Capítulo 1

Cálculo Numérico: Precisión, Programación y Algoritmos

1.1. Objetivos del cálculo numérico

El cálculo numérico tiene por objetivo la obtención de soluciones numéricas con precisión arbitraria y estimación del error, y mediante algoritmos eficientes, a problemas matemáticos bien planteados y solubles numéricamente. Esta sencilla frase contiene, de forma implícita, una alusión a las principales bestias negras del cálculo numérico, que trataremos en este capítulo. En primer lugar la frase *precisión arbitraria* contiene una alusión implícita al tipo de aritmética empleado. Los ordenadores utilizan una cantidad de memoria determinada para almacenar los números, lo cual limita de partida la precisión de los cálculos. Si se utilizan los tipos nativos de un lenguaje de programación, es decir las clases de números definidos directamente por el compilador o interpretador de dicho lenguaje, la representación de los número está limitada a una veintena de cifras decimales (depende del tamaño de las palabras del procesador, 32 o 64 bits). Sin embargo, llegado el caso, se puede siempre utilizar aritméticas de precisión arbitraria mediante programación, pudiendo realizar los cálculos con miles de cifras decimales si se considera necesario. Por lo tanto, los algoritmos del cálculo numérico deben de funcionar para cualquier precisión deseada. Por otro lado, el error cometido en los resultados numéricos proporcionados por un algoritmo no es debido únicamente al tipo de aritmética utilizado, sino sobre todo a las aproximaciones empleadas en la elaboración del algoritmo, por lo cual los algoritmos numéricos deben de proporcionar también una estimación del error cometido, que debe ser controlable, es decir, deben de existir parámetros del algoritmo que cuando tiendan a un determinado valor, el error del algoritmo tienda a 0. La frase *problemas matemáticos bien planteados y solubles numéricamente* contiene también una alusión a uno de los principales problemas, la solubilidad numérica. Si se aplica un algoritmo a un problema que no está bien planteado matemáticamente, conducirá sin lugar a dudas a soluciones erróneas. Pero un problema aparentemente bien planteado desde el punto de vista matemático puede no ser abordable numéricamente. Este es el caso de problemas matemáticos que implican valores infinitos o funciones discontinuas, que no son susceptibles de una representación numérica correcta. Esto conduce al concepto de *cal-*

culabilidad, ampliamente estudiado en tiempos recientes, y que trataremos brevemente en este capítulo. Obviamente, el cálculo numérico sólo puede abordar la solución de problemas calculables, que puedan ser codificados sin ambigëedad mediante programas de ordenador y que puedan ser adecuadamente resueltos con un número finito de operaciones. Finalmente, es frecuente que un mismo problema se pueda resolver de forma correcta por diferentes algoritmos, con similar precisión. Sin embargo, dos algoritmos distintos, comparables desde el punto de vista de la precisión numérica obtenida, pueden necesitar volúmenes de cálculo muy diferentes. Diremos que el algoritmo que necesita un menor volumen de cálculo es más *eficiente*. Una de las principales motivaciones del cálculo numérico es el encontrar algoritmos lo más eficientes posible.

Cuando se utilizan ordenadores para resolver problemas numéricos, los algoritmos deben de ser programados en un lenguaje de ordenador. Actualmente existen un gran número de lenguajes informáticos que permiten realizar cálculos numéricos. Podríamos haber elegido cualquiera de ellos. En este curso nos hemos decidido por el C++ como lenguaje de cálculo numérico, que se expone en paralelo en las prácticas. Las razones que nos han motivado a ello son múltiples, y comprenden la velocidad de cálculo, la elegancia y simplicidad de su programación, la posibilidad de programar de forma interactiva, la disponibilidad del lenguaje en múltiples plataformas (procesadores y sistemas operativos), su popularidad y la calidad y número de librerías existentes así como su disponibilidad como software libre y gratuito.

1.2. Representación digital de los números

1.2.1. Sistemas de numeración

Un número entero se expresa como una cadena de dígitos

$$d_1 d_2 \dots d_k$$

en una base B. La base B queda fijada por el sistema de numeración utilizado. Los valores de B más frecuentemente empleados son 2 (sistema binario), 8 (octal), 16 (hexadecimal) y 10 (decimal). Los tres primeros sistemas son relevantes para los cálculos realizados con ordenador, mientras que el sistema decimal es el que empleamos ordinariamente los humanos. En un sistema de numeración de base B el número $d_1 d_2 \dots d_k$ corresponde a $d_k B^0 + d_{k-1} B^1 + \dots d_1 B^k$. Por ejemplo, el número 129 en base decimal es $1 \times 10^2 + 2 \times 10^1 + 9 \times 10^0$.

