Y.

6. La integral definida de una función en un intervalo $\int_a^b f(x)\,dx$, puede aproximarse numéricamente mediante la integral —o suma— de Riemann $S(n) = \sum_{i=1}^{n} f(x_i) \cdot (x_i - x_{i-1})$, donde los puntos x_i corresponden a una partición del intervalo de integración $[a, b]$: $x_0 = a < x_1 < x_2 < \cdots < x_{n-1} < x_n = b$. Se trata por tanto de una aproximación al área encerrada debajo de la curva de la función $f(x)$.

   Si la función está acotada en el intervalo y es continua, o bien tiene un conjunto numerable de discontinuidades, se cumple que

$$\int_a^b f(x)\,dx = \lim_{n\to\infty} S(n)$$

   Escribir un programa que aproxime la integral de una función mediante la integral del Riemann. Elegir la expresión matemática $f(x)$ que se desee integrar y definirla en una función externa que será llamada desde la función `main()`. Deberemos introducir por línea de comandos el intervalo de integración $[a, b]$. El programa deberá calcular la integral de Riemann para diferentes valores de n y exportar los valores de $S(n)$ obtenidos a un archivo de modo que luego puedan ser representados con Gnuplot y así poder comprobar que estos tienden al límite dado por la integral.

   Por simplicidad, supondremos que la partición del intervalo es una potencia de 2, de modo que n variará entre 2^{n_1} y 2^{n_2}, aumentando en un factor de 2 en cada iteración. Los valores n_1 y n_2 también deberán ser introducidos por línea de comandos.

   Solución: Ver código `integral_Riemann.c`

7. Obtenga una aproximación a la integral de Riemann con una cierta precisión, indicada por línea de comandos. Para ello deberá comparar el resultado obtenido para cada n con el obtenido para el n anterior: cuando el valor absoluto de la diferencia sea menor que la precisión requerida, se entenderá que esa es la suma de Riemann que aproxima la integral definida.

   Solución: Ver código `integral_Riemann_precision.c`

8. Explicar detalladamente qué realiza el siguiente programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NMAXPTS 10000
int main(int argc, char** argv) {
     int i, j, n, npts, CONTROL;
     float num[NMAXPTS], x;
     FILE *fin, *fout;
     fin=fopen("datos.dat", "r");
     n=fscanf(fin, "%f", &x);
     if(n==1) {
          num[0]=x;
          npts=1;
     }
     do {
          n=fscanf(fin, "%f", &x);
          if(n==1) {
               CONTROL=0;
               for(i=0; i<npts; i++) {
                    if(x < num[i]) {
                         for(j=npts; j>i; j--) {
                              num[j]=num[j-1];
                         }
                    num[i]=x;
                    CONTROL=1;
                    break;
                    }
               }
               if(!CONTROL) num[npts]=x;
               npts++;
          }
     } while( n==1 && npts<=NMAXPTS );

