# Programación

**Asignatura 240205 Programación.**

## Lectura[1]: **Presentación**

Dr. José Ramón González de Mendívil
mendivil@unavarra.es
*Departamento de Estadística, Informática y Matemáticas*
Edificio Las Encinas
**Universidad Pública de Navarra**

**Resumen**

El propósito de esta presentación es proporcionar a los estudiantes unas ideas básicas sobre la estructura de los programas, su funcionamiento, y la terminología que vamos a emplear a lo largo del curso. La aproximación que utilizamos es utilizar programas escritos en pseudocódigo. Un programa consta de dos partes esenciales: una descripción de las acciones que deben ser ejecutadas; y una descripción de los datos que son manipulados por estas acciones. Las acciones o instrucciones se describen mendiante sentencias y los datos mediante declaraciones. Se presentan ejemplos de ejecuciones de programas para determinar de manera precisa la noción de estado de ejecución. Finalmente, introducimos la codificación de programas en el lenguaje C; lenguaje que se utilizará en las prácticas de la asignatura.

## Índice

## 1. Ordenadores y programas

Si preguntamos a cualquier persona sobre lo que hace un *ordenador*[1], recibiremos una gran variedad de respuestas. Un niño probablemente te responda que un ordenador le sirve para jugar y hacer los deberes (algo impensable a principios de los años 70). Una persona joven, quizás responda que lo que le interesa de un ordenador es su capacidad para navegar por la red y acceder a las redes sociales. Un adulto puede responder que un ordenador es muy útil para organizar las finanzas y para gestionar los negocios mediante hojas de cálculo. Habrá un sin fin de respuestas diferentes, pero lo que es realmente destacable, es que es el mismo tipo de máquina, el ordenador, el que permite, mediante el **software** apropiado (**los programas** que tiene disponibles), realizar cualquiera de las tareas indicadas anteriormente.

En la actualidad, disponemos de una amplia variedad de dispositivos que tienen capacidades muy similares a los ordenadores tradicionales de sobremesa. Los dispositivos móviles, por ejemplo, disponen de una gran variedad de programas y con una alta capacidad de conectividad. Ya apenas utilizamos las calculadoras tradicionales. De hecho, nuestros móviles tienen un programa cuya apariencia, al ejecutarse, es la misma que la de una calculadora. Cuando necesitamos realizar un cáculo, por ejemplo, en la solución final de un determinado problema tenemos que calcular el valor de $\sqrt{2}$, símplemente, en el programa calculadora:

tecleamos en la pantalla $[\quad 2]$
elegimos el símbolo que representa la operación raíz cuadrada $\sqrt{2}$
y obtenemos inmedianteamente el resultado en la pantalla $[1,4142135623730]$

A un usuario del programa calculadora de su movil sólo le importa lo que hace (y que funcione bien!!), pero llegado el caso, un Graduado en Ingeniería Informática por la Universidad Pública de Navarra tiene el conocimiento necesario para crear toda la funcionalidad que requiere dicho programa, desarrollarlo, instalarlo y ponerlo en funcionamiento. Por el momento, vamos a conformarnos con entender un procedimiento que nos permita calcular cualquier raíz cuadrada.

## 2. Problemas y soluciones

**Problema**: En el apartado anterior se nos pide calcular $\sqrt{2}$. Sabemos que $\sqrt{2}$ no es un número racional[2] y por lo tanto, tiene un número ilimitado de cifras decimales. Así que sólo podemos calcular dicha raíz por medio de un procedimiento aproximado. En las escuelas se estudia un método de extracción de raíces cuadradas. En esta sección vamos a presentar un método que ya se utilizaba mucho antes de nuestra era en Babilonia. Lo empleaba también Herón, un matemático de Alejandría. Este método fue abandonado pero se puede recurrir precisamente a él cuando queremos calcular raices cuadradas en las calculadoras.

Vamos a considerar un número positivo $N$, $N > 0$, del cual, queremos extraer la raíz cuadrada $\sqrt{N}$. En realidad, queremos resolver el problema dado por la ecuación $x = \sqrt{N}$ siendo $x$

**Solución**: Como no podemos conocer en general de forma exácta el número $\sqrt{N}$, entonces vamos probando con diferentes números, $X_1$, $X_2$,...,$X_k$. Resolver el problema de esta forma es hacerlo mediante **aproximaciones sucesivas**. Vamos generando dicha secuencia de soluciones aproximadas y podemos elegir, por ejemplo, la que quede más cerca de 0 en el polinomio $p(x)$ anterior. La idea del método es que, además, cada nueva solución esté más cera del resultado real, es decir, $X_2$ está más cerca de $\sqrt{N}$ que su antecesor $X_1$ y así sucesivamente.