En la tabla 1.1 se dan los 15 primeros números en los cuatro sistemas de numeración mencionados. El sistema hexadecimal completa las cifras del 1 al 9 con las primeras 6 letras del alfabeto.

El sistema natural para los ordenadores es el binario. Esto es debido a que la memoria del ordenador consiste en dispositivos electrónicos en los que podemos pensar como celdillas, que pueden estar en uno de dos estados, denominados 1 y 0. Para el ordenador, un número es una cadena de 1 y 0. Cada celdilla (1 o 0) se denomina *bit*. Estas celdillas se agrupan en grupos de 8 bits, que se denominan *bytes*. A su vez los bytes se agrupan en *palabras*, que consisten de varios bytes. Actualmente los ordenadores utilizan en general palabras 4 u 8 bytes (32 o 64 bits), aunque los PC antiguos utilizaban 2 bytes (16 bits). La palabra es la unidad básica de

memoria. En general se utilizan una o varias palabras para representar los números. Como es más rápido la utilización de una palabra de 64 bits que dos palabras de 32 bits, las arquitecturas de procesadores de 64 bits son más adecuadas que las de 32 bits para cálculos numéricos de gran volumen de operaciones y que requieran una precisión elevada.

La ventaja de los sistemas octal y hexadecimal es su relación extremadamente simple con el sistema binario. Si tomamos un número en binario y lo agrupamos en grupos de tres dígitos y reemplazamos cada uno de estos grupos por su valor en el sistema octal, obtenemos el número en octal; Si agrupamos los dígitos de 4 en 4 y los reemplazamos por su equivalente hexadecimal, obtenemos el número en el sistema hexadecimal. Los sistemas octal y hexadecimal son más compactos y fáciles de manipular para los humanos que el sistema binario. y se utilizan para leer valores en la memoria del ordenador de una manera relativamente sencilla.

Ejemplo: Consideremos el número binario 111110000101. Agrupado en grupos de 3 dígitos es 111 110 000 101 que, consultando la tabla, tiene la expresión 7605. Si lo agrupamos en grupos de 4 digitos obtenemos 1111 1000 0101 que, consultando de nuevo la tabla, vemos que vale F85 en hexadecimal.

Cuando hay un error informático, el ordenador da un mensaje de error diciendo la dirección de memoria en notación hexadecimal en la que este error ha ocurrido.

Como los números tienen un signo algebráico, se utiliza el primer bit para represental el signo. Usualmente 1 corresponde al signo - y 0 al signo +.

Normalmente se utiliza una palabra de ordenador para representar números enteros. En un ordenador de 32 bits, quedan 31 bits disponibles para el valor del número (uno se utiliza para el signo). El mayor número entero que se puede representar es por lo tanto $2^{30}+2^{29}+2^{28}+\cdots+2^1+2^0$ que es una progresión geométrica con razón 2, cuya suma vale $2^{31}-1=2147483647$.

1.2.2. Representación de números reales

Existen diversas formas de representar un número real en la memoria del ordenador. Las principales son:

1.-*Punto flotante*. El estándar IEEE 754 representa los números reales en punto flotante en forma binaria como

$$(signo)1.d_1d_2\ldots d_k2^{e-c}$$

donde $d_1d_2\ldots d_k$ se denomina la mantisa, e recibe el nombre de *exponente almacenado*, k es un número fijo que da la precisión del número (el número de digitos de la mantisa). El número c es el *desplazamiento* y sirve para que se puedan representar exponentes tanto positivos como negativos. La diferencia $e-c$ es el *exponente*. La parte entera vale 1 y no es necesario almacenarla. Es de hecho el primer dígito significativo del número, que forzosamente vale 1.