     fclose(fin);
     fout=fopen("salida.dat", "w");
     for(i=0; i<npts; i++) {
          fprintf(fout, "%g\n", num[i]);
     }
     fclose(fout);
     return 0;
}
```

   Solución: El programa lee de un archivo de datos `datos.dat` una lista de números reales y los imprime de forma ordenada en otro archivo `salida.dat`.

9. En el archivo de texto plano `datos.dat` tenemos N pares de puntos reales (x_i, y_i), $i = 1, \ldots N$, que representan los valores de la función $y(x)$ en el conjunto discreto de valores de x en los que se ha dividido el intervalo $[x_1, x_N]$. Escribir un programa que obtenga la derivada de la función en cada punto y exporte las parejas de datos a un archivo `derivada.dat`.

   Para obtener la derivada de un conjunto discreto de puntos consideraremos la aproximación basada en diferencias finitas centradas en el punto:

$$y'(x_i) = \frac{1}{2}\left(\frac{y_i - y_{i-1}}{x_i - x_{i-1}} + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}\right)$$

   que, como puede observarse, calcula el promedio de las pendientes entre el punto y sus dos vecinos más próximos. En los puntos extremos sólo consideraremos la pendiente con su vecino más próximo:

$$y'(x_1) = \frac{y_2 - y_1}{x_2 - x_1} \qquad y'(x_N) = \frac{y_N - y_{N-1}}{x_N - x_{N-1}}$$

   Solución: Ver código `derivada_numerica.c`

10. Escribir un programa que lea desde un archivo de texto `datos.dat` una serie de valores numéricos x_i (un valor, no necesariamente entero, por cada línea del archivo) y escriba luego por la salida estándar los siguientes resultados:

    - número de valores leídos (N)
    - media de los valores leídos, $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$
    - desviación típica de los valores leídos, $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2 - \bar{x}^2}$

    Téngase en cuenta que el número de valores puede ser arbitrariamente grande (no se conoce a priori), por lo que no se deben guardar los x_i en un array.

    Nota: Para leer los valores desde el archivo use la función fscanf como

    ```
    int fscanf(FILE*, char*, double*)
    ```

    que recibe como argumentos el archivo desde el que leer, una cadena de formato, en este caso "%lf", y la posición de memoria de una variable de tipo double donde guardará el valor leído, y retorna 0 si no ha podido leer el valor (por haber llegado al final del archivo) o 1 si ha podido leer el valor double.

    Solución: Ver código `estadistica_datos.c`

11. El método de Newton-Raphson es un método numérico para resolver ecuaciones algebraicas no-lineales. Este método está basado en el hecho de que las soluciones de la ecuación $h(x) = g(x)$ son las raíces de la función $h(x) - g(x)$. Para obtener numéricamente estas raíces se utiliza el algoritmo iterativo de Newton-Raphson:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

    donde $f'(x_n)$ es la derivada de la función $f(x_n)$ evaluada en el punto x_n. Partiendo de una semilla inicial x_0 la sucesión convergerá (no siempre, dependerá de las características de la ecuación y de la semilla de partida) a una de las raíces de la función $f(x)$. En general el algoritmo concluye cuando el error relativo entre dos aproximaciones sucesivas es menor que una cierta tolerancia fijada, que llamaremos E, que se considera como el error relativo de la aproximación:

$$\frac{|x_{n+1} - x_n|}{|x_{n+1}|} < E$$

El objetivo de este ejercicio es escribir un programa que calcule las soluciones reales de la ecuación $e^x = x^3$. La función $f(x)$ cuyas raíces queremos obtener, y su derivada $f'(x)$ deberán ser definidas en dos funciones. Los valores de la semilla x_0 y del error E serán introducidos por línea de comandos (como argumentos de la función `main()`) cuando se ejecute el programa. El programa deberá imprimir en pantalla el valor aproximado de la raíz obtenida con esa tolerancia.

Solución: Ver código `Newton_Raphson.c`

12. La aproximación numérica de 5 puntos para la derivada primera de una función $f(x)$ tiene la forma

$$f'(x) \approx \frac{1}{12h}\left[f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)\right]$$

donde h es una intervalo real pequeño ($h \ll 1$). El error de esta aproximación es, aproximadamente, h^4.

Escriba una función en C que calcule la derivada de una función $f(x)$ en un cierto punto x_0 con una cierta tolerancia que llamaremos `err`. Consideraremos que la función matemática que queremos derivar, $f(x)$, se ha definido previamente en el programa como por ejemplo:

```
double funcion(double x)
{
    double y;
    //...
    return y;
}
```

La función que calcula la derivada debe recibir como argumentos la función matemática que queremos derivar, el punto x_0 en el que queremos obtener la derivada y la tolerancia de la aproximación `err`. El intervalo h que deberemos emplear estará dado por $h = \texttt{err}^{1/4}$

Solución: Este ejercicio admite muchas soluciones, todas ellas válidas. A continuación mostramos tres posibilidades. Hemos incluido la función `main` (que no se pedía en el enunciado del ejercicio) para mostrar cómo se realizaría la llamada a la función pedida, que hemos denominado `derivada()`.

- **Solución 1**