Supongamos que hemos calculado $X_1$, $X_2$,..., $X_k$, $k$-aproximaciones, a $\sqrt{N}$, todas positivas. Dada la última aproximación vemos que

$$\sqrt{N} = \sqrt{X_k \cdot \frac{N}{X_k}} \tag{1}$$

Es decir, $\sqrt{N}$ es el promedio geométrico de $X_k$ y $\dfrac{N}{X_k}$. El promedio geométrico de dos números está acotado por el promedio aritmético, entonces (y aquí está es la **clave del método**) como nuevo valor aproximado de este promedio geométrico podemos tomar el promedio aritmético de los números $X_k$ y $\dfrac{N}{X_k}$, es decir, hagamos que

$$X_{k+1} = \frac{1}{2}(X_k + \frac{N}{X_k}) = \frac{X_k^2 + N}{2X_k} \tag{2}$$

El método se completa dando una primera solución aproximada, $X_1$, por ejemplo, $X_1 = N$ es una opción válida para este método. Veamos el procedimiento con $\sqrt{2}$.

| Aproximación | valor |
|---|---|
| $X_1$ | $2$ |
| $X_2$ | $\frac{1}{2}(2 + \frac{2}{2}) = 1{,}5$ |
| $X_3$ | $\frac{1}{2}(1{,}5 + \frac{2}{1{,}5}) = 1{,}4166$ |
| $X_4$ | $\frac{1}{2}(1{,}4166 + \frac{2}{1{,}4166}) = 1{,}4142$ |

Aparentemente, el método anterior es *convergente*. Para comprobarlo, vamos a calcular los *errores absolutos*, $\alpha_k = \sqrt{N} - X_k$ y $\alpha_{k+1} = \sqrt{N} - X_{k+1}$, de dos aproximaciones consecutivas.

$$\alpha_{k+1} = \sqrt{N} - X_{k+1} = \sqrt{N} - \frac{X_k^2 + N}{2X_k} = -\frac{X_k^2 - 2X_k\sqrt{N} + N}{2X_k}$$

$$\text{como, } X_k^2 - 2X_k\sqrt{N} + N = (X_k - \sqrt{N})^2 = \alpha_k^2$$

se concluye,

$$\alpha_k^2 \qquad (3)$$



por esto, $|\alpha_{k+1}| = |\frac{1}{2} - \frac{}{2X_k}| \cdot |\alpha_k|$

Teniendo, en cuenta que $X_k$ es positivo y $X_k > \sqrt{N}$ (asegurado para $k \geq 2$), entonces $0 < \dfrac{\sqrt{N}}{X_k} < 1$, lo cual nos lleva a que

$$|\alpha_{k+1}| < \frac{1}{2}|\alpha_k| \tag{4}$$

En cada nuevo paso de aproximación, el error absoluto va disminuyendo al menos el doble, y se va aproximando a 0 con mucha rapidez. Esto implica que los números $X_k$ tienden a $\sqrt{N}$ cuando $k$ va creciendo. Además, el método permite comenzar con cualquier valor para $X_1$, nosotros hemos elegido el propio número $N$ pero puedes elegir cualquier número positivo.

Para que veas lo que significa dividir por dos el error absoluto, imagina que para calcular $\sqrt{2}$ comienzas con la aproximación $X_1 = 10^6$. El error $\alpha_1 \approx 10^6$ es aproximadamente de un millón. Cuando has realizado 40 aproximaciones, $|\alpha_{40}| < 10^{-6}$, es decir, te has aproximado a la solución en menos de una millonésima. Finalmente, si una aproximación $X_k$ cumple que $X_k^2 - N$ es una cantidad muy próxima a 0 se finaliza el cálculo.

## 3. Programas

El procedimiento de cálculo de la raíz cuadrada lo vamos a presentar utilizando un pseudocódigo apropiado, bastante más simple que un lenguaje de programación moderno. Los programas escritos en pseudocódigo también se denominan **algoritmos**[3].

En general, un **programa** es un texto escrito que especifica, sin ambigüedad, la secuencia de acciones que hay que llevar a cabo para realizar un determinado procedimiento de cálculo (en el sentido más ámplio posible del término cálculo) previamente definido. El programa es un texto creado por un programador de manera que las acciones que indica el texto pueden ser ejecutadas por el mismo programador o, actualmente, mediante un ordenador.

Una norma que debemos tener en cuenta a la hora de escribir los programas es que sólo podemos emplear 'nombres' de elementos que previamente han sido declarados explícitamente en el texto del programa, o 'palabras reservadas' que forman parte del pseudocódigo. En este último caso, procuramos escribirlas en negrita para no confundir.