Los números reales en *precisión simple* ($float$) se almacenan en 32 bits, que es una palabra en los procesadores usuales actualmente. Los bits se numeran de 1 a 32, y el bit 32 (el más significativo) se utiliza para denotar el signo (0 positivo y 1 negativo según el estándar). Nos quedan 31 bits disponibles a repartir entre exponente y mantisa. La convención del estándar es reservar 8 dígitos para el exponente y 23 para la mantisa. Con 8 dígitos binarios se puede contar hasta 255, por lo que este será el valor máximo del exponente almacenado. Como hay

Tabla 1.1: Primeras 15 cifras en decimal, hexadecimal, octal y binario

decimal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
hexadecimal	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
octal	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
binario	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

que considerar tanto exponentes positivos como negativos, se elige c como 127, lo que permite representar números reales entre 2^{-127} y 2^{128}, en base binaria. De hecho, los exponentes totales -127 (todos los bits del exponente a 0) y 128 (todos los bits del exponente a 1), se utilizan para propósitos especiales que veremos más adelante, por lo que los máximos exponentes utilizables son +127 y -126. El número de dígitos significativos de la mantisa es 23, que corresponde a unos 7 dígitos decimales (el entero máximo con 23 dígitos binarios es $2^{23}-1$ =8388607). El menor número representable es cuando el exponente es -126 y todos los dígitos de la mantisa son nulos, que da $\pm 2^{-126} = 1{,}17549 \times 10^{-38}$. El máximo número representable es cuando el exponente es 127 y todos los dígitos de la mantisa valen 1 (el valor de la mantisa en este caso es $2-2^{-23}$). Este máximo valor es $\pm(2-2^{-23}) \times 2^{127} = \pm 3{,}40282 \times 10^{38}$.

Ejemplo: El número $\pi = 3{,}1415926535897932$ en doble precisión, tiene como representación binaria $1{,}10010010000111111011010 \times 10^1$ con 23 bits para la mantisa. Como el exponente vale 1 el exponente almacenado es 128, que en notación binaria es 01000000. Por lo tanto la representación binaria de π es (primero signo, luego exponente y luego parte fraccionaria de la mantisa) 00100000010010010000111111011010.

En el caso de doble precisión se utilizan 64 bits para representar los números. El bit más significativo se utiliza para el signo, y los 63 bits restantes se distribuyen entre el exponente (11 bits) y la mantisa (52 bits). Con 11 bits se puede contar hasta $2^{11}-1 = 2047$. El desplazamiento se toma como 1023, con lo que podemos representar en principio exponentes entre 1024 y -1023. Los valores 1024 (todos los bits del exponente a 1) y -1023 (todos los bits a 0) se utilizan para propósitos especiales, con lo que el mayor real en doble precisión representable es $\pm(2-2^{-52}) \times 10^{1023} = 1{,}79769 \times 10^{308}$y el mínimo número representable es $\pm 2^{-1022} = 2{,}22507 \times 10^{-308}$.

Pueden existir además tipos de precisión extendida. Por ejemplo, en C/C++ existe el tipo *long double, que* no viene especificado por el standard ISO del C++ ni corresponde a un tipo de precisión definido en el estándar IEEE 754, pero las versiones recientes de gcc en procesadores de 32 bits lo toman como 4 palabras, es decir 128 bits, con 106 bits en la mantisa. La implementación del tipo *long double* se realiza mediante dos números en doble precisión cada uno con 53 dígitos de mantisa y 11 de exponente. Por lo tanto el tipo l*ong double* proporciona más precisión pero no aumenta la magnitud de los números que se pueden almacenar. De hecho los números más pequeños que se pueden almacenar son mayores que en doble precisión.

2.- *Punto flotante con precisión variable*. Esta aritmética no se utiliza como aritmética nativa de procesadores, sino solo en librerías especializadas. Los números se representan en esta aritmética como

$$(signo)0.d_1 d_2 \ldots d_k B^e$$

En este caso k es variable, y su valor máximo solo está limitado por los parámetros de la librería y la memoria disponible, aparte de por el tiempo de procesador disponible. En general su límite superior es un número N del orden de varios miles, que depende de la librería utilizada. Un paquete de precisión arbitraria sencillo y flexible *es mpfun,* que se expone en las prácticas.

3.- *Punto fijo*. Es el método que se utilizaba en ordenadores sin procesador de coma flotante. Los dígitos se dividen arbitrariamente en dos grupos, uno para la parte entera y otro para la mantisa:

$$(signo)d_1d_2\ldots d_k.d_{k+1}\ldots d_n \quad n\,fijo.$$

El error cometido en esta representación es $0{,}00000_{n-k+1}1 = 1\times 2^{-(n-k)}$.