```
double derivada (double f(double), double x0, double err) {
        double h;
        h=pow(err,1./4);
        return 1/(12*h)*(f(x0-2*h)-8*f(x0-h)+8*f(x0+h)-f(x0+2*h));
}

int main(int argc, char** argv) {
         double x0, err;
         x0=atof(argv[1]);
         err=atof(argv[2]);
         printf("La derivada de la funcion en el punto %g, con una
             tolerancia de %g, es %g\n", x0, err, derivada(funcion,
             x0, err));
         return 0;
}
```

- **Solución 2**

```
double derivada (double (*f)(double), double x0, double err) {
        double h;
        h=pow(err,1./4);
        return 1/(12*h)*(f(x0-2*h)-8*f(x0-h)+8*f(x0+h)-f(x0+2*h));
}

int main(int argc, char** argv) {
         double x0, err;
         x0=atof(argv[1]);
         err=atof(argv[2]);
         printf("La derivada de la funcion en el punto %g, con una
             tolerancia de %g, es %g\n", x0, err, derivada(funcion,
             x0, err));
         return 0;
}
```

- **Solución 3**

```
double derivada (double x0, double err) {
        double h;
        h=pow(err,1./4);
        return 1/(12*h)*(funcion(x0-2*h)-8*funcion(x0-h)+8*funcion(
            x0+h)
        -funcion(x0+2*h));
}

int main(int argc, char** argv) {
         double x0, err;
         x0=atof(argv[1]);
         err=atof(argv[2]);
         printf("La derivada de la funcion en el punto %g, con una
             tolerancia de %g, es %g\n", x0, err, derivada(x0, err))
             ;
         return 0;
}
```

13. El *teorema de Bolzano* del análisis matemático establece que si una función continua $f(x)$ toma valores de distinto signo en los extremos de un intervalo $[a, b]$, es decir $f(a) \cdot f(b) < 0$, entonces esa función tiene al menos una raiz en dicho intervalo, es decir, existe al menos un valor $x_0 \in [a, b]$, tal que $f(x_0) = 0$. Este teorema es la base teórica del *método de bisección*, un método numérico iterativo para encontrar las raíces de una función. Supongamos que tenemos el intervalo anterior en el que se cumplen las premisas del teorema de Bolzano. En la primera iteración se calcula el punto medio del intervalo x_m y se evalúa la función en ese punto. Si $f(x_m) = 0$ entonces hemos encontrado la raíz buscada. Si no es así, debemos determinar en cuál de los dos subintervalos la función cambia de signo. De este modo, si $f(a) \cdot f(x_m) < 0$ nos quedamos con el intervalo $[a, x_m]$, y si $f(b) \cdot f(x_m) < 0$ nos quedamos con el intervalo $[x_m, b]$. Si repetimos el proceso de forma iterativa sobre el intervalo resultante iremos acotando el punto donde la función se anula. Asumiremos que la función sólo tiene una raíz en el intervalo inicial, de modo que sólo una de las dos condiciones se cumplirá en cada paso. Después de n iteraciones, nuestro intervalo tendrá la forma $[a_n, b_n]$, con $a_0 = a$ y $b_0 = b$, y nuestra aproximación numérica de la raiz vendrá dada por el punto medio $x_0 = (a_n + b_n)/2$, con un error ϵ que está acotado mediante la expresión

$$\epsilon < \frac{b-a}{2^n}$$

El objetivo de este ejercicio es escribir un programa que calcule e imprima en pantalla la aproximación numérica de la raíz de una cierta función $f(x)$ obtenida por el método de bisección, con un error menor que una cierta tolerancia `Err`. El programa debe seguir el esquema indicado más abajo. La función cuya raíz queremos determinar ha sido definida previamente y devuelve el valor que toma para cada valor de x pasado como argumento de la función. Los extremos del intervalo inicial $[a, b]$ y la tolerancia del error, `Err`, deberán ser pasados por línea de comandos como argumentos de la función `main()`. Por supuesto, asumiremos que existe una única raíz de la función dentro del intervalo.

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

double funcion(double x) {
    double f;
    ...     //f=f(x)
    return f;
}

int main(int argc, char **argv) {
     ...
     return 0;
}
```

