RESUMEN DEL CURSO DE METODOS NUMERICOS

impartido por

Virginia Muto Foresi

Departamento de Matemática Aplicada
y Estadística e Investigación Operativa

Facultad de Ciencia y Tecnología

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Los capitulos que siguen constituyen una versión resumida del texto de la autora Virginia Muto Foresi, publicado por el Servicio Editorial de la Universidad del País Vasco, UPV/EHU, con titulo **Curso de Métodos Numéricos** e I.S.B.N. 84-8373-062-6, cuyos índices se detallan a continuación.

CURSO DE METODOS NUMERICOS — INDICE

PRIMERA PARTE: INTRODUCCION AL ANALISIS NUMERICO Y A LA COMPUTACION

SEGUNDA PARTE: **SOLUCION APROXIMADA DE ECUACIONES DE UNA VARIABLE**

CURSO DE METODOS NUMERICOS

PRIMERA PARTE

INTRODUCCION AL ANALISIS NUMERICO

Y A LA COMPUTACION

CAPITULO I. INTRODUCCION AL ANALISIS NUMERICO

1. ALGORITMOS Y DIAGRAMAS DE FLUJO

El hecho de que el Análisis Numérico sea tanto una ciencia como un arte es la opinión de los especialistas en este campo pero, frecuentemente es mal entendido por los no especialistas. ¿Se dice que es un arte, y a la vez una ciencia, únicamente como eufemismo, para ocultar el hecho de que el Análisis Numérico no es una disciplina suficientemente precisa para merecer el que se le considere como una ciencia? ¿Es cierto que el nombre de *análisis numérico* se emplea erróneamente, porque el significado clásico del análisis en matemáticas no es aplicable al trabajo numérico? De hecho, la respuesta a ambas preguntas es "no". Más bien la yuxtaposición de ciencia y arte se debe a un principio de incertidumbre que frecuentemente se presenta en la solución de problemas, es decir, el hecho de que para determinar la mejor forma de resolver un problema, puede ser necesaria la solución del problema en sí. En otros casos, la mejor forma de resolver un problema puede depender de un conocimiento de las propiedades de las funciones involucradas, las que no se pueden obtener ni teórica ni prácticamente.

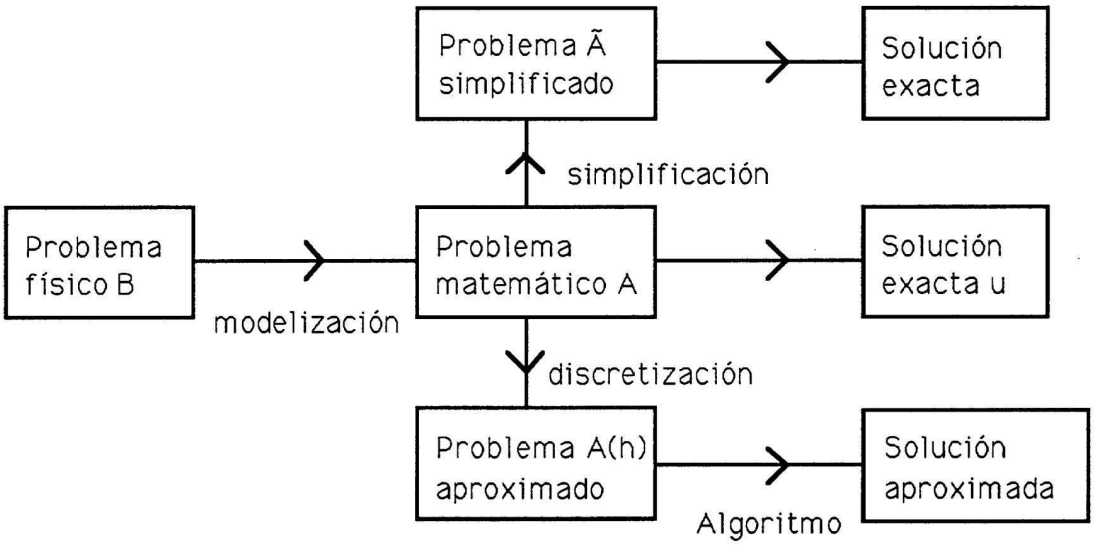
Como una ciencia, el Análisis Numérico está interesado en los procesos por los cuales pueden resolverse los problemas matemáticos, por las operaciones de la aritmética. Algunas veces esto involucrará el desarrollo de algoritmos para resolver un problema que está ya en una forma en la cual pueda encontrarse la solución por medio aritméticos. Frecuentemente involucrará la necesidad de sustituir cantidades que no pueden ser calculadas aritméticamente, por aproximaciones que permiten que sea determinada una solución aproximada. En este caso estaríamos interesados, naturalmente, en los errores cometidos en nuestra aproximación. Pero, en cualquier caso, las herramientas que usaríamos en el desarrollo de los procesos de análisis numérico, serán las herramientas del análisis matemático exacto, tan conocidas clásicamente.

Como un arte, el Análisis Numérico está interesado en la elección del procedimiento, y conveniente aplicación del mismo, "más" adecuado a la solución de un problema particular. Esto implica la necesidad de desarrollar la experiencia y con ello esperar que se desarrolle la intuición del especialista.