### 3.1. Variables y la instrucción de asignación

**Nota**: *Algunas nociones por ser fundamentales, y no estar formadas a partir de otras, hay que presentarlas directamente a través de ejemplos. Se requiere que el estudiante adquiera estas nociones de forma intuitiva pero precisa.* (del maestro Niklaus)

Fundamental en el diseño de programas es la **declaración de variables**. Por **variable** queremos categorizar a aquellos objetos en los programas que tienen un (1) 'nombre' (ideado por el programador); (2) pertenencen a un tipo de dato; y (3) que en la ejecución de las acciones del programa pueden cambiar su *valor*.

En otros términos, una variable se puede ver como un 'contenedor' de datos. En la decla-

podemos utilizar la nomenclatura **var**...**fvar** para agrupar una declaración de variables:

```
1: var
2:     velocidad, posicion: real          ▷ 'esto es un comentario': unidades en m/s y m
3:     tiempo: entero                                                ▷ unidad en s
4: fvar
```

En Programación interpretamos una variable como un contenedor de datos, de manera que un dibujo razonable de este hecho es, por ejemplo, □ *tiempo* [.....]□. El valor de la variable lo dejamos en el contenedor, por ejemplo, □ *tiempo* [341]□. Si este es el caso, entonces el valor de *tiempo* (en este momento) es el número entero 341. Los número reales no encajan con los números enteros. Cualquier intento de encajar 341,34 como valor de *tiempo* nos daría un error[4] (siendo más precisos, lo consideraríamos un error).

Una cuestión importante es cómo modificamos los valores de las variables. La respuesta es que sólo podemos modificar los valores de las variables mediante una instrucción especial que denominamos **instrucción de asignación** (acción de asignación, o sentencia de asignación). Considera que tu programa debe incrementar la velocidad □ *velocidad* [124,53]□ en 15,5 m/s, la instrucción de asignación correspondiente sería:

$velocidad := velocidad + 15{,}5$

El símbolo := lo reservamos para indicar que es una instrucción de asignación[5] y no debe confundirse con el símbolo que empleamos habitualmente para indicar la igualdad =.

Cómo funciona la instrucción de asignación anterior, o cómo se ejecuta:

1. Supongamos que justo antes de ejecutar la instrucción □ *velocidad* [124,53]□, es decir, su valor es 124,53
2. Se evalua la expresión aritmética que se encuentra en la parte derecha de la asignación mediante sustitución de los nombres de las variables por sus valores, en este caso, en $velocidad + 15{,}5$ se sustituye por $124{,}53 + 15{,}5$
3. Se calcula el valor de la expresión, $124{,}53 + 15{,}5$, esto es, 140,03, y se deposita en el contenedor de la variable que se encuentra en la parte izquierda de la asignación
4. □ *velocidad* [140,03]□

Observa, que el valor anterior de *velocidad* se ha perdido, ya que las variables no tienen 'memoria' y su valor es el último que se ha escrito en su contenedor. Observa tambien, que la ejecución de la instrucción produce un **cambio de estado**.

Esto es común a los lenguajes imperativos: si le decimos a Juan que 'pulse el interruptor de un poco más largo.

1: $\square$ *posicion* [120,5]$\square$ $\square$ *velocidad* [10,1]$\square$ $\square$ *tiempo* [3]$\square$
2: *posicion* := *posicion* + *velocidad* ∗ *tiempo*
3: $\square$ *posicion* [150,8]$\square$ $\square$ *velocidad* [10,1]$\square$ $\square$ *tiempo* [3]$\square$

**Nota**: El entorno de ejecución de las instrucciones que aparecen en los programas (su habitación) lo forman las variables que han sido declaradas. En un determinado momento después de haber ejecutado una o varias intrucciones, el **estado de la ejecución del programa** viene representado por los valores que toman todas las variables en ese momento..., y algo más, un indicador de cuál es la siguiente instrucción que se tiene que ejecutar. En los algoritmos, al ser el programador el que ejecuta sus instrucciones, la cuenta de cuál es la siguiente instrucción a ejecutar la tiene que llevar él mismo, por ejemplo, apuntando al texto con un lápiz. Los cambios de valores de las variables las tendrá que ir reflejando en algún lugar, por ejemplo, escribiendo en un hoja una tabla con el nombre de las variables y los valores que van tomando.

En el ejemplo anterior, se observa también un aspecto importante de la instrucción de asignación y es que sólo cambia su valor la variable asignada y ninguna otra. Se dice entonces que la instrucción de asignación no tiene **efectos colaterales**.