Solución: Ver código `metodo_biseccion.c`

14. Supongamos que tenemos la siguiente ecuación diferencial ordinaria de primer orden:

$$\frac{dy}{dt} = f(t, y)$$

donde $f(t, y)$ es una función conocida que depende del tiempo t y de la variable y. La solución de esta ecuación es una función $y(t)$ que verifica la ecuación anterior y también la condición inicial $y(t_0) = y_0$.

El *método de Runge-Kutta de cuarto orden* es uno de los método más utilizados para la integración numérica de este tipo de ecuaciones diferenciales. Este médodo consiste en discretizar la ecuación anterior de la siguiente forma:

$$\begin{aligned} y_{i+1} &= y_i + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\ t_{i+1} &= t_i + h \end{aligned}$$

donde

$$\begin{aligned} k_1 &= f(t_i, y_i) \\ k_2 &= f\left(t_i + \frac{1}{2}h,\, y_i + \frac{1}{2}hk_1\right) \\ k_3 &= f\left(t_i + \frac{1}{2}h,\, y_i + \frac{1}{2}hk_2\right) \\ k_4 &= f\left(t_i + h,\, y_i + hk_3\right) \end{aligned}$$

De este modo la ecuación se resuelve de forma iterativa en pasos de tiempo h, dando como resultado una serie de puntos (t_i, y_i) con $i = 0, 1, 2, ..., N$, los cuales representan la aproximación numérica a la solución exacta $y(t)$, donde N es el número de pasos considerado en la integración.

El objetivo de este ejercicio es escribir un programa en C que implemente el método de Runge-Kutta para resolver una ecuación diferencial con la forma anterior: $dy/dt = f(t, y)$.

El programa debe cumplir con la siguiente estructura.

En primer lugar debemos crear una función externa a `main` en la que definimos la ecuación diferencial que queremos integrar numéricamente, y que nos devuelve el valor $f(t, y)$ para un cierto valor de t y de y que son pasados como argumentos:

```
double f(double t, double y) {
    double funcion;
    ...
    funcion=...;  // la funcion calcula el valor f(t,y) y lo asigna a la
        variable funcion
    return (funcion);
}
```

El estudiante **deberá proponer** una ecuación diferencial y rellenar el cuerpo de esta función `f(t,y)`.

A continuación tenemos que crear otra función externa a `main` que calcule los términos k_1, k_2, k_3 y k_4 en cada paso de la integración para unos valores de t y de y. Para ello deberá llamar a la función anterior `f(t,y)`. Mientras las variables t e y son pasadas por valor, las variables k_1, k_2, k_3 y k_4 deben ser pasadas por referencia. Por lo tanto, la función deberá estar declarada del siguiente modo:

```
void calcula_coeficientesRK4(double t, double y, double h,
                             double *k1, double *k2, double *k3, double *k4)
```

Finalmente, la función `main` deberá recibir como argumentos la condición inicial t_0 e y_0, el paso del integrador h, el número de pasos N y el nombre del archivo en el que se imprimirán los datos. Estos argumentos deberán ser introducidos por línea de comandos. La función `main` deberá implementar el algoritmo de integración de Runge-Kutta y exportar al archivo de datos los puntos (t_i, y_i) con $i = 0, \ldots, N$ en forma de dos columnas de datos.

Solución: Ver código `Runge_Kutta_cuarto_orden.c`

15. El *método de Heun* es un conocido método de integración numérica de ecuaciones diferenciales ordinarias de primer orden:

$$\frac{dy}{dt} = g(t, y)$$

donde $g(t, y)$ es una función conocida que depende del tiempo t y de la variable y. La solución de esta ecuación es una función $y(t)$ que verifica la ecuación anterior y también la condición inicial $y(t_0) = y_0$.

