

Introducción a la lógica de programación

Grupo de Dinámica de Flujos Ambientales
Universidad de Granada



Índice

Índice	2
Lista de Figuras	2
Lista de Tablas y Algoritmos	2
Algunos Fundamentos de Computadores	4
Breve historia.....	5
La era mecánica (1623-1945).....	5
La primera generación de ordenadores electromecánicos (1937-1953): Válvulas de vacío.	6
Segunda Generación (1954-1962): Transistor.....	6
Tercera generación (1963-1972): Integración de transistores en superficies planas: circuitos integrados.....	7
Cuarta generación (1972-1984). Desarrollo de circuitos integrados.....	7
Quinta generación (1984-1990).....	7
La actualidad	8
Sistemas numéricos y errores	8
Representación en coma flotante.....	10
Lógica de programación y estructuras de datos.....	15
Programación estructurada	15
Algoritmos.....	16
Diagramas de Flujo.....	17
Pseudocódigo.....	20
Elementos de programación	21
Inicio, Fin	21
Tipos de datos: declaración y asignación de variables	22
Estructuras de control de flujo.....	23
Bibliografía.....	32

Lista de Figuras

Figura 1. Representación binaria de enteros.....	9
Figura 2. Representación en computador de un número fraccionario en notación punto flotante.....	9
Figura 3. Representación en coma flotante en simple precisión para un computador con 32 bits de longitud de palabra.	11
Figura 4. Conceptos de las tres clases de bloques de programación.	16
Figura 5. (Izquierda) Diagrama de flujo de un algoritmo que calcula la suma de los primeros 50 números enteros. Implementado mediante un ciclo <i>while</i> . (Derecha) Diagrama de flujo de un algoritmo que calcula el producto escalar de los vectores A y B . Implementado mediante un ciclo <i>for</i>	19
Figura 6. (Izquierda) Pseudocódigo asociado al algoritmo cuya función es sumar los 50 primeros números enteros. (Derecha) Pseudocódigo asociado al algoritmo cuya función es realizar el producto escalar de los vectores A y B	20

Lista de Tablas y Algoritmos

Tabla 1. Símbolos estándar empleados en diagramas de flujo.....	18
Tabla 2. Estructuras en pseudocódigo.....	21
Tabla 3. Inicio/fin en diagramas de flujo, pseudocódigo, C y Matlab.	21
Tabla 4. Declaración y asignación de variables.	22

Tabla 5. Estructuras de selección simple.....	24
Tabla 6. Estructuras de selección doble.....	25
Tabla 7. Estructura de selección anidada.	26
Tabla 8. Estructura de selección múltiple.....	27
Tabla 9. Bucle <i>for</i>	29
Tabla 10. Bucle <i>while</i>	30
Tabla 11. Bucle <i>do-while</i>	32

Para este capítulo se han seguido las siguientes referencias [Antonakos y Jr., 1997; Burden y Faires, 1985; Conte y Boor, 1980; Gerald y Wheatley, 2000; Hefferson, 2006; Kernighan y Ritchie, 1988; Kincaid y Cheney, 1990; Lang, 1987; Molina, 1996; O'connor, 1993].

Algunos Fundamentos de Computadores

Una bien conocida caracterización de Física Computacional (FC) y de los métodos numéricos en general, fue dada por K. Wilson en 1986. Según él, un problema típico de Física Computacional (*i*) tiene una formulación matemática precisa, (*ii*) es intratable por métodos analíticos, (*iii*) tiene objetivos relevantes en ciencia y (*iv*) requiere un conocimiento profundo del problema.

La forma de hacer ciencia usando ordenadores es inherentemente multidisciplinar. Así, por ejemplo, es casi preceptivo tener excelentes conocimientos en matemáticas aplicadas, técnicas de uso de ordenadores y conocimientos de arquitectura de los mismos. Además de conocer en los fundamentos físicos del problema. En efecto, a diferencia de un experto informático, el objetivo intelectual último que nos proponemos no es la comprensión del funcionamiento de la máquina sino su uso para resolver y comprender problemas científicos concretos. Algunos problemas típicos: la comprensión de la estructura electrónica de materiales, el comportamiento de un fluido en un régimen turbulento, el modelado del clima, la obtención de las claves genéticas del genoma humano,...

Las cuestiones que no pueden ser abordadas por métodos tradicionales suelen ser de tal complejidad que se han de desarrollar algoritmos numéricos específicos para darles respuesta por medio de técnicas computacionales. Ya en 1952 Hartree había reconocido la importancia de los desarrollos algorítmicos:

With the development of new kinds of equipment of greater capacity, and particularly of greater speed, it is almost certain that new methods will have to be developed in order to make the fullest use of this new equipment. It is necessary not only to design machines for the mathematics, but also to develop a new mathematics for the machines

En esta línea, el algoritmo para la realización de la transformada de Fourier de una forma eficaz (FFT) fue un hito en el cálculo numérico y, parece ser, que el artículo donde se publicó originalmente es uno de los más citados de la literatura científica. Por último, insistiremos en que las propiedades de un algoritmo dado, es muchas veces, objeto de estudio *per se*. Así por ejemplo, en la discretización de ecuaciones diferenciales para su resolución numérica uno puede estar incluyendo soluciones espurias *que solo aparecen en la versión discreta de las ecuaciones diferenciales originales*, el comprender y controlar estas soluciones espurias es actualmente un importante campo de investigación en las teorías matemáticas.

En cualquier caso, la sinergia entre mejores algoritmos y mejores ordenadores nos ha llevado en los últimos años a tener acceso a problemas más y más relevantes y a escalas mayores. Así, por ejemplo, cuando el Cray 1S apareció, los programas de predicción meteorológica daban datos precisos hasta un límite de 12 horas. El Cray XMP elevó el límite a 24 horas y ya se empezó a poder modelar el plasma. Con el Cray 2 se podía predecir el tiempo hasta 48 horas, se pudo iniciar el modelado de la dinámica de los

compuestos químicos y se hicieron las primeras estimaciones de la masa de los bosones de Higgs. El Cray YMP permitió 72 horas de predicciones meteorológicas y el estudio de hidrodinámica en 2D. Por último, el C90 permite el diseño de productos farmacéuticos y de aviones.

Breve historia

Una historia completa de los ordenadores debería incluir las *máquinas analógicas* como los ábacos chinos, las *máquinas analíticas* de J. Loom (1805) y C. Babbage (1834), y la calculadora de Marchant (1965). Sin embargo nos vamos a restringir principalmente al área de los ordenadores digitales.

La evolución de los ordenadores digitales se divide en *generaciones*. Cada generación se caracteriza por una mejora substancial sobre la generación precedente en tecnología usada para su construcción, su organización interna (arquitectura) y sus lenguajes de programación.

La era mecánica (1623-1945)

La idea de utilizar máquinas para resolver problemas matemáticos se remonta a principios del siglo XVII. Algunos matemáticos como W. Schickhard, G. Leibnitz y B. Pascal (la contribución de Pascal al cálculo por ordenador fue reconocida por N. Wirth, quien en 1972 bautizó su nuevo lenguaje de ordenador como Pascal) diseñaron e implementaron calculadoras que eran capaces de sumar, restar, multiplicar y dividir.

El primer aparato *programable* fue probablemente la *máquina de diferencias* de C. Babbage, la cual se comenzó a construir en 1823 pero nunca se finalizó. Otra máquina mucho más ambiciosa fue la *máquina analítica*, que fue diseñada en 1842 pero solo fue parcialmente completada por Babbage. Los historiadores piensan que Babbage no pudo completar sus máquinas debido a que la tecnología de la época no era lo suficientemente fiable. Aun cuando no completó ninguna de sus máquinas, Babbage y sus colegas, sobre todo Ada, condesa de Lovelace, reconocieron importantes técnicas de programación como los ciclos iterativos, los índices en las variables o las ramas condicionales.