Finalmente, para tener una notación más compacta de la representación de los estados de ejecución emplearemos la siguiente notación basada en conjuntos y predicados[6]. El ejemplo anterior lo escribimos de la siguiente manera:

1: $\{posicion = 120{,}5 \wedge velocidad = 10{,}1 \wedge tiempo = 3\}$ ▷ estado de partida $\sigma$
2: *posicion* := *posicion* + *velocidad* ∗ *tiempo* ▷ ejecutar la instrucción de asignación
3: $\{posicion = 150{,}6 \wedge velocidad = 10{,}1 \wedge tiempo = 3\}$ ▷ estado alcanzado $\sigma'$

**Nota**: Un comentario final. Como el **estado de ejecución** viene determinado por los valores de las variables y éstas sólo pueden cambiar por instrucciones de asignación entonces también lo llamamos **estado de asignación de variables**.

### 3.2. Algoritmo para las raíces cuadradas

**Nota**: La explicación completa de estos programas se presenta en clase. Tened en cuenta que esta asignatura no presupone ningún conocimiento previo en lenguajes de programación, ni en la construcción de programas. Se explica la instrucción de asignación, la composición secuencial de sentencias, y el funcionamiento básico de los dos bucles: **mientras**...**hacer**...**fmientras**.

En el caso que nos ocupa, vemos que para escribir el programa de la raíz cuadrada necesitamos:

1. una variable para el número sobre el que queremos extraer su raiz cuadrada. Vamos a considerar números mayores o iguales a 1.



[6] Esto lo trataremos más adelante, por el momento, sólo es una cambio de notación

4. el programa tiene que parar en algún momento: cuando hayamos calculado una aproximación bastante cercana al valor real de la raíz cuadrada del número correspondiente, paramos y devolvemos el resultado. En principio podemos pensar en disponer de una variable con valores entre 0 y 1, por ejemplo, 0,000001 si queremos tener una aproximación en seis decimales.

El tipo de esas variables será el tipo *real* ya que deben contener números con decimales.

Es una cuestión de organización del programador con respecto al procedimiento a realizar, ajustar la cantidad de variables que necesita y los nombres que va a utilizar. Un mismo procedimiento puede realizarse de formas ligeramente diferentes.

```
1:  algoritmo calculo_raiz_v1
2:      var
3:          num, raiz, limite: real                    ▷ Declaración de variables
4:      fvar
5:      Requisito: El número tiene que ser > 1.
6:      Requisito: El limite tiene que ser un número entre (0, 1)
7:      Leer(num);                                     ▷ introducir el número
8:      Leer(limite);                                  ▷ introducir el límite
9:      raiz := num;
10:     mientras raiz * raiz − num ≥ limite hacer
11:         raiz := (raiz * raiz + num) / (2 * raiz)
12:     fmientras;
13:     Escribir(raiz)                                 ▷ presentar el resultado
14: falgoritmo
```

Observa que en la ejecución de este programa se calcula dos veces el producto $raiz * raiz$ en cada paso del bucle. Puedes mejorar el programa evitando hacer dos veces ese cálculo si te aprovechas de que la aproximación es siempre por exceso desde el principio, y que $X_k - X_{k+1} > 0$, con lo que $X_k - X_{k+1} = \frac{1}{2}(X_k - \frac{N}{X_k})$. Terminas cuando, $X_k - X_{k+1} < limite$ o lo que es lo mismo cuando, $X_k - \frac{N}{X_k} < 2 \cdot limite$. En el algoritmo $X_k$ son los valores que toma la variable *raiz*, y $\frac{N}{X_k}$ son los valores que va tomando la variable *med*. Puedes seguir definiendo el valor del *limite* como un número entre 0 y 1 para la condición de terminación.

```
1:  algoritmo calculo_raiz_v2
2:      var
3:          num, raiz, med, limite: real                    ▷ Declaración de variables
4:      fvar
5:      Requisito: El número tiene que ser > 1.
6:      Requisito: El limite tiene que ser un número entre (0, 1)
7:      Leer(num);                                          ▷ introducir el número
8:      Leer(limite);                                       ▷ introducir el límite
9:      raiz := num;
10:     med := 1;                                           ▷ la primera vez num/raiz = 1
11:     mientras raiz − med ≥ limite hacer
12:         raiz := (raiz + med) / 2;
13:         med := num/raiz
14:     fmientras;
15:     Escribir(raiz)                                      ▷ presentar el resultado
16: falgoritmo
```