Este método consiste en discretizar la ecuación anterior de la siguiente forma:

$$y_{n+1} = y_n + h\left(\frac{1}{4}k_1 + \frac{3}{4}k_2\right)$$
$$t_{n+1} = t_n + h$$

donde

$$k_1 = g(t_n, y_n)$$
$$k_2 = g(t_n + \frac{2}{3}h, y_n + \frac{2}{3}hk_1)$$

De este modo la ecuación se resuelve de forma iterativa en pasos de tiempo h, dando como resultado una serie de puntos (t_n, y_n) con $n = 0, 1, 2, ..., N$, , los cuales representan la aproximación numérica a la solución exacta $y(t)$, donde N es el número de pasos considerado en la integración.

El objetivo de este ejercicio es escribir un programa en C que implemente el método de Heun para resolver una ecuación diferencial con la forma anterior: $dy/dt = g(t, y)$.

El programa debe cumplir con la siguiente estructura.

En primer lugar debemos crear una función externa a `main` en la que definimos la ecuación diferencial que queremos integrar numéricamente, y que nos devuelve el valor $g(t, y)$ para un cierto valor de t y de y que son pasados como argumentos:

```
double g(double t, double y) {
    double funcion;
    ...
    funcion=...;  // la funcion calcula el valor g(t,y) y lo asigna a la
        variable funcion
    return (funcion);
}
```

El estudiante **deberá proponer** una ecuación diferencial y rellenar el cuerpo de esta función `g(t,y)`.

A continuación tenemos que crear otra función externa a `main` que calcule los términos k_1 y k_2 en cada paso de la integración para unos valores de t y de y. Para ello deberá llamar a la función anterior `g(t,y)`. Mientras las variables t e y son pasadas por valor, las variables k_1 y k_2 deben ser pasadas por referencia. Por lo tanto, la función deberá estar declarada del siguiente modo:

```
void calcula_coeficientesHeun(double t, double y, double h, double *k1,
    double *k2)
```

Finalmente, la función `main` deberá recibir como argumentos la condición inicial t_0 e y_0, el paso del integrador h, el tiempo final de la integración t_N y el nombre del archivo en el que se imprimirán los datos. Estos argumentos deberán ser introducidos por línea de comandos. La función `main` deberá implementar el algoritmo de integración de Heun y exportar al archivo de datos los puntos (t_i, y_i) con $i = 0, \ldots, N$ en forma de dos columnas de datos.

Solución: Ver código `Heun.c`

16. El objetivo de este ejercicio es implementar un método para integrar numéricamente ecuaciones diferenciales ordinarias de primer orden de la forma

$$\frac{dy}{dt} = f(t, y)$$

donde $f(t, y)$ es una función conocida que depende del tiempo t y de la variable y. La solución de esta ecuación es una función $y(t)$ que verifica la ecuación anterior y también la condición inicial $y(t_0) = y_0$.

Este método se conoce como *regla-3/8* y consiste en discretizar la ecuación anterior de la siguiente forma:

$$y_{i+1} = y_i + \frac{1}{8} h \left(k_1 + 3k_2 + 3k_3 + k_4\right)$$
$$t_{i+1} = t_i + h$$

donde

$$
\begin{aligned}
k_1 &= f(t_i, y_i) \\
k_2 &= f\left(t_i + \frac{1}{3}h,\, y_i + \frac{1}{3}hk_1\right) \\
k_3 &= f\left(t_i + \frac{2}{3}h,\, y_i - \frac{1}{3}hk_1 + hk_2\right) \\
k_4 &= f\left(t_i + h,\, y_i + hk_1 - hk_2 + hk_3\right)
\end{aligned}
$$

De este modo la ecuación se resuelve de forma iterativa en pasos de tiempo h, dando como resultado una serie de puntos (t_i, y_i) con $i = 0, 1, 2, ..., N$, los cuales representan la aproximación numérica a la solución exacta $y(t)$, donde N es el número de pasos considerado en la integración.