Una máquina inspirada por el diseño de Babbage fue la primera en ser usada para el cálculo científico. G. Scheutz supo de la *máquina de diferencias* en 1833, y junto con su hijo, E. Scheutz, comenzó a construir una versión más pequeña. Por 1853, habían construido una máquina que podía procesar números de 15 dígitos y calcular diferencias de cuarto orden. Su máquina ganó una medalla de oro en la exhibición de París de 1855, y posteriormente vendieron su máquina al observatorio de Dudley, Albany (New York), donde se utilizó para calcular la órbita de Marte. Uno de los primeros usos comerciales de los ordenadores mecánicos lo realizó la oficina del censo de los Estados Unidos que utilizó un equipo diseñado por H. Hollerith para tabular el censo de 1890. En 1911 la compañía Hollerith se fusionó con un competidor para fundar la corporación que en 1924 sería IBM.

La primera generación de ordenadores electromecánicos (1937-1953): Válvulas de vacío.

Los primeros ordenadores electrónicos se caracterizaron por utilizar interruptores electrónicos (válvulas de vacío) en lugar de relés electro-mecánicos. Las válvulas de vacío se caracterizaban en que no tenían partes móviles y podían abrirse y cerrarse unas 1000 veces más rápidos que los interruptores mecánicos.

El primer intento para construir un ordenador electrónico fue realizado por J.V. Atanasoff, un profesor de Física y Matemáticas en Iowa State Univ. en 1937. Atanasoff quería construir una máquina que ayudase a sus estudiantes a resolver sistemas de ecuaciones diferenciales. En 1941 él y su estudiante C. Berry consiguieron construir una máquina que podía resolver 29 ecuaciones simultáneas con 29 incógnitas. Sin embargo la máquina no era programable y se parecía más a una calculadora electrónica.

Otra máquina construida en aquellos tiempos fue Colossus, diseñada por Alan Turing para los militares británicos en 1943. Esa máquina contribuyó decisivamente a la ruptura de los códigos secretos del ejército alemán durante la segunda guerra mundial. La existencia de Colossus fue mantenida en secreto incluso después de la finalización de la II guerra mundial.

El primer ordenador electrónico programable para cualquier tipo de problema fue el ENIAC (Electronic Numerical Integrator And Computer), construido por J.P. Eckert and J.V. Mauchly en la Universidad de Pensilvania. Su trabajo comenzó en 1943 financiado por el ejército norteamericano que necesitaba un forma rápida de realizar cálculos balísticos durante la guerra. La máquina no estuvo completa hasta 1945 y luego se utilizó extensivamente para el diseño de la bomba de hidrógeno. En 1955 ENIAC se jubiló después de haber hecho importantes contribuciones en el diseño de túneles de viento, números aleatorios y predicciones meteorológicas. Eckert, Mauchly and J. von Neumann habían comenzado ya a diseñar una nueva máquina: EDVAC en la que introdujeron el concepto de *programa almacenado*. ENIAC era controlado por un conjunto de interruptores externos de tal forma que, para cambiar el programa uno tenía que alterar su configuración. EDVAC usaba una memoria que era capaz de almacenar las instrucciones y los datos lo que la hacía incrementar en órdenes de magnitud su velocidad con respecto a ENIAC. Así pues el gran avance de EDVAC fue el reconocer que las instrucciones de programación del ordenador podían ser codificadas como números. Sin embargo, los primeros programas fueron escritos en código máquina. Eckert y Mauchly desarrollaron posteriormente el primer ordenador con éxito comercial: UNIVAC.

Segunda Generación (1954-1962): Transistor

La segunda generación de ordenadores introdujo importantes desarrollos en la arquitectura de los mismos, la tecnología para construir sus circuitos básicos y los lenguajes para programarlos. En esta época los interruptores electrónicos estaban basados en la tecnología de los transistores cuyo tiempo de encendido/apagado era de 0.3 microsegundos. Los primeros ordenadores construidos utilizando esta tecnología fueron TRADIC en los laboratorios Bell (1954) y TX-0 en el laboratorio Lincoln del MIT. La tecnología de las memorias se basaba en núcleos magnéticos que podían ser accedidos en orden aleatorio.

Durante esta segunda generación se introdujeron importantes lenguajes de alto nivel: FORTRAN (1956), ALGOL (1958) y COBOL (1959). Ordenadores comerciales relevantes fueron los IBM 704, 709 y 7094.

Tercera generación (1963-1972): Integración de transistores en superficies planas: circuitos integrados

La tercera generación trajo un notable incremento en la potencia de cálculo. Las mayores innovaciones de esta etapa fueron el uso de circuitos integrados, memorias de semiconductor, formas de programación en paralelo y la introducción de los sistemas operativos y tiempo compartido.

En 1964 S. Cray desarrolló el CDC 6600 que fue la primera arquitectura en usar el paralelismo. Usando 10 unidades funcionales que podían trabajar simultáneamente, y 32 bancos de memoria independientes, el CDC 6600 era capaz de llegar a 1 millón de operaciones en coma flotante por segundo (1 Mflops). En 1969, Cray desarrolló el CDC 7600 que llegó a los 10 Mflops. En esa misma época el IBM 360-195 también llegó a los 10 Mflops utilizando la técnica de la caché de memoria.

En 1963 se desarrolló el lenguaje CPL (Combined Programming Language) que intentaba simplificar el ALGOL. Aun así, CPL resultaba demasiado complicado y fue simplificado aún más por M. Richards en (1967), creando el BPCL (Basic CPL). En 1970 K. Thompson de los laboratorios Bell lo simplificó aún más y los llamó B.

Cuarta generación (1972-1984). Desarrollo de circuitos integrados

En esta generación se introdujeron los circuitos integrados con hasta 100000 componentes por chip. Los procesadores caben completamente en un chip. Las memorias de semiconductores reemplazaron totalmente a las memorias magnéticas. Los ordenadores más emblemáticos fueron los CRAY 1, CRAY X-MP y CYBER 205. Se desarrollaron lenguajes de alto nivel como el FP (Functional Programming) y Prolog (Programming in logic). Esos lenguajes tendían a usar programación *declarativa* en lugar de la programación *imperativa* del Pascal, C, FORTRAN... En un lenguaje declarativo, el programador da la especificación matemática de lo que se debe calcular, dejando el como se calcula al compilador.

Dos sucesos importantes aparecieron durante este periodo: el desarrollo del lenguaje C y del sistema operativo UNIX ambos realizados en los laboratorios Bell. En 1972, D. Ritchie, buscando obtener los comportamientos del CPL y la sencillez del B de Thompson, desarrolló el C. Thompson y Ritchie utilizaron entonces el C para escribir una versión de UNIX para del DEC PDP-11. Este UNIX basado en C fue incluido rápidamente en otros ordenadores de forma que el usuario ya no tenía que aprender un nuevo sistema operativo cada vez que se cambiaba el ordenador.

Quinta generación (1984-1990)

Esta generación se caracterizó por el uso extendido del proceso en paralelo y la fabricación de chips con millones de transistores. Ordenadores estrella de esta época

son: Sequent Balance 8000 (20 procesadores-1 módulo de memoria), iPSC-1 de Intel (128 procesadores-cada uno con su memoria), Connection Machine de Thinking Machines (miles de procesadores muy sencillos trabajando bajo la dirección de una unidad de control).

La actualidad

www.top500.org

Sistemas numéricos y errores

En esta primera sección revisaremos brevemente algunos conceptos de fundamentos de computadores y de representación y almacenaje de números reales en la memoria de un computador. El fin último es evaluar los errores de computación que son directamente atribuibles a las limitaciones de almacenaje.

La mayoría de los computadores de altas prestaciones operan con representación binaria de número reales. Este es así porque los transistores dan dos tipos de salida: 1 ó 0. El sistema binario emplea sistema en base 2, del mismo modo que un sistema decimal usa como base el 10.

En el sistema binario se emplean únicamente dos dígitos: el 1 y el 0. Por ejemplo, el número natural (en representación binaria) 1101 puede escribirse en detalle como

$$1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0. \quad (0.1)$$

Es fácil comprobar que en representación decimal $1101_{(2)} = 13_{(10)}$, donde el subíndice indica la base.

Puesto que un computador, que trabaja en binario, debe comunicarse con los usuarios en sistema decimal, son necesarios procedimientos de conversión de una base a otra.