A continuación mostramos un ejemplo de ejecución del algoritmo en su versión 2. En la descripción del estado de ejecución, demás del estado de asignación de las variables, tambien indicamos la siguiente instrucción que se tiene que ejecutar mediante 'PC linea...'. Las letras PC vienen de los términos en ingles 'Program Counter' (Contador de Programa). Cada instrucción 'i-ésima' a ejecutar se nombra por el símbolo $\pi_i$ y cada estado 'i-ésimo' por $\sigma_i$.

| | Ejecutar | la Sentencia | Estado de Ejecución |
|---|---|---|---|
| | ——— | ——— | valores de variables sin definir<br>PC línea 7. |
| | ejecutar 7. | leer de dispositivo | $\{\ num = 2\ \}$<br>PC línea 8. |
| | ejecutar 8. | leer de dispositivo | $\{\ num = 2, limite = 0{,}00005\ \}\ \sigma_0$<br>PC línea 9. |
| $\pi_1$ | ejecutar 9 | $raiz := num$ | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 2\ \}\ \sigma_1$<br>PC línea 10. |
| $\pi_2$ | ejecutar 10 | $med := 1$ | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 2, med = 1\ \}\ \sigma_2$<br>PC línea 11. |
| $\pi_3$ | ejecutar 11 | $raiz - med \geq limite$<br>se evalua a cierto: PC a 12. | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 2, med = 1\ \}\ \sigma_3$<br>PC línea 12. |
| $\pi_4$ | ejecutar 12 | $raiz := (raiz + med)\ /\ 2$ | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 1{,}5, med = 1\ \}\ \sigma_4$<br>PC línea 13. |
| $\pi_5$ | ejecutar 13 | $med := num/raiz$ | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 1{,}5, med = 1{,}333333\ \}\ \sigma_5$<br>PC línea 14. |
| $\pi_6$ | ejecutar 14 | PC a la línea 11. | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 1{,}5, med = 1{,}333333\ \}\ \sigma_6$<br>PC línea 11. |
| $\pi_7$ | ejecutar 11 | $raiz - med \geq limite$<br>se evalua a cierto: PC a 12. | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 1{,}5, med = 1{,}333333\ \}\ \sigma_7$<br>PC línea 12. |
| $\pi_8$ | ejecutar 12 | $raiz := (raiz + med)\ /\ 2$ | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 1{,}416666, med = 1{,}333333\ \}\ \sigma_8$<br>PC línea 13. |
| $\pi_9$ | ejecutar 13 | $med := num/raiz$ | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 1{,}416666, med = 1{,}411765\ \}\ \sigma_9$<br>PC línea 14. |
| $\pi_{10}$ | ejecutar 14 | PC a la línea 11. | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 1{,}416666, med = 1{,}411765\ \}\ \sigma_{10}$<br>PC línea 11. |
| $\pi_{11}$ | ejecutar 11 | $raiz - med \geq limite$<br>se evalua a cierto: PC a 12. | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 1{,}416666, med = 1{,}411765\ \}\ \sigma_{11}$<br>PC línea 12. |
| $\pi_{12}$ | ejecutar 12 | $raiz := (raiz + med)\ /\ 2$ | $\{\ num = 2, limite = 0{,}00005$<br>$raiz = 1{,}41421, med = 1{,}411765\ \}\ \sigma_{12}$<br>PC línea 13. |
| | ejecutar 15 | resultado $1{,}414215$ | estado final $\sigma_{15}$ |

Partiendo del estado incial $\sigma_0$ donde se recogen los valores inciales para las variables *num* y *limite*, el algoritmo genera la secuencia de ejecución,

$$\sigma_0\ \pi_1\ \sigma_1\ \pi_2\ \sigma_2\ ....\ \sigma_{14}\ \pi_{14}\ \sigma_{15}$$

La ejecución termina en el estado $\sigma_{15}$ y se muestra el resultado de la variable *raiz*. Observa que el contador de programa PC siempre apunta a la siguiente instrucción que se tiene que ejecutar. Por otra parte, la composición secuencial ; hace que el PC se actualice automáticamente en la siguiente instrucción. En cambio, la composición iterativa puede producir una acutalización del PC muy diferente si la condición es cierta o falsa. Esto lo observas claramente en la tabla. En el fondo, lo que te muestra la tabla es una visión de lo que sucede en la realidad cuando ejecutas el programa en un ordenador. Un ejemplo de tal programa para calcular la raíz cuadrada codificado en el lenguaje C se presenta en la siguiente sección.