4.- *Aritmética de intervalos*. Es otra aritmética utilizada en librerías. En este tipo de aritmética, los números se representan con su error:

$$(signo)0.d_1d_2\ldots d_n \pm r10^e = (signo)(0.d_1d_2\ldots d_n \pm 0{,}00\ldots 0_{n-1}r)\times 10^e$$

De esta manera se pueden especificar los errores que afecten a los números en cada etapa del cálculo. Las reglas de determinación de errores de los resultados de las operaciones matemáticas son similares a las utilizadas en el cálculo de errores:

$$\begin{aligned}
(m_1\pm\varepsilon_1)\pm(m_2\pm\varepsilon_2) &= (m_1\pm m_2)\pm(\varepsilon_1+\varepsilon_2)\\
(m_1\pm\varepsilon_1)\times(m_2\pm\varepsilon_2) &= m_1\times m_2\pm(\varepsilon_1\times|m_2|+|m_1|\times\varepsilon_2+\varepsilon_1\times\varepsilon_2)\\
\frac{m_1\pm\varepsilon_1}{m_2\pm\varepsilon_2} &= \frac{m_1}{m_2}\pm\frac{\varepsilon_1+\left|\frac{m_1}{m_2}\right|\varepsilon_2}{|m_2|-\varepsilon_2} \quad si \quad |m_2|>\varepsilon_2\\
\frac{m_1\pm\varepsilon_1}{m_2\pm\varepsilon_2} &= error \quad si \quad |m_2|\leq\varepsilon_2
\end{aligned}$$

Este tipo de aritmética es importante en el caso de errores de redondeo incontrolados, como sucede en la resolución de ciertas ecuaciones diferenciales y problemas numéricos mal condicionados. Actualmente existen varias librerías que implementan este tipo de aritmética.

1.3. Precisión en el cálculo

Aunque los datos de un programa informático sean constantes exactas, cada operación matemática introduce un error en el cálculo que se propaga y puede ser muy elevado en operaciones iterativas. Es importante conocer la precisión con que se realiza un cálculo. Si tomamos una serie lentamente convergente y calculamos un millón de términos con una precisión de 10^{-6}, el error de la suma será próximo a la unidad. Hay problemas que son intrínsecamente díficiles de resolver desde el punto de vista de precisión. Incluso utilizando los algoritmos más adecuados, en estos problemas se acumulan rápidamente errores de redondeo. Este es el caso de la inversión de determinadas matrices y problemas asociados. En este caso, la solución es utilizar mayor precisión para resolver el problema. En C++ tenemos los tipos nativos $float$, $double$ y $long\ double$. En los procesadores de 32 bits $double$ es suficiente para un amplio espectro de problemas numéricos. Actualmente los procesadores de 64 bits tienen precios muy asequibles. Estos procesadores son

especialmente adecuados para cálculo numérico, con compiladores nativos que admiten hasta cuádruple precisión, es decir cuatro palabras de ordenador por número real. Sin embargo en los problemas más patológicos incluso esta precisión no es suficiente. En C++ se han desarrollado librerías de precisión arbitraria, en las que los números se representan en base 128, utilizando la tabla de caracteres completa como dígitos de la base; es decir, los números son cadenas de carácteres. De esta manera es posible codificar un número con miles e incluso millones de cifras decimales. El empleo de este tipo de librerías es especialmente cómodo en C++, ya que los números de precisión arbitraria pueden ser tratados como un tipo adicional de números reales, análogo a *double* por ejemplo (esta ventaja viene proporcionada por la posibilidad de definir *clases* en C++, equivalentes a los tipos nativos). El paso conflictivo desde el punto de vista de precisión numérica en un problema dado se puede resolver con números de precisión arbitraria, para luego continuar la solución del problema en una precisión nativa que consume menos recursos. Vamos a tratar brevemente el origen de los problemas de errores. Los valores que se pueden representar cuando se trabaja con una precisión dada es un conjunto discreto. Por ejemplo el número inmediatamente superior al cero es en precisión simple $2^{-126} = 1{,}17549435 \times 10^{-38}$ que corresponde a un exponente binario de -11111110[1] y a una mantisa binaria de 10000000000000000000000. Todos los números reales entre 0 y este número tienen que ser representados por uno de los dos. Si el resutado de un problema es por ejemplo 7×10^{-39} podemos tomarlo como 0, que es el número representable inmediatamente inferior en valor absoluto (esto se denomina truncado) o como $1{,}175494 \times 10^{-38}$, que es el número representable nmediatamente superior, y que además está más próximo al resultado (esto de denomina redondeo). Dicho de otra manera, si podemos representar k cifras decimales de un número dado a, y este número tiene más de k cifras, si tomamos el número representable con los primeros k dígitos, hemos realizado un truncado. Si por el contrario, consideramos el dígito d_{k+1} de la mantisa, de forma que si es mayor que 5 añadimos 1 a d_k y si es menor que 5 dejamos d_k como estaba, hemos realizado un redondeo. El redondeo y el truncado coinciden cuando $d_k \leq 5$. El redondeo es obviamente más preciso que el truncado y es el procedimiento que se utiliza actualmente. Los errores de redondeo pueden acumularse en procesos iterativos, produciendo resultados erróneos.