El primer y mayor problema que uno encuentra es cómo almacenar números en la memoria del computador. Obviamente, cualquier computador que seamos capaces de construir tiene una memoria finita, y, por tanto, sólo puede almacenar una cantidad finita de dígitos. Esto es, sólo es capaz de representar números enteros y fraccionales, Y no todos. En general, sólo es capaz de almacenar números reales de forma aproximada, con una cantidad determinada de dígitos.

En computación, los números se almacenan por *palabras*, siendo la *longitud de palabra* el número de bits que se usa para almacenar un número. Por ejemplo, un computador puede emplear N bits para representar un entero, reservando uno para indicar el signo:



Figura 1. Representación binaria de enteros.

Por consiguiente, si un entero es representado con los N bits n_1, n_2, \dots, n_N de forma que $n_i \in \{0, 1\}$, podemos representar sólo 2^N enteros. Puesto que el signo del entero utiliza un bit (0 representa +), nos deja solo $N-1$ bits para representar *enteros* entre 0 y 2^{N-1} .

Cada procesador maneja de manera óptima una longitud de palabra (386-16 bits, Pentium-32 bits, Mercer-64 bits), de acuerdo con su diseño.

Esta representación de los números enteros, denominada *signo-magnitud*, presenta ciertas desventajas. Una de ellas es que tiene dos representaciones del 0. Suponiendo que $N = 4$, tanto 1000 como 0000 representan el número 0. Otras representaciones alternativas de números enteros en el computador son la representación en *exceso*, representación en *complemento a uno* y representación en *complemento a dos*. Describir con detalle estos otros tipos de representación queda fuera del ámbito de este curso.

Por ser los más relevantes en cálculos científicos, aquí nos centraremos en números reales, en cómo éstos se representan en un computador y cómo se llevan a cabo operaciones con ellos. En cálculos de índole científica, cálculos exclusivamente con enteros no son frecuentes.

Los números reales pueden representarse de dos modos en el ordenador: representación en *coma fija* y representación en *coma flotante*. En la notación en *coma fija*, se reservan un número determinados de bits (véase Figura 2) para la parte decimal.

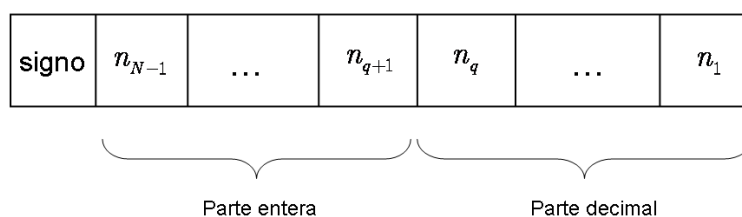


Figura 2. Representación en computador de un número fraccionario en notación punto flotante.

Los valores de N y q dependen del computador en cuestión. En general, cualquier número podría ser representado como

$$(-1)^{signo} \cdot (n_{N-1}n_{N-2} \dots n_{q+1} . n_q \dots n_1)_{(2)} \quad (0.2)$$

Esto es, un bit se utiliza para almacenar el signo, q para la parte decimal y el resto $N - 1 - q$ para la parte entera. Por ejemplo, en una máquina con longitud de palabra $N = 8$ y con dos dígitos reservados para la parte decimal ($q = 2$) del número $001101.11_{(2)}$ se corresponde en base decimal a

$$\begin{aligned} 001101.11_{(2)} &= (-1)^0 \cdot (0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2}) \\ &= 13.75_{(10)}. \end{aligned} \tag{0.3}$$

La ventaja de esta representación es que sabemos en todo momento el error absoluto que cometemos en todos los números: 2^{-q-1} . Su mayor desventaja es que los números pequeños tienen un error relativo (que es el que nos interesa que sea pequeño en métodos numéricos) que puede ser muy grande. Para evitar este problema se emplea la representación en *punto o coma flotante*.

Representación en coma flotante

En un sistema decimal, cualquier número real puede expresarse de forma normalizada en notación científica. Esto significa que el punto decimal se desplaza (de ahí lo de coma o punto flotante), dando lugar a las potencias de 10 necesarias, hasta que haya parte entera y que el primer dígito decimal mostrado sea no nulo. Por ejemplo,

$$\begin{aligned} 0.0002548_{(10)} &= 0.2548 \cdot 10^{-3}_{(10)} \\ 181.1973_{(10)} &= 0.1811973 \cdot 10^3_{(10)}. \end{aligned} \tag{0.4}$$

Esta misma filosofía es la que se sigue para representar (en binario) número reales en un computador: un número real no nulo siempre puede representarse en la forma

$$x_{(10)} = \pm r \cdot 10^n, \tag{0.5}$$

o en base 2:

$$x_{(2)} = \pm q \cdot 2^m, \tag{0.6}$$

donde m es un entero (*exponente*) y $1/2 \leq q < 1$ (si $x \neq 0$) es la *mantisa* normalizada (el primer bit de q es siempre 1, luego no es necesario almacenarlo). En un computador, tanto la mantisa como el exponente, se representan en binario (en base 2).

Supongamos que estamos trabajando con una computadora sencilla que podría tener una longitud de palabra de 32 bits, destinando

- 1 bit para el signo del número real x
- 1 bit para el signo del exponente m
- 7 bits para representar el exponente $|m|$
- 23 bits para representar la mantisa $|q|$.

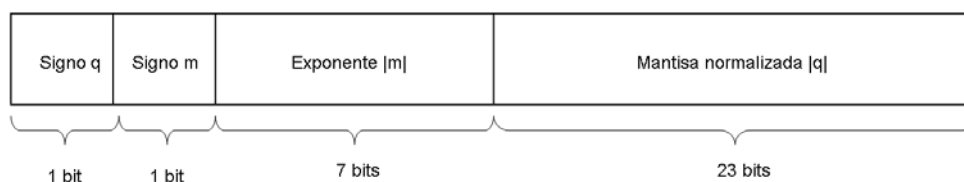


Figura 3. Representación en coma flotante en simple precisión para un computador con 32 bits de longitud de palabra.

Cualquier número almacenado de esta manera se dice que está almacenado en *simple precisión*.

Puesto que para la mantisa hay destinados 7 bits, los números más grandes que se podrían almacenar con esta máquina (cuando $|m| = 111111_{(2)}$) son del orden de $2^{127} \approx 10^{38}$ y los más pequeños como $2^{-127} \approx 10^{-38}$, los cuales suelen ser insuficientes para muchas aplicaciones científicas. Por ello, suelen escribirse programas con variables definidas en *doble precisión* (representadas por dos palabras en vez de una), aunque ralentiza notablemente los cálculos (las operaciones en simple precisión son realizadas por el *hardware*, mientras que las de *doble precisión* se implementan mediante *software*).

La limitación en q (23 bits) implica que $2^{-23-1} \approx 10^{-7}$, esto es, nuestra computadora tiene una precisión limitada a 7 cifras decimales. Por ello, cualquier número con más de 7 cifras decimales será aproximado cuando sea almacenado en memoria.

¿Cómo aproximar esos números? Una solución obvia es truncar el número descartando el resto de decimales. Esto es, el número real $x_{(2)} = (0.n_1n_2 \cdots n_{24}n_{25}n_{26} \cdots) \cdot 2^m$ es aproximado por $x'_{(2)} = (0.n_1n_2 \cdots n_{24}) \cdot 2^m$, resultando $x' < x$. La sucesión llega hasta 24 y no hasta 23, puesto que, como hemos mencionado antes, el primer bit de la mantisa es siempre 1 y no es necesario almacenarlo.

Otra solución podría ser el redondeo hacia arriba, resultando $x''_{(2)} = (0.n_1n_2 \cdots n_{24} + 2^{-24}) \cdot 2^m$. El resultado es $x' < x < x''$.

Se puede demostrar que el error relativo en el redondeo hacia arriba es $|(x - x'')/x| \leq 2^{-24}$. O dicho de otro modo, definiendo $\delta \equiv (x - x'')/x$, se tiene

$$x'' = x \cdot (1 + \delta), \quad |\delta| \leq 2^{-24} \tag{0.7}$$

donde x'' es el número propio de la máquina (machine number) más cercano al número real x . También lo denotaremos como $x'' \equiv fl(x)$.