## 4. Codificación y lenguajes de programación

El proceso de la Programación, o las tareas que debe realizar un programador, tiene cuatro pasos bien diferenciados: (1) **Especificar** los requisitos y resultados del problema que se trata de solucionar mediante un programa informático; (2) **Diseñar** una solución correcta, eficiente y general en forma de diagramas de flujo de operaciones, o programas en un pseudocódigo; (3) **Codificar** la solución mediante un lenguaje de programación; y (4) **Ejecutar, verificar, medir el rendimiento** del programa resultante, comprobando su funcionamiento con 'casos de prueba' significativos.

Se debe dedicar bastante tiempo a realizar un buen diseño de la solución puesto que los errores cometidos en el diseño se pasan a las etapas posteriores. De nada nos sirve un programa que puede contener errores en su funcionamiento. Los errores que se comenten en esta etapa suponen pérdidas de tiempo (y económicas) muy importantes. Si la solución, en forma de algoritmos, diagramas UML, u otros elementos descriptivos válidos, es correcta, entonces podemos codificar la solución en un lenguaje de programación. Los errores que se comenten en la codificación se pueden corregir, bien porque son errores sintácticos, o bien porque son errores, que algunas de las pruebas de ejecución se pueden detectar. Las pruebas de ejecución siguen siendo importantes y hay que planificarlas muy bien para encontrar los límites de aplicabilidad del programa desarrollado. Al nivel de Programación, los programas son bastante cortos, no más largos que un par de caras de un folio. La longitud de un programa no mide la dificultad del problema. Los problemas que abordamos en Programación aparecen en innumerables ocasiones en el desarrollo de programas, y son utilizados para que el estudiante conozca las técnicas apropiadas para su diseño.

El siguiente texto se corresponde con la codificación en lenguaje C del algoritmo calculo_raiz_v2 presentado en la sección anterior:

```
// Programa: calculo_raiz_v2
```



```
{
```

```
    float num, raiz, med, limite;
    char res;
    printf("Programa: calculo_raiz_v3\n");
    printf("Autor: Iñigo Ezcurdia Aguirre\n");
    printf("Fecha: 22 Abril 2020\n");
    printf("Descripcion: Este programa calcula la aproximacion de\n");
    printf("la raiz cuadrada de un numero real mayor que uno\n");
    do
    {
        printf("Introduzca un valor real mayor que 1: ");
        scanf("%f", &num);
        printf("Introduzca un valor real entre (0,1), limite de margen de error: ");
        scanf("%f", &limite);
        raiz = num;
        med = 1;
        while( (raiz - med) >= limite )
        {
            raiz = ( raiz + med ) / 2;
            med = num / raiz;
        }
        printf("La raiz cuadrada aproximada de %f es %f\n", num, raiz);
        printf("Desea continuar? S/N: ");
        scanf(" %c", &res);
    }while( res == 's' || res == 'S' );
    return 0;
}
```

**Nota**: Los estudiantes pueden comprobar estos programas en C. Lo hemos comprobado en,

`https://www.onlinegdb.com/online_c_compiler`

Hay otro aspecto que es importante comentar. El desarrollo de la Informática tal y como la conocemos actualmente no hubiese sido posible sin los mecanismos adecuados para poder reutilizar programas ya desarrollados como parte de otros programas mas complejos. Una herramienta muy útil del (diseñador) programador es la 'abstracción', y la forma más concreta de esta abstracción en Programación se plasma mediante el mecanismo de las **acciones** y **funciones**. Considera de nuevo el programa de la raíz cuadrada desarrollado anteriormente. La cuestión es cómo lo utilizamos como parte de otro programa. Observa, que en el programa anterior tiene, por decirlo así, dos entradas, el número, y el límite para el error, y una salida, el resultado de la raíz cuadrada. Vamos a considerar que fijamos el error para todos los casos como una constante dada. Pues bien, con esa misma idea encapsulamos el código mediante una función que justamente, tiene como parámetros (formales) de entrada el número del cuál queremos calcular

```
1:  algoritmo lista_de_raíces
2:
3:                          ▷ Este algoritmo calcula las raíces cuadradas de los números del 1 al N
4:                                  ▷ y los guarda en una tabla de N posiciones
5:
6:      constante
7:          LIMITE 0,000001                          ▷ Límite para el error de los cálculos
8:          N 100                                                ▷ tamaño de la tabla
9:      fconstante
10:
11:     tipo
12:         Tabla1N = tabla [1..N] de enteros
13:     ftipo
14:
15:     funcion calculo_raiz (num: real) dev raiz: real
16:         var
17:             med: real
18:         fvar
19:         raiz := num;
20:         med := 1;
21:         mientras raiz − med ≥ LIMITE hacer
22:             raiz := (raiz + med) / 2;
23:             med := num/raiz
24:         fmientras;
25:         dev (raiz)
26:
27:     ffuncion
28:
29:     ▷ ——————————————————- programa principal ——————————————————-
30:     var
31:         lista: Tabla1N
32:         k: 1..N
33:     fvar
34:
35:     para k := 1 hasta N hacer
36:         lista[k] := calcula_raiz(k)
37:     fpara;
38:     escribir(lista)                                  ▷ presenta la lista de raíces
39: falgoritmo
```