1.3.1. Overflows y underflows

El estándar IEEE 754 reserva una serie de valores especiales de los campos de exponente y mantisa. Uno de ellos es para representar el valor 0, que no se puede representar directamente debido a la suposición de que el primer dígito de la mantisa es 1. El cero se representa con todos los bits del exponente y de la parte fracionaria de la mantisa nulos. Hay un +0 y un -0, dependiendo de que el bit del signo sea 0 o 1, aunque ambos valores son equivalentes. Otro valor especial corresponde a infinito, también llamado *overflow*, representado por los símbolos *Infinity, infty o inf*, que se representa con todos los bits del exponente a 1 y todos los de la mantisa nulos. Dependiendo del bit del signo se distinguen entre *+Infinity* y *-Infinity*. Estos números especiales sirven para definir el resultado de una operación que es mayor en valor absoluto que cualquier número representable. Finalmente está el valor *Not a Number* (representado por *NaN*

[1] El exponente -11111111 se utiliza para representar underflows

Tabla 1.2: Operaciones con valores especiales

Operación	Resultado
$n \div \pm Infinity$	$\pm Infinity$
$\pm Infinity \times \pm Infinity$	$\pm Infinity$
$\pm n \div 0$	$\pm Infinity$
$+Infinity + Infinity$	$+Infinity$
$\pm 0 \div \pm 0$	NaN
$+Infinity - Infinity$	NaN
$\pm Infinity \div \pm Infinity$	NaN
$\pm Infinity \times 0$	NaN

o nan) que sirve para etiquetar los resultados operaciones ilegales (raíces cuadradas de números negativos, arcosenos con aggumento superior a la unidad, etc.). Se representa con todos los bits del exponente a 1 y valores no nulos de los bits de la mantisa, que se utilizan para definir varias clases de NaN (clasificados en las categorías de QNaN para operaciones indeterminadas y SNaN para operaciones inválidas). Por último, si todos los bits del exponente son nulos, pero los de la mantisa no lo son, se supone que el bit principal no es 1 como en el caso ordinario, y se llama a estos números *números no normalizados*. Los números no normalizados representan valores de $(-1)^s \times 0.f \times 2^{-126}$ en precisión simple y $(-1)^s \times 0.f \times 2^{-1022}$ en precisión doble, donde s es el bit del signo y f la parte fraccionaria de la mantisa. Son por lo tanto números más pequeños en valor absoluto que cualquier número normalizado representable y se utilizan para representar *underflows*. Se puede pensar en ellos como un tipo de 0.

Cuando el programa produce como resultado de un cálculo un número que no puede representar porque resultado no está definido o la operación es ilegal, asigna al resultado uno de los valores especiales anteriores, dando mensajes como *NaN, overflow o underflow,* y seguidamente da un error de ejecución, que puede acabar, dependiendo de la severidad del error y del compilador, en un aborto de la ejecución del programa. Errores de este estilo son producidos, por ejemplo, por una llamada a $sqrt(-5)$ o $asin(3)$, que dan como resultado NaN. El compilador suele dejar libertad al usuario sobre como tratar los errores de ejecución a través de opciones en el caso de valores no representables (NaN), números reales mayores que el máximo representable (*overflow*) o menores que el mínimo representable en valor absoluto (*underflow*). El compilador g++, sin ninguna opción sobre el tratamiento de valores especiales, se limita a propagar los resultados de operaciones con valores especiales de acuerdo con la rabla 1.2, sin abortar la ejecución. El caso menos severo es el de *underflow*, en el que valores menores que el mínimo número representable se pueden reemplazar por 0, continuando los cálculos. El caso de *overflow* es más severo, pero existe la posibilidad de reemplazarlos siempre por el máximo número representable. Si ese número es x_m, tendríamos que, por ejemplo, $x_m * x_m = x_m$, desde el punto de vista del ordenador. La tabla 1.2 resume la forma en que el ordenador trata operaciones que implican valores especiales.