El objetivo de este ejercicio es escribir un programa en C que implemente el método de la regla-3/8 para resolver una ecuación diferencial con la forma anterior: $dy/dt = f(t, y)$.

El programa debe cumplir con la siguiente estructura.

En primer lugar debemos crear una función externa a `main` en la que definimos la ecuación diferencial que queremos integrar numéricamente, y que nos devuelve el valor $f(t, y)$ para un cierto valor de t y de y que son pasados como argumentos:

```
double f(double t, double y){
    double funcion;
    ...
    funcion=...;  // la funcion calcula el valor f(t,y) y lo asigna a la
        variable funcion
    return (funcion);
}
```

El estudiante **deberá proponer** una ecuación diferencial y rellenar el cuerpo de esta función `f(t,y)`.

A continuación tenemos que crear otra función externa a `main` que calcule los términos k_1, k_2, k_3 y k_4 en cada paso de la integración para unos valores de t y de y. Para ello deberá llamar a la función anterior `f(t,y)`. Mientras las variables t e y son pasadas por valor, las variables k_1, k_2, k_3 y k_4 deben ser pasadas por referencia en forma de vector. Por lo tanto, la función deberá estar declarada del siguiente modo:

```
void calcula_coeficientesRegla3_8(double t, double y, double h, double k[]);
```

Finalmente, la función `main` deberá recibir como argumentos la condición inicial t_0 e y_0, el paso del integrador h, el número de pasos N y el nombre del archivo en el que se imprimirán los datos. Estos argumentos deberán ser introducidos por línea de comandos. La función `main` deberá implementar el algoritmo de integración de la regla-3/8 y exportar al archivo de datos los puntos (t_i, y_i) con $i = 0, \ldots, N$ en forma de dos columnas de datos.

Solución: Ver código `regla_3_8.c`

17. Supongamos que tenemos el siguiente sistema compuesto por dos ecuaciones diferenciales ordinarias de primer orden:

$$
\begin{aligned}
\frac{dx}{dt} &= f(t, x, y) \\
\frac{dy}{dt} &= g(t, x, y)
\end{aligned}
$$

donde $f(t,x,y)$ y $g(t,x,y)$ son funciones conocidas que dependen del tiempo t y de las variables dependientes x e y. La solución de este sistema son las funciones $x(t)$ e $y(t)$ que verifican las ecuaciones anteriores y también las condiciones iniciales $x(t_0) = x_0$ e $y(t_0) = y_0$.

El *método de Euler* es un sencillo método de integración numérica de ecuaciones diferenciales ordinarias que consiste en aproximar la derivada de una función mediante:

$$\frac{dy}{dt} = \frac{y(t+h) - y(t)}{h}$$

De este modo podemos discretizar el sistema anterior en pasos de tiempo h y resolverlo de forma iterativa del siguiente modo:

$$\begin{aligned} x_{i+1} &= x_i + hf(t_i, x_i, y_i) \\ y_{i+1} &= y_i + hg(t_i, x_i, y_i) \\ t_{i+1} &= t_i + h \end{aligned}$$

El resultado de este método es una serie de puntos (t_i, x_i) e (t_i, y_i) con $i = 0, 1, 2, ..., N$, los cuales representan la aproximación numérica a las soluciones exactas $x(t)$ e $y(t)$, donde N es el número de pasos considerado en la integración.

El objetivo de este ejercicio es escribir un programa en C que implemente el método de Euler para resolver el sistema de ecuaciones diferencias ordinarias anterior.

El programa debe cumplir con la siguiente estructura.