Desbordamiento

Aparte de los errores por la finitud de espacio reservado en memoria para representar número reales, de forma ocasional, pueden surgir problemas de desbordamiento. Cuando $|m| > 127$, en el transcurso de un cálculo, queda supera del rango permitido por la computadora, se dice que se ha producido un *desbordamiento*, deteniéndose

generalmente la computación. Cuando esto ocurre, las computadoras suelen denotar la variable desbordada como *nan* (not a number, esto es, infinito). En caso contrario, si $|m|$ es demasiado pequeño ($|m| < -127$), la variable es puesta a cero.

Errores derivados de operaciones en coma flotante

El diseño de los computadores suele ser tal que, siempre que se realicen operaciones entre números máquina, primero se realiza la operación, luego se normaliza el resultado, se redondea y, finalmente, se almacena en memoria. En cualquier computadora, las operaciones de suma, resta, producto y división, verifican

$$fl(x \odot y) = (x \odot y) \cdot (1 + \delta), \quad |\delta| \leq \varepsilon, \quad (0.8)$$

donde el símbolo \odot representa una de las cuatro operaciones y ε es la unidad de redondeo. Si x e y son números propios de la máquina, entonces (por (0.7)) $\varepsilon = 2^{-24}$, esto es, $\varepsilon = 2^{-24}$, acota el error relativo en cualquier operación: siempre que se realice una operación estarán presentes errores de redondeo. En caso que x e y sean números propios de la máquina, los errores de truncamiento aparecen ya al almacenar los números en memoria, antes incluso de realizar la operación:

$$fl(x \odot y) = \left((x \cdot (1 + \delta_x)) \odot (y \cdot (1 + \delta_y)) \right) \cdot (1 + \delta_\odot), \quad |\delta_i| \leq 2^{-24} >. \quad (0.9)$$

Al realizar operaciones sucesivas, los errores de redondeo se acumulan. Es fácil comprobar que una suma de dos número propios de la máquina seguido de un producto da lugar a $\varepsilon \approx 2^{-23}$, que es de un orden de magnitud superior a 2^{-24} . En efecto, se puede demostrar que sumar $n + 1$ números máquina da lugar a un error relativo $n\varepsilon$, siendo ε la unidad de redondeo de la máquina que se esté usando [Kincaid y Cheney, 1990].

En general, los resultados que hemos mostrado para la máquina binaria con 32 bits de longitud de palabra son aplicables a otras máquinas que operen con otras bases y con otras longitudes de palabra. No obstante, el valor de la unidad de redondeo ε varía según la máquina: para una máquina que opera en base β , con n bits de mantisa y con redondeo $\varepsilon = \frac{1}{2}\beta^{1-n}$, y con truncado $\varepsilon = \beta^{1-n}$.

Errores absolutos y relativos

Cuando un número real x es aproximado por otro x_A , el error cometido es $x - x_A$. Se define *error absoluto* como

$$|x - x_A| \quad (0.10)$$

y el *error relativo* como

$$\left| \frac{x - x_A}{x} \right|. \quad (0.11)$$

Ambos ya han sido mencionados anteriormente: la desigualdad

$$\left| \frac{x - fl(x)}{x} \right| \leq \varepsilon \quad (0.12)$$

representa el error relativo cometido al representar un número real en una máquina que opera en coma flotante.

Un error común es el de *pérdida de precisión (loss of significance)*. Éste ocurre cuando una operación en coma flotante produce, aparte del error por redondeo, un error relativo muy significativo. Por ejemplo, cuando se sustraen dos cantidades muy próximas: considérese la diferencia de los números reales $x=0.12345678912345$ e $y = 0.123456789$. El resultado es 0.00000000012345 , siendo almacenado en una máquina que opera en coma flotante con 10 dígitos decimales como 0.0000000002 , por redondeo (la normalización posterior del exponente sólo añade ceros a la derecha del resultado). El error relativo de la operación es, por tanto,

$$\left| \frac{0.00000000012345 - 0.0000000002}{0.00000000012345} \right| \approx 62\%. \quad (0.13)$$

El error sería del 18% en una máquina que opera por truncamiento.

Este tipo de errores, pueden evitarse realizando una programación cuidadosa. Por ejemplo, la resta $y \leftarrow -1 + \sqrt{x^2 + 1}$ produce pérdida de precisión para valores pequeños de x . El problema podría evitarse reprogramando la operación como

$$y \leftarrow \frac{x^2}{1 + \sqrt{x^2 + 1}}.$$

En otras situaciones el error puede reducirse empleando declaración de variables en *doble precisión*. En este modo de cálculo, cada número real es representado mediante dos palabras (a la hora de operar, no de almacenarse). No obstante, la aritmética en doble precisión incrementa el coste computacional, puesto que usualmente se implementa por software en vez de hardware como la precisión simple.

¿Se puede cuantificar la pérdida de precisión? El teorema siguiente proporciona las cotas superior e inferior en las que se encuentra el error relativo [Kincaid y Cheney, 1990].

Si x e y son dos números propios de una máquina binaria positivos, tal que $x > y$ y

$$2^{-q} \leq (1 - y/x) \leq 2^{-p} \quad (0.14)$$

Entonces al menos q bit y como mucho p bits se pierden en la diferencia $x - y$.

Existen otras situaciones en las que puede ocurrir una pérdida de precisión en las operaciones. Es muy típico el caso de funciones que reciben argumentos elevados. Por ejemplo, si consideramos el $\cos(x)$, la subrutinas disponibles en las computadoras suelen reducir el rango de x si este sobrepasa 2π , haciendo uso de la periodicidad de la función. Las operaciones de reducción de rango (productos y restas) de x pueden dar lugar a pérdida de precisión (x_A), donde el error relativo $|(x - x_A)/x|$ sea muy elevado. En tal caso, la función $\cos(x_A)$ dará resultados totalmente ficticios.

Cálculos estables e inestables

Se dice que un cálculo es *inestable* cuando pequeños errores cometidos en un determinado paso del proceso de resolución se propagan a los pasos siguientes en mayor

magnitud afectando seriamente a la precisión del cálculo. Esto es, cuando da lugar a errores relativos grandes. Véase algunos ejemplos en [Kincaid y Cheney, 1990].

En la mayoría de los casos es posible implementar las ecuaciones de tal forma que los cálculos vuelvan a ser estables y no produzcan errores relativos significativos. Cómo reescribir las ecuaciones, depende del problema concreto.

Condicionamiento

Comúnmente se emplean las palabras *condicionado* y *condicionamiento* para indicar cuán sensible es la solución de un problema a pequeños cambios en los datos de entrada. Se dice que un problema está *mal condicionado* si pequeños cambios en las entradas producen respuestas dispares. En algunos problemas es posible definir un número de condicionamiento, el cual si es grande indica cuándo un problema está mal condicionado.

Un ejemplo sencillo para ilustrar este concepto es el siguiente. Dada una función $f(x)$, supongamos que queremos evaluar el efecto en f de una perturbación pequeña $x + \delta$. El error relativo será entonces

$$\frac{f(x + \delta) - f(x)}{f(x)} \approx \frac{\delta f'(x)}{f(x)} = \left(\frac{x \cdot f'(x)}{f(x)} \right) \left(\frac{\delta}{x} \right). \quad (0.15)$$

El factor $\left(\frac{\delta}{x} \right)$ es la magnitud relativa del error y, por tanto, $\left(\frac{x \cdot f'(x)}{f(x)} \right)$ representa el *número de condicionamiento*.

Otro ejemplo de número de condicionamiento es el asociado a la resolución de sistemas de ecuaciones lineales $Ax = b$, donde A es una matriz $n \times n$, x es el vector de incógnitas $n \times 1$ y b es el vector $n \times 1$ de términos independientes. En este caso, el número de condicionamiento $\kappa(A)$ del problema, esto es, de la matriz A , se define como

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|, \quad (0.16)$$

donde $\|\cdot\|$ es una norma matricial y A^{-1} es la inversa de la matriz A . En el caso que $\kappa(A)$ sea elevado, el problema está mal condicionado y las soluciones (numéricas) dadas por $Ax = b$ deben ser tenidas en cuenta con mucho cuidado.