Los *underflows* y *overflows*, en general, conducen a soluciones erróneas, salvo en casos donde un estudio previo permita asegurar que el resultado no es erróneo. Los problemas numéricos

deben de programarse de forma que se eviten operaciones que produzcan resultados no representables.

1.4. Lenguajes de programación

Con el advenimiento de los ordenadores, se hizo necesario la creación de lenguajes de programación para codificar los problemas de cálculo numérico. El lenguaje natural de los ordenadores es el binario, en el que tanto órdenes como datos se especifican como series de bits (0 o 1). En el comienzo de la era de los ordenadores se utilizaba el *ensamblador*, que es un lenguaje máquina mnemotécnico. Posteriormente, se desarrollaron lenguajes de ordenador mucho más simples, adaptados a las aplicaciones particulares a las que estaban destinados. El lenguaje de programación FORTRAN (FORmula TRANslation) se diseñó como un lenguaje destinado a resolver problemas de cálculo numérico. Las versiones modernas del FORTRAN son todavía las más eficientes en lo que a velocidad de cálculo se refiere, aunque la diferencia con el C++ no es importante. El compilador de un lenguaje de ordenador realiza una conversión de una orden del lenguaje en una serie de órdenes en lenguaje máquina. Un mismo cálculo se puede realizar con un número diferente de órdenes. Por ejemplo si calculamos $ab + ac$ podemos realizar dos multiplicaciones y una suma o podemos reordenar el cálculo como $a(b + c)$ con lo cual realizaremos una suma y una multiplicación. El proceso de reordenar los cálculos de forma que se realice el menor número de operaciones posible se llama *optimización*, y en la medida en que el compilador de un lenguaje informático pueda realizar procesos complicados de optimización, dicho lenguaje es adecuado para el cálculo numérico.

Los principales inconvenientes del FORTRAN77 son, en primer lugar, su incapacidad de ajustar dinámicamente la memoria requerida, lo que obliga a resevar la memoria máxima requerida durante toda la ejecución del programa, o a introducir funciones en ensamblador para gestionar dinámicamente la memoria[2]. En segundo lugar, el FORTRAN77 carece de posibilidades de interaccionar con el hardware. En los años 70 surgieron una serie de lenguajes que permitían un ajuste dinámico de memoria, como el PASCAL y el C. También surgieron nuevos sistemas operativos, y el hecho de que el popular sistema UNIX estuviera escrito en C motivó sin duda el que el lenguaje C se impusiera rápidamente como lenguaje de programación. Sin embargo, hasta finales de los años 80 los compiladores de C no eran capaces de realizar una optimización satisfactoria, lo que originaba que el lenguaje C estuviera caracterizado por una lentitud inherente en la resolución de problemas numéricos. A principios de los años 90, debido a los progresos realizados en los compiladores, el lenguaje C empezó a aparecer como una opción razonable para programar problemas numéricos. El lenguaje C++ apareció a mediados de los 80 como una adaptación del C que permitía la creación de nuevos tipos, junto con características tales colo la herencia y la orientación a objetos. C++ es lo que se llama un lenguaje de alto nivel.