Mostramos también el algoritmo anterior codificado en C.

```
// Programa: lista_de_raices
// Autor: Iñigo Ezcurdia Aguirre
// Fecha: 27 Abril 2020
// Descripción: Este programa imprime por pantalla un listado
// de las raices cuadradas de los numeros del 1 al N

#include <stdio.h>

#define LIMITE 0.000001 //Límite para el error de los cálculos
#define N 100 // tamaño de la tabla

typedef float Tabla1N[N+1];
void escribir(Tabla1N tabla);
float calcula_raiz(float num);

int main (void)
{
    Tabla1N lista;
    int k;
    char res;
    printf("Programa: lista_de_raices\n");
    printf("Autor: Iñigo Ezcurdia Aguirre\n");
    printf("Fecha: 27 Abril 2020\n");
    printf("Descripción: Este programa imprime por pantalla un
    listado de las raices cuadradas de los numeros del 1 al N.\n");
    //
    // programa principal
    do
    {
        for( k = 1; k <= N; k = k+1 )
            lista[k] = calcula_raiz(k);
        escribir(lista); // presenta la lista de raíces
        printf("\nDesea continuar? S/N: ");
        scanf(" %c", &res);
    }while( res == 's' || res == 'S' );
    return 0;
}

void escribir(Tabla1N tabla)
{
    int i;
```

```
    while ( (raiz - med) >= LIMITE )
    {
        raiz = (raiz + med) / 2;
        med = num / raiz;
    }
    return raiz;
}
```

**Nota**: Si observas los programas en C, estos tienen una gran cantidad de detalles que no son necesarios en el pseudocódigo que empleamos para la construcción de las soluciones. Ese es uno de los motivos para utilizar pseudocódigos. Otro motivo, es la notación que utiliza C para muchas cosas. No es natural. Procuramos que la notación del pseudocódigo sea mucho más fácil. ¿Por qué usar dos lenguajes diferentes?: bueno, antes de hacer una gran obra, un buen arquitecto plantea unos buenos esquemas, bocetos del diseño, y buenos cáculos... Pero la realidad de la programación es árida en algunos aspectos, no todo encaja a la perfeccción: entre el pseudocódigo dado en el algoritmo lista_de_raices y su programa en C hay una sutíl diferencia en la declaración de la tabla. Los estudiantes pueden intentar encontrar una explicación a tal diferencia y modificarla si piensan que es oportuno hacerlo.

**Ejercicio 1** *Realiza los siguientes ejercicios:*

1. *En los algoritmos de la raíz cuadrada hemos puesto el requisito de que el número incial sea mayor o igual a uno. Pero este requisito parece que no se presenta en el procedimiento de aproximaciones propuesto. Estudia que le sucede al algoritmo versión 2, en el caso de introducir un número entre* 0 *y* 1*. Si se produce algún error estudia cómo se puede corregir.*

2. *Prueba los programas en C anteriores en alguna de las webs on-line sugeridas. También puedes pasarte por algún laboratorio de informática para hacerlo. Consulta a los profesores de prácticas.*

## 5. Algoritmo de Euclides: cálculo del máximo común divisor

Dados dos números enteros $a$ y $b$, si el cociente de la división de $a$ por $b$ es entero, denotado como $q$, se tiene que $a = bq$. Decimos entonces que $b$ es *divisor* de $a$ y que $a$ es *múltiplo* de $b$. En lo que sigue consideramos sólo divisores positivos. El teorema de la división entera indica que *todo entero $a$ se expresa de un modo único mediante un entero positivo $b$ en la forma*

$$a = bq + r \text{ con } 0 \leq r < b \tag{5}$$

Todos los lenguajes de programación disponen de dos operaciones **div** y **mod** que permiten calcular el cociente y el resto de la división entera. Así, siguiendo (5), el cociente entero entre $a$ y $b$ es $q = a$ div $b$; y su resto es $r = a$ mod $b$. Estas operaciones producen error cuando el divisor