Problemas de discretización en ecuaciones diferenciales

Otros conceptos surgidos a la hora de discretizar ecuaciones diferenciales son el *error de truncamiento local* y *global* (página 519 de [Kincaid y Cheney, 1990]), *convergencia*, *monotonidad* ([Toro, 1999]), *estabilidad* (página 515 de [Kincaid y Cheney, 1990]) del método numérico y *consistencia* (página 517 de [Kincaid y Cheney, 1990]). Estos conceptos, asociados a la bondad del método de resolución, serán tratados con detalle en el capítulo de resolución de ecuaciones diferenciales ordinarias y en derivadas parciales.

Lógica de programación y estructuras de datos

Programación estructurada

Uno de los requisitos que debe seguir el buen programador es redactar un programa legible e inteligible, incluso para personas que no saben programar. Para ello es importante que el código mantenga una buena *estructura* (formato que se usa al redactar un programa) y que esté bien comentado, de tal forma que facilite la comprensión, modificación y depuración del programa. Por el contrario, un programa no estructurado es aquel en el que no se ha hecho ningún esfuerzo, o muy poco, para ayudar a leerlo y comprenderlo. Hay que decir que, desde el punto de vista del computador, no hay diferencia entre un programa estructurado y otro que no lo es. La diferencia está en cómo de fácil le resulta a una persona entender y manipular el código.

Un programa bien estructurado está constituido por *bloques*, donde cada bloque está bien definido y formado por un grupo de instrucciones que desempeñan una determinada función. Cada bloque tiene una única entrada al mismo al comienzo de los mismos, y una única salida al final. Así pues, un programa estructurado no es más que un conjunto de bloques, en vez de una lista continua de instrucciones.

Es importante hacer notar que un programa basado en una estructura de bloques no debe tener instrucciones de salto (v.gr. el GOTO de Basic o Fortran en sus primeras versiones), también llamadas de transferencia incondicional, que rompen el desarrollo secuencial y por bloques del programa.

Existen tres tipos de bloques necesarios (y suficientes) para programar cualquier aplicación de interés. Sin excepción. Estos son (véase Figura 4),

1. Bloque secuencial
2. Bloque repetitivo (bucle o iteración)
3. Bloque de selección

El *bloque secuencial* es el tipo más simple de bloque de programación, estando formado por un conjunto de sentencias, una seguida de otra. Un bloque repetitivo efectúa la ejecución iterada de una sección del programa y, por último, un bloque de selección ofrece la posibilidad de realizar secuencias diferentes de instrucciones (ramas) según una condición.

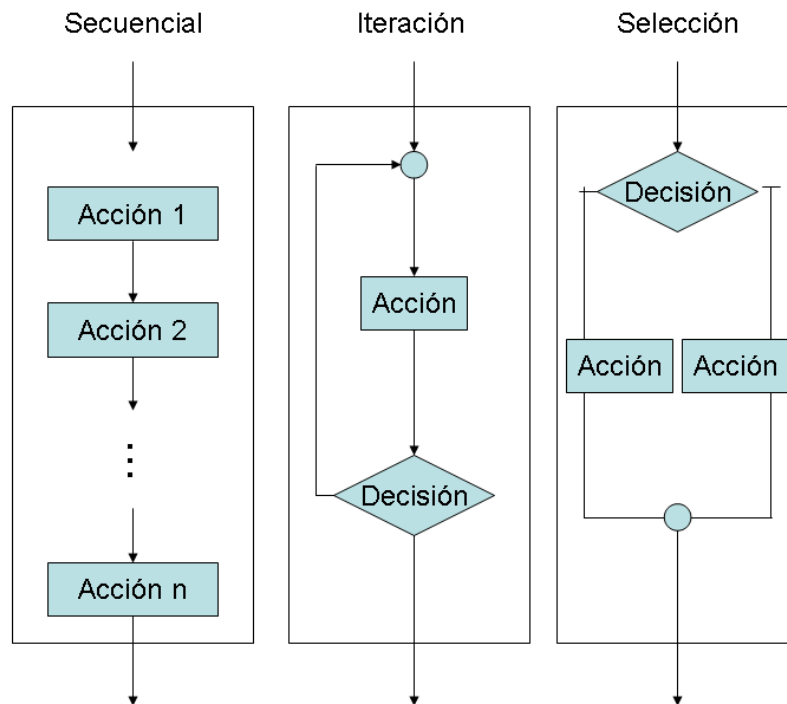


Figura 4. Conceptos de las tres clases de bloques de programación.

No obstante, hoy en día las aplicaciones informáticas crecen en volumen y complejidad, por lo que en numerosas aplicaciones las técnicas de programación estructurada no son suficientes. Existen otros paradigmas de la programación como por ejemplo la programación orientada a objetos u otros entornos de programación que facilitan la programación de grandes aplicaciones. Ambos quedan fuera del objeto de este curso.

Algoritmos

Un algoritmo es un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

Mediante algoritmos, somos capaces de resolver problemas cotidianos, como cocinar una tortilla de patatas, o problemas matemáticos, como resolver una ecuación diferencial ordinaria por métodos numéricos. En ambas situaciones el algoritmo verifica las siguientes propiedades:

1. Por definición, son finitos.
2. Reciben información de entrada y producen salidas (resultados).
3. Están bien definidos: las operaciones que se llevan a cabo se establecen de forma precisa sin ambigüedades.

Uno podría además añadir que son deterministas, en el sentido que diferentes realizaciones con las mismas condiciones (mismas entradas), siguiendo el algoritmo, se llega a idéntica solución (mismas salidas). No obstante, uno podría definir *algoritmos probabilísticos*, los cuales dependen de una o varias variables aleatorias. En cualquier caso, al ser implementados en un computador, el resultado es invariablemente determinista.

Los algoritmos pueden ser implementados de muchas maneras: mediante nuestro lenguaje, *pseudocódigo*, *diagramas de flujo*, lenguajes de programación, etc. Las descripciones en lenguaje natural tienden a ser ambiguas y extensas, por lo que se recurre a un lenguaje de símbolos para expresarlo (como las matemáticas). Pseudocódigo y diagramas de flujo recurren a símbolos para expresar algoritmos, y son independientes (más generales) que un lenguaje de programación específico.

En resumen, la descripción de un algoritmo usualmente se hace en tres niveles (www.wikipedia.org):

- Descripción a alto nivel: Se establece el problema, se selecciona un modelo matemático y se explica el algoritmo de manera verbal, posiblemente con ilustraciones (v.gr. diagramas de flujo) y omitiendo detalles.
- Descripción formal: Se emplea pseudocódigo para describir la secuencia de pasos que encuentran la solución.
- Implementación: Se muestra el algoritmo expresado en un lenguaje de programación específico o algún objeto capaz de llevar a cabo instrucciones.

Diagramas de Flujo

Los diagramas de flujo u organigramas son esquemas que se emplean para representar gráficamente un algoritmo. Se basan en la utilización de diversos símbolos convenidos para representar operaciones específicas. Se les llama diagramas de flujo porque los símbolos utilizados se conectan por medio de flechas para indicar la secuencia de operación.

Es recomendable (especialmente en problemas complicados o muy largos), a la hora de establecer el problema y antes de comenzar a programar, expresar mediante diagramas de flujo el algoritmo.

Los diagramas de flujo se dibujan generalmente usando símbolos estándares. No obstante, símbolos especiales pueden ser introducidos *ad hoc* siempre y cuando sean requeridos. Los símbolos más comunes se muestran a continuación:






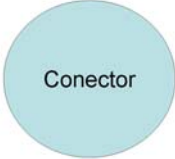

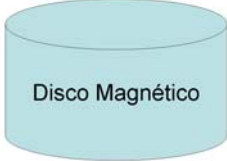
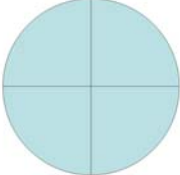
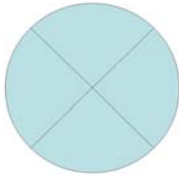




 <p>Inicio / Fin</p>	 <p>Proceso</p>
 <p>Datos I/O</p>	 <p>Decisión</p>
 <p>Documento Impresora</p>	 <p>Conector</p>
 <p>Datos Almacenados</p>	 <p>Disco Magnético</p>
 <p>Operación lógica OR</p>	 <p>Operación lógica AND</p>
 <p>Display Mostrar datos</p>	 <p>Líneas de flujo</p>
 <p>Entrada manual</p>	 <p>Retardo</p>

Tabla 1. Símbolos estándar empleados en diagramas de flujo.