En primer lugar crearemos una estructura para el conjunto de valores (t_i, x_i, y_i) y a partir de ella definimos un nuevo tipo de variable denominada `punto`, del siguiente modo:

```
struct valores{
    double t, x, y;
};
typedef struct valores punto;
```

A continuación debemos crear dos funciones externas a `main` en las que definimos las ecuaciones diferenciales que queremos integrar numéricamente, y que nos devuelven el valor de $f(t,x,y)$ y de $g(t,x,y)$ para un cierto valor de nuestra variable `punto`, que es pasada como argumento. Estas funciones deben estar declaradas del siguiente modo:

```
double f(punto p);
double g(punto p);
```

El estudiante **deberá proponer** dos ecuaciones diferenciales y rellenar el cuerpo de estas funciones `f(p)` y `g(p)`.

Finalmente, la función `main` deberá recibir como argumentos las condiciones iniciales t_0, x_0 e y_0, el paso del integrador h, el número de pasos N y el nombre del archivo en el que se imprimirán los datos. Estos argumentos deberán ser introducidos por línea de comandos. La función `main` deberá implementar el algoritmo de integración de Euler utilizando variables `punto` y exportar al archivo de datos los puntos (t_i, x_i, y_i) con $i = 0, \ldots, N$ en forma de tres columnas de datos.

Solución: Ver código `Euler_1.c`

18. Supongamos que tenemos el siguiente sistema compuesto por dos ecuaciones diferenciales ordinarias de primer orden:

$$\frac{dx}{dt} = f(t,x,y)$$
$$\frac{dy}{dt} = g(t,x,y)$$

donde $f(t,x,y)$ y $g(t,x,y)$ son funciones conocidas que dependen del tiempo t y de las variables dependientes x e y. La solución de este sistema son las funciones $x(t)$ e $y(t)$ que verifican las ecuaciones anteriores y también las condiciones iniciales $x(t_0) = x_0$ e $y(t_0) = y_0$.

El *método de Euler* es un sencillo método de integración numérica de ecuaciones diferenciales ordinarias que consiste en aproximar la derivada de una función mediante:

$$\frac{dy}{dt} = \frac{y(t+h) - y(t)}{h}$$

De este modo podemos discretizar el sistema anterior en pasos de tiempo h y resolverlo de forma iterativa del siguiente modo:

$$x_{i+1} = x_i + hf(t_i, x_i, y_i)$$
$$y_{i+1} = y_i + hg(t_i, x_i, y_i)$$
$$t_{i+1} = t_i + h$$

El resultado de este método es una serie de puntos (t_i, x_i) e (t_i, y_i) con $i = 0, 1, 2, ..., N$, los cuales representan la aproximación numérica a las soluciones exactas $x(t)$ e $y(t)$, donde N es el número de pasos considerado en la integración.

El objetivo de este ejercicio es escribir un programa en C que implemente el método de Euler para resolver el sistema de ecuaciones diferencias ordinarias anterior.

El programa debe cumplir con la siguiente estructura.

En primer lugar debemos crear dos funciones externas a `main` en las que definimos las ecuaciones diferenciales que queremos integrar numéricamente, y que calculan el valor de $f(t,x,y)$ y de $g(t,x,y)$ para unos ciertos valores de t, x e y. Mientras estas variables son pasadas como argumentos de la función por valor, las variables `F` e `G` donde se guardarán los valores de f y g deben ser pasadas por referencia. Por lo tanto, estas funciones deberán estar declaradas del siguiente modo:

```
void f(double t, double x, double y, double *F);
void g(double t, double x, double y, double *G);
```

El estudiante **deberá proponer** dos ecuaciones diferenciales y rellenar el cuerpo de estas funciones `f(t,x,y,*F)` y `g(t,x,y,*G)`.

Finalmente, la función `main` deberá recibir como argumentos las condiciones iniciales t_0, x_0 e y_0, el paso del integrador h, el tiempo final de la integración t_N y el nombre del archivo en el que se imprimirán los datos. Estos argumentos deberán ser introducidos por línea de comandos. La función `main` deberá implementar el algoritmo de integración de Euler y exportar al archivo de datos los puntos (t_i, x_i, y_i) con $i = 0, \ldots, N$ en forma de tres columnas de datos.

Solución: Ver código `Euler_2.c`