Para encontrar el máximo común divisor de dos números se emplea el **algoritmo de Euclides**. Este algoritmo consiste en lo siguiente. Sean $a$ y $b$ dos enteros positivos. Según el teorema de la división entera podemos hallar la siguiente serie de igualdades:

$$\begin{array}{ll} a = b\,q_1 + r_2 & 0 < r_2 < b \\ b = r_2\,q_2 + r_3 & 0 < r_3 < r_2 \\ r_2 = r_3\,q_3 + r_4 & 0 < r_4 < r_3 \\ ..... & ..... \\ r_{n-2} = r_{n-1}\,q_{n-1} + r_n & 0 < r_n < r_{n-1} \\ r_{n-1} = r_n\,q_n & r_{n+1} = 0 \end{array}$$

La serie anterior termina cuando encontramos $r_{n+1} = 0$. Pero efectivamente, la serie anterior termina puesto que la sucesión decreciente $b, r_2, r_3, ...$ no puede contener más de $b$ números positivos. Examinando las igualdades anteriores y por la propiedad indicada en el párrafo anterior, los divisiores comunes de $a$ y $b$, $DC(a,b)$, son iguales a los divisiores comunes de $b$ y $r_2$, $DC(a,b) = DC(b,r_2)$ y así sucesivamente hasta llegar a que son iguales a los divisores del número $r_n$[7],

$$DC(a,b) = DC(b,r_2) = DC(r_2,r_3) = ... = DC(r_{n-2},r_{n-1}) = DC(r_{n-1},r_n) = DC(r_n,0) \quad (6)$$

Por lo tanto,

$$mcd(a,b) = mcd(b,r_2) = mcd(r_2,r_3) = ... = mcd(r_{n-2},r_{n-1}) = mcd(r_{n-1},r_n) = r_n \quad (7)$$

Para programar el algoritmo de Euclides vamos a utilizar cuatro variables enteras, dos de ellas para recoger los números y otras dos variables para los cálculos de la serie anterior. Si observas la serie, en cada paso de una igualdad a la siguiente, el divisor de la igualdad anterior se convierte en el nuevo dividendo y el resto anterior pasa a ser el nuevo divisor.

```
1:  algoritmo mcd
2:      var
3:          a, b, num, div: entero
4:      fvar
5:      Requisito: Los números a y b tiene que ser enteros positivos.
6:      Leer(a);                          ▷ introducir el primer número
7:      Leer(b);                          ▷ introducir el segundo número
8:      num := a;
9:      div := b;
10:     mientras (num mod div) ≠ 0 hacer
11:         ⟨num, div⟩ := ⟨div, num mod div⟩
12:     fmientras;
13:     Escribir(div)                     ▷ presentar el resultado
14: falgoritmo
```



[7] Los divisores de $r_n$, son los divisores comunes con el 0, ya que 0 se divide por cualquier número positivo.

```
1: algoritmo mcd_v2
2:     var
3:         a, b, num, div, aux: entero
4:     fvar
5:     Requisito: Los números a y b tiene que ser enteros positivos.
6:     Leer(a);                                  ▷ introducir el primer número
7:     Leer(b);                                  ▷ introducir el segundo número
8:     num := a;
9:     div := b;
10:    mientras (num mod div) ≠ 0 hacer
11:        aux:= num;
12:        num := div;
13:        div := aux mod div
14:    fmientras;
15:    Escribir(div)                             ▷ presentar el resultado
16: falgoritmo
```

**Ejercicio 2** *Realiza los siguientes ejercicios:*

1. *Demuestra el teorema de la división entera dado en esta sección.*

2. *Ejecuta el algoritmo de Euclides (mcd) para a con valor* 28 *y b con valor* 144*. Observa que no es necesario que el valor de a deba ser mayor que b al principio. Indica los estados de ejecución que se obtienen al ejecutar el programa con los números anteriores.*

3. *Observa que el programa mcd mantiene en todo momento los valores originales de los números dados en las variables a y b ya que no hay ninguna sentencia de asignación que afecte a estas variables. El número de variables en mcd y en mcd_v1 son cuatro y cinco respectivamente. El número mínimo de variables que se necesita es 2 en el caso de usar una asignación múltiple y 3 en el caso de usar asignaciones simples. Escribe dichos programas.*

4. *Dada una serie de números positivos, A1, A2,...,AN. Indica y justifica un procedimiento correcto para calcular el máximo común divisor de todos ellos.*

5. *Se utiliza una variable t:* **tabla** *[1..N]* **de** *entero, para almacenar los números anteriores donde N es una constante conocida. Intenta escribir un programa en pseudocódigo para calcular el máximo común divisor de todos los números almacenados en la tabla.*

## Referencias

[1] Niklaus Wirth. Introducción a la Programación Sistemática. Editorial El Ateneo. Buenos Aires. 1982.