Algunas reglas para dibujar diagramas de flujo son

1. Los Diagramas de flujo deben escribirse de arriba hacia abajo, y/o de izquierda a derecha.
2. Los símbolos se unen con líneas, las cuales tienen en la punta una flecha que indica la dirección que fluye la información, se deben de utilizar solamente líneas de flujo horizontal o verticales (nunca diagonales).
3. Se debe evitar el cruce de líneas, para lo cual se quisiera separar el flujo del diagrama a un sitio distinto, se pudiera realizar utilizando los conectores. Se debe tener en cuenta que solo se vaya a utilizar conectores cuando sea estrictamente necesario.
4. No deben quedar líneas de flujo sin conectar.
5. Todo texto escrito dentro de un símbolo debe ser legible, preciso, evitando el uso de muchas palabras.
6. Todos los símbolos pueden tener más de una línea de entrada, a excepción del símbolo final.
7. Solo los símbolos de decisión pueden y deben tener más de una línea de flujo de salida.

Dos ejemplos de diagramas de flujo son los siguientes:

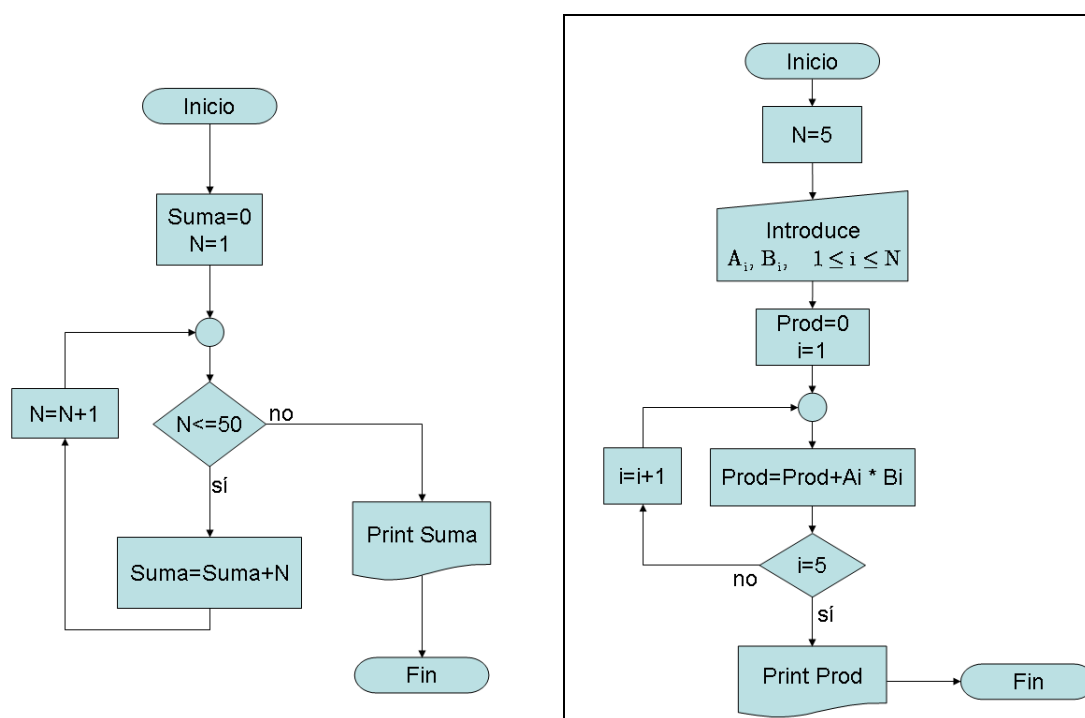


Figura 5. (Izquierda) Diagrama de flujo de un algoritmo que calcula la suma de los primeros 50 números enteros. Implementado mediante un ciclo *while*. (Derecha) Diagrama de flujo de un algoritmo que calcula el producto escalar de los vectores A y B . Implementado mediante un ciclo *for*.

Pseudocódigo

El pseudocódigo es un lenguaje de alto nivel empleado para describir algoritmos, igual que los diagramas de flujo aunque más cercano a la computadora. El objetivo último del pseudocódigo es facilitar la comprensión del algoritmo. Emplea ciertas convenciones de los lenguajes de programación, como C o Fortran, pero omite subrutinas, declaración de variables, etc. y no sigue reglas estructurales rígidas.

Concretamente, el pseudocódigo describe un algoritmo utilizando una mezcla de frases en lenguaje común, instrucciones de programación y palabras clave que definen las estructuras básicas. Su objetivo es permitir que el programador se centre en los aspectos lógicos de la solución a un problema. El pseudocódigo varía de un programador a otro, es decir, no hay una estructura semántica ni arquitectura estándar.

El pseudocódigo asociado a los ejemplos de la Figura 5 es el siguiente

<pre> Suma ← 0 N ← 1 While N ≤ 50 Then Suma ← Suma + N N ← N + 1 End While Output Suma </pre>	<pre> N ← 5 Input A_i, B_i 1 ≤ i ≤ N Prod ← 0 For i = 1 to N Prod ← Prod + A_i * B_i End for Output Prod </pre>
---	--

Figura 6. (Izquierda) Pseudocódigo asociado al algoritmo cuya función es sumar los 50 primeros números enteros. (Derecha) Pseudocódigo asociado al algoritmo cuya función es realizar el producto escalar de los vectores A y B .

Un breve resumen de símbolos que emplearemos en pseudocódigo que sirvan de referencia, podría ser el siguiente

Asignación	Operadores aritméticos
$N \leftarrow 1$	$*, /, +, -, \dots$
Operadores relacionales	Entrada manual de datos
$<, >, \leq, \geq, \neq, \text{and, or}$	Input a, b, c
Controladores por contador	Controladores por centinela
For $i = 1$ to N ... End for	While $<\text{condición}>$ Then ... End While
Selección simple	Selección doble
If $<\text{condición}>$ Then ... End If	If $<\text{condición}>$ Then ... Else ...

	End If
Selección anidada	Selección múltiple
If <condición 1> Then ... Else If <condición 2> Then ... Else ... End If End If	Switch <expresión entera o carácter> Case valor1: ... End Case Case valor2: ... End Case ... End Switch
Otro controlador por centinela	Comentarios
Do ... While <condición>	% comentario
Salida datos por pantalla	Salida datos por impresora
Output	Print

Tabla 2. Estructuras en pseudocódigo.

Elementos de programación

En esta sección presentaremos los ingredientes básicos requeridos para poner en funcionamiento un algoritmo. Compararemos diferentes ejemplos mediante diagramas de flujos, pseudocódigo, C y Matlab.

Inicio, Fin

Todos los programas tienen un inicio y un final (son finitos) y por regla general tienen directivas especiales para que el compilador los identifique.

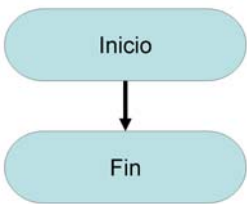
	<p><i>% En pseudocódigo no es usual indicar el principio y el final del algoritmo, aunque pudieran definirse ad hoc.</i></p>
<pre># include <stdio.h> main(){ printf("Programa simple en C\n"); }</pre>	<p><i>% Matlab no emplea directivas específicas para indicar el inicio y el final de los programas</i></p>

Tabla 3. Inicio/fin en diagramas de flujo, pseudocódigo, C y Matlab.

Tipos de datos: declaración y asignación de variables

Un variable no es más que un espacio de memoria del computador reservado para un tipo específico de datos y con un nombre para identificarlo. Es frecuente que una misma variable vaya cambiando de valor según vaya avanzando el programa.