El aumento de velocidad de procesador, memoria y capacidad de disco duro de los ordenadores en los últimos dos decenios, ha posibilitado que un modesto ordenador personal sea capaz de

[2]Actualmente existen versiones recientes del FORTRAN, como FORTRAN90 y FORTRAN95, que incorporan conceptos como programación orientada a objetos, punteros y otras conceptos del C/C++, pero desde el punto de vista de velocidad de cálculo son comparables al C++.

realizar tareas que hace 20 años estaban reservadas a los grandes ordenadores. Esto ha hecho posible el desarrollo de nuevos conceptos informáticos, como la programación orientada a objetos (OOP), que facilita enormemente la interacción de los programas con el usuario. La gestión de periféricos mediante programas de ordenador se ha generalizado en los últimos tiempos, lo que hace necesario que los lenguajes de ordenador puedan interaccionar fácilmente con el sistema operativo, frecuentemente en tiempo real. El cálculo numérico se ha convertido en una herramienta necesaria que forma parte integrante de dichos programas interactivos. Por ejemplo, en los programas gráficos en los que marcando tres puntos hace pasar una curva Bezier por ellos, se resuelve en realidad un problema de interpolación. Dado que los métodos numéricos se suelen emplear un número repetido de veces, el interés en economizar tiempo y memoria de CPU en la resolución de cálculos numéricos no ha disminuido, a pesar del aumento de los recursos informáticos disponibles.

Los lenguajes tipo UNIX son los más empleados en entornos donde la velocidad, eficiencia, estabilidad y seguridad son indispensables. La versión LINUX del UNIX está siendo utilizada cada vez con más frecuencia como sistema operativo de base en grandes centros de cálculo. El sistema UNIX está programado en C, por lo que el lenguaje C es el más conveniente cuando se desea interaccionar con este sistema operativo (OS). Como el lenguaje C++ es una extensión del C orientada a objetos, los programas en C son en principio compatibles con el compilador C++ (todos los compiladores C y C++ no son necesariamente estándar, lo que puede producir incompatibilidades en casos concretos). Esto hace que el lenguaje C++ sea el lenguaje orientado a objetos más popular (se han vendido 500.000 ejemplares de los textos más populares) y en cualquier caso el más práctico cuando se necesita interaccionar con el OS. El OS Windows está programado en C++, lo que genera un fuerte interés por este lenguaje fuera del entorno UNIX. Además, existen compiladores públicos en C++ para prácticamente todos los OS de interés, y los compiladores comerciales son más baratos para C++ que para los otros lenguajes. Para FORTRAN por ejemplo y en entorno Windows, existen muy pocos compiladores, a precios exorbitantes con frecuencia. Es por estas razones que hemos elegido C++ como lenguaje de este curso. En realidad, el cálculo numérico se puede codificar en cualquiera de los lenguajes existentes, ya que sus únicos requerimientos esenciales son los operadores aritméticos +, -, *, / y las relaciones lógicas, =, $\leq\geq$, <, >, y $\neq$, junto con la posiblididad e repetir de forma automática una operación y de decidir entre dos posibilidades. Todos los lenguajes existentes (mencionemos los más populares: FORTRAN, COBOL, BASIC, ALGOL, PASCAL, ADA, SIMULA2, C, C++, Java y Python entre ellos) proporcionan estos requerimientos. Sin embargo no todos están optimizados. Aunque las primeras versiones del C y C++ eran ineficientes para el cálculo numérico, los compiladores actuales tienen rendimientos próximos al FORTRAN y en algunos casos superiores. El lenguaje JAVA apareció como un lenguaje puramente orientado a objetos, que careciese del inconveniente del C++ de estar construído sobre un lenguaje de bajo nivel. Sin embargo, JAVA no es todavía hoy en día una solución aceptable en programas donde la eficiencia numérica es crítica. Para quien está interesado en resolver problemas numéricos en los que la velocidad de procesado no es un requerimiento crítico, existen lenguajes interpretados (es decir, no se compilan) que son cómodos de utilizar. El primero de ellos fue el BASIC, similar al FORTRAN. Hoy en día son especialmente populares en el ámbito de las aplicaciones matemáticas y numéricas los programas comerciales MATLAB (y su equivalente en software público con licencia GNU,

OCTAVE) y *Mathematica*. Los lenguajes *Python* y *Ruby* son otros lenguaje interpretados que está ganando popularidad. JAVA puede interpretarse o compilarse. Todos los lenguajes orientados a objetos interpretados guardan muchos aspectos que recuerdan al C y al C++, por lo que aprenderlos es relativamente simple si se conoce el C++.