Por regla general, todos los lenguajes de programación necesitan declarar las variables; aunque existen excepciones (Matlab). Por una parte, la declaración es necesaria para que el compilador conozca por adelantado el nombre y el tipo de variables que se están usando. Por otra, facilita la comprensión y lectura del código, minimizando errores de programación.

Ni diagramas de flujo, ni pseudocódigos, ni Matlab declaran variables, aunque es una buena costumbre inicializarlas. Véase la tabla siguiente.

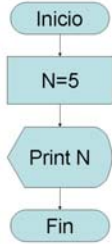
 <pre> graph TD Inicio([Inicio]) --> N5[N=5] N5 --> PrintN{{Print N}} PrintN --> Fin([Fin]) </pre>	<p><i>% En pseudocódigo no se declaran las variables. No obstante, es recomendable asignarles un valor inicial</i></p> <p>$N \leftarrow 5$ Print N</p>
<pre> #include <stdio.h> // Declaro la variable int N; main(){ // Asigno valor N=5; // Resultado por pantalla printf("N=%d es un entero\n",N); } </pre>	<p><i>% En Matlab no se declaran las variables. No obstante, es recomendable asignarles un valor inicial</i></p> <p><i>% Asigno valor inicial</i> N=5; <i>% Saco resultado por pantalla</i> sprintf('N=%d es un entero\n',N)</p>

Tabla 4. Declaración y asignación de variables.

En C, además de declarar las variables, éstas deben incluir explícitamente el tipo, el cual no se puede cambiar dinámicamente. Matlab, en cambio, es más flexible: cuando Matlab encuentra un nuevo nombre de variable, reserva el espacio debido en memoria (mayor o menor según el tipo) y crea la variable. Si la variable ya existe le cambia el contenido y, si es necesario, reserva un nuevo espacio de memoria.

Por ejemplo,

```
% Programa en Matlab que asigna diferentes valores y tipos a una variable
```

Variable=35.5	% Variable es una matriz real de 1x1
Variable=Variable * i	% Variable es ahora una matriz compleja de 1x1
Variable='Hola'	% Finalmente Variable es una cadena de caracteres

Este tipo de asignaciones y declaraciones serían imposibles de realizar.

Las operaciones más frecuentes en Matlab son

+	Suma
-	Resta
*	Producto
/	División
^	Potencia
()	Paréntesis (precedencia)
'	Traspuesta compleja conjugada

Además, Matlab proporciona un elevadísimo número de constantes y funciones predeterminadas, como $\pi(pi)$, $\sin()$, $\cos()$, $\exp()$, entre otras. Listas de funciones se pueden obtener mediante las siguientes instrucciones en línea de comandos

help elfun	% Listado de funciones matemáticas elementales
help especfun	% Listado de funciones matemáticas avanzadas
help elmat	% Listado de funciones matemáticas avanzadas

Estructuras de control de flujo

A continuación, presentamos los diferentes tipos de estructuras de control de flujo.

Estructuras de selección

Permiten decidir qué bloque de instrucciones se van ejecutar, dependiendo de una o varias condiciones.

Simple

<pre> graph TD Inicio([Inicio]) --> Numero[Número] Numero --> Condicion{Número > 5} Condicion -- si --> Mayor[Mayor que 5] Condicion -- no --> Conector(()) Mayor --> Conector Conector --> Fin([Fin]) </pre>	<p>Input <i>numero</i></p> <p>If <i>numero</i> > 5 Then Output “Mayor que 5.” End If</p>
<pre> #include <stdio.h> // Declaro la variable int numero; main(){ // Valor proporcionado por usuario printf(“Déme un número entero\n”); scanf(“%d”,&numero); if (numero>5) printf(“Mayor que 5.\n”); } </pre>	<p><i>% Definido como una función</i> <i>% largertanfive.m</i> <i>% Ejecutable en línea de comando</i> <i>como</i> <i>% >> largertanfive(7), por ejemplo</i></p> <pre> function largertanfive(numero) if (numero>5) f='Mayor que 5' end </pre>

Tabla 5. Estructuras de selección simple.

Doble

<pre> graph TD Inicio([Inicio]) --> Numero[Número] Numero --> Condicion{Número > 5} Condicion -- si --> Mayor1[Mayor que 5] Condicion -- no --> Mayor2[Mayor que 5] Mayor1 --> Conector(()) Mayor2 --> Conector Conector --> Fin([Fin]) </pre>	<p>Input <i>numero</i></p> <p>If <i>numero</i> > 5 Then Output “Mayor que 5.” Else Output “Menor o igual a 5.” End If</p>
<pre> #include <stdio.h> // Declaro la variable int numero; </pre>	<p><i>% Definido como una función</i> <i>% largertanfive.m</i> <i>% Ejecutable en línea de comando</i> <i>como</i> <i>% >> largertanfive(7), por ejemplo</i></p>

<pre>main(){ // Valor proporcionado por usuario printf("Déme un número entero\n"); scanf("%d",&numero); if (numero>5){ printf("Mayor que 5.\n"); } else{ printf("Menor o igual a 5.\n"); } }</pre>	<pre>function largertanfive(numero) if (numero>5) f='Mayor que 5.' elseif f='Menor o igual a 5.' end</pre>
--	--

Tabla 6. Estructuras de selección doble.

Anidada

	<p>Input <i>numero</i></p> <p>If <i>numero</i> > 5 Then Output "Mayor que 5." Else Output "Menor o igual a 5." If <i>numero</i> > 4 Then Output "Igual a 5." Else Output "Menor que 5." End If End If</p>
<pre># include <stdio.h> // Declaro la variable int numero; main(){ // Valor proporcionado por usuario printf("Déme un número entero\n"); scanf("%d",&numero); if (numero>5){ printf("Mayor que 5.\n"); } }</pre>	<pre>% Definido como una función % largertanfive.m % Ejecutable en linea de comando como % >> largertanfive(7), por ejemplo function largertanfive(numero) if (numero>5) f='Mayor que 5.' elseif f='Menor o igual a 5.' if (numero>4) f='Igual a 5.'</pre>

<pre> else{ printf("Menor o igual a 5."); if (numero>4){ printf("Igual a 5.\n"); } else{ printf("Menor que 5.\n"); } } } </pre>	<pre> elseif f='Menor que 5.' end end </pre>
--	--

Tabla 7. Estructura de selección anidada.

Múltiple

	<p>Input <i>numero</i></p> <p>Switch <i>numero</i></p> <p>Case 4: Output "Igual a 4." End Case</p> <p>Case 3: Output "Igual a 3." End Case</p> <p>Case 2: Output "Igual a 2." End Case</p> <p>Case 1: Output "Igual a 1." End Case</p> <p>Default : Output "Otro caso." End Case</p> <p>End Switch</p>
<pre> #include <stdio.h> // Declaro la variable int numero; main(){ // Valor proporcionado por usuario </pre>	<pre> % Definido como una función % seleccion.m % Ejecutable en línea de comando como % >> seleccion(7), por ejemplo function seleccion(numero) </pre>

<pre> printf("Déme un número entero\n"); scanf("%d",&numero); switch (numero){ case 4: printf("Igual a 4.\n"); break; case 3: printf("Igual a 3.\n"); break; case 2: printf("Igual a 2.\n"); break; case 1: printf("Igual a 1.\n"); break; default: printf("Otro caso.\n"); break; } </pre>	<pre> switch (numero) case 4 'Igual a 4' case 3 'Igual a 3' case 2 'Igual a 2' case 1 'Igual a 1' otherwise 'Otro caso' end </pre>
--	--

Tabla 8. Estructura de selección múltiple.

Estructuras iterativas

Permiten que una o más instrucciones se ejecuten varias veces.

Controladas por contador

Contiene 4 partes importantes:

1. El valor en el que comienza el bucle
2. La condición bajo la cual el bucle continua
3. Los cambios que tienen lugar en cada bucle
4. Las instrucciones del bucle

En los siguientes ejemplos se calcula la aproximación N -ésima del desarrollo de Taylor para la exponencial $\exp(x) \approx \sum_{n=0}^N \frac{x^n}{n!}$. Tanto C como Matlab poseen funciones

propias para determinar la función exponencial en cualquier punto sin recurrir a este tipo de bloques. Es sólo para mostrar los ejemplos de estructuras iterativas.

<pre> graph TD Inicio([Inicio]) --> Input[/N, x/] Input --> Init[factorial_n = 1 x_n = 1 exponencial = 0 i = 1] Init --> Loop(()) Loop --> Process[exponencial = exponencial + x_n / factorial_n x_n = x_n * x factorial_n = factorial_n * contador i = i + 1] Process --> Decision{i = N} Decision -- no --> Loop Decision -- si --> Output[exponencial] Output --> Fin([Fin]) </pre>	<p>Input x, N</p> <p>$factorial_n \leftarrow 1.0$ $x_n \leftarrow 1.0$ $exponencial \leftarrow 0.0$</p> <p>For $i = 1$ to N $exponencial \leftarrow exponencial$ $\quad + x_n / factorial_n$ $x_n \leftarrow x_n * x$ $factorial_n$ $\quad \leftarrow factorial_n * i$</p> <p>End for</p> <p>Output $exponencial$</p>
<pre> #include <stdio.h> // Declaro variables double exponencial; // exp(x) approx. double x; double factorial_n; // para el factorial del // denominador double x_n; // producto int i; // índice int N; // orden aproximación main(){ // Valor proporcionado por usuario printf("Dónde desea evaluar exp(x)?\n"); scanf("%lf",&x); printf("Qué orden de aproximación?\n"); scanf("%d",&N); // Inicializo exponencial, factorial_n, // x_n factorial_n=1.0; x_n=1.0; exponencial=0.0; for(i=1;i<N;i++){ exponencial += x_n/factorial_n; </pre>	<pre> function exponencial(x,N) factorial_n=1.0; x_n=1.0; exponencial=0.0; for i=1:N exponencial = exponencial + x_n./factorial_n; x_n = x_n * x; factorial_n = factorial_n * i; end sprintf('exp(%g) approx. %g.\n',x,exponencial) </pre>

```

        x_n *= x;
        factorial_n *= i;
    }

    printf("exp(%lf) approx. %lf\n",x,
exponencial);
}
    
```

Tabla 9. Bucle *for*.

Controladas por centinela

Otra estructura iterativa es el bucle *while* consta de los mismos elementos del bucle *for*. La diferencia se encuentra en la disposición de los elementos dentro del programa. El bucle *while* es más conveniente cuando no se sabe por adelantado el número de veces que se va a repetir el bucle.

El ejemplo siguiente determina la aproximación de la función exponencial empleando un bucle *while*.

	<p>Input x, N</p> <p>$factorial_n \leftarrow 1.0$ $x_n \leftarrow 1.0$ $exponencial \leftarrow 0.0$ $i = 1$</p> <p>While $i \leq N$ $exponencial \leftarrow exponencial$ $\quad + x_n / factorial_n$ $x_n \leftarrow x_n * x$ $factorial_n$ $\quad \leftarrow factorial_n * i$ $i = i + 1$</p> <p>End While</p> <p>Output $exponencial$</p>
<pre> # include <stdio.h> // Declaro variables double exponencial; // exp(x) approx. double x; double factorial_n; // para el factorial del // denominador double x_n; // producto </pre>	<pre> function exponencial(x,N) factorial_n=1.0; x_n=1.0; exponencial=0.0; i=1; while i<N exponencial = exponencial + </pre>

<pre> int i; // índice int N; // orden aproximación main(){ // Valor proporcionado por usuario printf("Dónde desea evaluar exp(x)?\n"); scanf("%lf",&x); printf("Qué orden de aproximación?\n"); scanf("%d",&N); // Inicializo exponencial, factorial_n, // x_n factorial_n=1.0; x_n=1.0; exponencial=0.0; i=1; while(i<N){ exponencial += x_n/factorial_n; x_n *= x; factorial_n *= i; i++; } printf("exp(%lf) approx. %lf\n",x, exponencial); } </pre>	<pre> x_n./factorial_n; x_n = x_n * x; factorial_n = factorial_n * i; i=i+1; end sprintf('exp(%g) approx. %g.\n',x,exponencial) </pre>
--	---

Tabla 10. Bucle *while*.

El último de los tres bucles iterados es el *do-while*. Es similar al ciclo *while*, con la diferencia que la condición se evalúa después de ejecutar el cuerpo del bucle. Es decir, el cuerpo del bucle siempre se ejecuta al menos una vez. Seguimos con el ejemplo de la exponencial.

<pre> graph TD Inicio([Inicio]) --> Input[/N, x/] Input --> Init[factorial_n = 1 x_n = 1 exponencial = 0 i = 1] Init --> Connector(()) Connector --> Loop[exponencial = exponencial + x_n / factorial_n x_n = x_n * x factorial_n = factorial_n * contador i = i + 1] Loop --> Decision{i < N} Decision -- si --> Connector Decision -- no --> Output[/exponencial/] Output --> Fin([Fin]) </pre>	<p>Input x, N</p> <p>$factorial_n \leftarrow 1.0$ $x_n \leftarrow 1.0$ $exponencial \leftarrow 0.0$ $i = 1$</p> <p>Do</p> <p>$exponencial \leftarrow exponencial$ $\quad + x_n / factorial_n$ $x_n \leftarrow x_n * x$ $factorial_n$ $\quad \leftarrow factorial_n * i$ $i = i + 1$</p> <p>While $i \leq N$</p> <p>Output $exponencial$</p>
<pre> #include <stdio.h> // Declaro variables double exponencial; // exp(x) approx. double x; double factorial_n; // para el factorial del // denominador double x_n; // producto int i; // índice int N; // orden aproximación main(){ // Valor proporcionado por usuario printf("Dónde desea evaluar exp(x)?\n"); scanf("%lf",&x); printf("Qué orden de aproximación?\n"); scanf("%d",&N); // Inicializo exponencial, factorial_n, // x_n factorial_n=1.0; x_n=1.0; exponencial=0.0; i=1; do{ </pre>	<pre> % Matlab no implementa este tipo de bucles, por otra parte muy semejantes a los anteriores. </pre>

<pre> exponencial += x_n/factorial_n; x_n *= x; factorial_n *= i; i++; }while(i<N); printf("exp(%lf) approx. %lf\n",x, exponencial); } </pre>	
---	--

Tabla 11. Bucle *do-while*.

Ejemplo

Implementar un algoritmo en Matlab, con su diagrama de flujo asociado, para

determinar coseno $\cos(x) \approx \sum_{n=0}^N \frac{(-1)^n x^{2n}}{(2n)!}$ hasta orden N .

Bibliografía

- Antonakos, J.L. and Jr., K.C.M., 1997. Programación Estructurada en C. Prentice Hall Iberia, Madrid, España.
- Burden, R.L. and Faires, J.D., 1985. Numerical Analysis. PWS-Kent Publishing Company, Boston, EE. UU.
- Conte, S.D. and Boor, C.d., 1980. Elementary Numerical Analysis: An algorithmic approach. International Series in Pure and Applied Mathematics. McGraw-Hill, Nueva York, EE. UU.
- Gerald, C.F. and Wheatley, P.O., 2000. Análisis Numérico con Aplicaciones. Pearson Educación, México D.F.
- Hefferson, J., 2006. Linear Algebra. Publicación electrónica (<http://joshua.smcvt.edu>).
- Kernighan, B.W. and Ritchie, D.M., 1988. The C Programming Language. Prentice Hall Inc.
- Kincaid, D. and Cheney, W., 1990. Numerical Analysis. Brooks/Cole Publishing, Pacific Grove, California, EE. UU.
- Lang, S., 1987. Linear Algebra. Undergraduate Texts in Mathematics. Springer, Nueva York, EE.UU. .
- Molina, J.J.Q., 1996. Ecuaciones Diferenciales, Análisis Numérico y Métodos Matemáticos. Editorial Santa Rita, Granada.
- O'Connor, J.L.F., 1993. Tecnicas de Cálculo para Sistemas de Ecuaciones, Programación Lineal y Programación Entera, Madrid, España.
- Toro, E.F., 1999. Riemann Solvers and Numerical Methods for Fluid Dynamics. Springer, Berlin.