1.5. Problemas solubles e insolubles

Hay problemas que tienen una dificultad intrínseca. Por ejemplo, el decidir si dos números reales son iguales. Aunque se verifique que las primeras n cifras decimales son iguales, la siguiente puede ser distinta y el cálculo nunca puede considerarse acabado. Como mucho, podemos preguntarnos si las primeras n cifras son iguales. Pero incluso en este caso, hay que tener en cuenta los errores de redondeo. Este problema sucede también con las relaciones aritméticas $=, <, >, \leq, \geq, \neq$ aplicadas a números reales. En lo que concierne la igualdad de a y b, solo podemos decidir si $a \in [b-10^{-n}, b+10^{-n}]$ para un valor de n fijado por la precisión con la que se trabaja; la manera de verificar esta relación es peguntar si se cumple la condición $|a-b| < 10^{-n}$. Son problemas en principio insolubles, los siguientes, donde a y b son números reales:

1. Decidir si $a = b$ o $a \neq b$.
2. Decidir si $a \Re b$ o no, donde $\Re$ es una de las relaciones $>, <, \geq, \leq$.
3. Seleccionar la única relación correcta entre $a < b, a = b, a > b$.
4. Decidir entre $a = 0$ y $a \neq 0$.
5. Decidir entre $a \geq 0$ y $a \leq 0$.

Aunque un problema sea soluble, puede ser que sea imposible de resolver con los ordenadores y algoritmos disponibles, bien a causa de la ineficiencia de los algoritmos o de la inmensidad del problema. En general, siempre es posible resolver casos simplificados de un problema soluble.

La programación de algunos conceptos matemáticos usuales conduce a problemas insolubles, como por ejemplo la función signo:

$$\text{sig (x)} = \begin{cases} +1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

Si $x = 0$, el programa entra en un bucle sin fin (siempre verifica si la siguiente cifra es nula).

En la práctica, como el número de cifras con las que se representan los números en el ordenador no es infinito, la introducción de elementos no calculables en un programa no conduce a un bucle infinito. Sin embargo, el resultado de las decisiones depende en general de errores de redondeo que afectan a la última cifra, y del número de bits de una palabra del ordenador. Como estos errores de redondeo dependen del compilador y del procesador, los elementos no calculables conducen a resultados del programa dependientes de la plataforma en la que se ejecute, lo que es en cualquier caso una característica altamente indeseable.

1.6. Algoritmos. Eficiencia y robustez.

Un algoritmo es el conjunto de operaciones con las que se resuelve un problema numérico. Dado un problema concreto, se puede resolver en principio de muchas formas diferentes. Un algoritmo determinado puede ser muy rápido en comparación con otros algoritmos concebidos para resolver el mismo problema. Entonces se dice que dicho algoritmo es *eficiente*. Un algoritmo puede funcionar en unos casos y fallar en otros. Un algoritmo que funciona en todos o asi todos los casos se dice que es *robusto*. El encontrar algoritmos eficientes y/o robustos es uno de los objetivos del Cálculo Numérico. Como un ejemplo concreto, consideremos la evaluación de un polinomio. Sea $P(x)$ un polinomio dado por

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Si lo evaluamos directamente, lo que en C ++ se codificaría como

$$P = a[n] * pow(x,n) + a[n-1] * pow(x,n-1) + a[1] * x + a[0]$$

tendríamos que realizar n sumas y

$$n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

multiplicaciones. Las multiplicaciones necesitan mucho más tiempo de procesador que las sumas; es por esto que el número de multiplicaciones define el orden de un algoritmo. Decimos por lo tanto que el algoritmo utilizado para evaluar el polinomio e forma directa es de orden n^2. Sin embargo, podemos redisponer el polinomio de las siguiente forma, conocida como *algoritmo de Horner*:

$$P(x) = (((\ldots(a_n(x + a_{n-1})x + a_{n-2})x + \ldots a_1)x + a_0)$$

Reordenado el polinomio de este modo, sólo hacen falta n multiplicaciones. El algoritmo de Horner es, por lo tanto, de orden n. Una programación ineficiente conduce inevitablemente a programas que requieren para su procesamiento un tiempo de CPU mucho mayor del necesario. Los compiladores de C++ realizan reordenaciones similares a las del algoritmo de Horner cuando se especifican opciones de optimización (g++ -On, con n=1,2,3,4). Sin embargo, estas opciones dificultan a veces la búsqueda de errores en un programa (se pierde la información sobre las variables optimizadas cuando se utiliza un debugger o buscador de errores), y nunca se sabe hasta que punto el compilador realiza una optimización total, por lo que es conveniente codificar los algoritmos directamente de forma optimizada.