Así pues, el Análisis Numérico trata de diseñar métodos para aproximar, de una manera eficiente, las soluciones de problemas expresados matemáticamente. La eficiencia del método depende tanto de la precisión que se requiera como de la facilidad con la que pueda implementarse. En una situación práctica, el problema matemático se deriva de un fenómeno físico sobre el cual se han hecho algunas suposiciones para simplificarlo y para poderlo representar matemáticamente. Generalmente cuando se relajan las suposiciones físicas llegamos a un modelo matemático más apropiado pero, al mismo tiempo, más difícil o imposible de resolver explícitamente.

Ya que normalmente el problema matemático no resuelve el problema físico exactamente, resulta con frecuencia más apropiado encontrar una solución aproximada del modelo matemático más complicado que encontrar una solución exacta del modelo sim-

plificado. Para obtener tal aproximación se idea un método llamado **algoritmo**. El algoritmo consiste en una secuencia de operaciones algebráicas y lógicas que permiten la aproximación al problema matemático y se espera que también al problema físico, con una tolerancia o precisión predeterminada. Los algoritmos determinan los métodos constructivos de resolución de problema matemáticos. Un **método constructivo** es todo proceso que permite obtener la solución a un problema con la precisión que se desee, en un número finito de pasos que se pueden efectuar racionalmente. Obviamente el número de pasos requeridos dependerá de la precisión que se desee en la solución.



Como se ha dicho, los métodos constructivos en matemáticas son métodos que muestran cómo construir soluciones de un problema matemático. Por ejemplo, una demostración constructiva de la existencia de una solución de un problema, no sólo hace ver que la solución existe, si no que describe también cómo se puede determinar esa solución. Una demostración que muestra la existencia de una solución por *reducción al absurdo* no es constructiva.

Los algoritmos tienen que satisfacer los siguientes requisitos:

a) *generalidad*: un algoritmo se tiene que poder aplicar a cualquier conjunto de datos que pertenezcan a un dominio establecido;
b) *finitud*: un algoritmo tiene que estar constituído por una sucesión de instrucciones que pueden ser ejecutadas por el ordenador un número finito de veces;
c) *no ambigüedad*: un algoritmo no tiene que estar constituído por instrucciones que se contradigan o que lleguen a una paradoja.

Los valores sobre los cuales operan las instrucciones de un lenguaje de programación para producir nuevos valores pueden ser:

a) *numéricos*;
b) *lógicos* (True, False);
c) *alfanuméricos*.

Nótese que los valores lógicos dados en el apartado b) son los dos únicos existentes.

Con reglas adecuadas los operadores actuán sobre las variables y las constantes para obtener valores nuevos. Una serie de símbolos usados para indicar los operadores se da en la tabla 1. Y en la tabla 2 se dan los resultados de los operadores *and*, *or* y *xor* de las variables lógicas.

Tabla 1

Simbolo	Tipo de valor del resultado	Operación
$+$	numérico	suma
$-$	numérico	resta
$*$	numérico	multiplicación
$**$	numérico	exponenciación
$/$	numérico	división
$[\,]$	numérico	parte entera
$\sum$	numérico	suma finita
$=$	lógico	igualdad
$\neq$	lógico	no igualdad
$<$	lógico	menor que
$>$	lógico	mayor que
$\leq$	lógico	menor o igual que
$\geq$	lógico	mayor o igual que
not	lógico	cambio de T (F) en F (T)
and	lógico	(a la vez)
or	lógico	(o bien)
xor	lógico	(o bien exclusivo)

Tabla 2

A	B	not A	A and B	A or B	A xor B
T	T	F	T	T	F
T	F	F	F	T	T
F	T	T	F	T	T
F	F	T	F	F	F

La mayoría de las veces, los métodos de tipo constructivo directos dan lugar a algoritmos finitos, mientras que los métodos iterativos producen algoritmos infinitos (convergentes).

Un ejemplo clásico de **algoritmo finito** lo constituye el algoritmo de Euclides para el cálculo del *máximo común divisor* (m.c.d.) de dos números. Sean a, b dos números enteros, $a > b$. Entonces:

$$
\begin{aligned}
a &= b * q_1 + r_2, && 0 < r_2 < b \\
b &= r_2 * q_2 + r_3, && 0 < r_3 < r_2 \\
r_2 &= r_3 * q_3 + r_4, && 0 < r_4 < r_3 \\
&\ldots. \\
r_{m-2} &= r_{m-1} * q_{m-1} + r_m, && 0 < r_m < r_{m-1} \\
r_{m-1} &= r_m * q_m \ .
\end{aligned}
$$

Entonces $r_m = \text{m.c.d.}(a, b)$.
El algoritmo sería:

- hacer $r_0 = \text{máx}\ (a, b)$ y $r_1 = \text{mín}\ (a, b)$
- hallar $r_n =$ resto de dividir r_{n-2} entre r_{n-1}, $n = 2, 3, ...$
- cuando $r_n = 0$, parar: el m.c.d. es r_{n-1}.

Un método para el cálculo de $\sqrt{N}$ es el siguiente:

- hacer $x_0 = \frac{1+N}{2}$
- $x_{n+1} = \frac{1}{2}\left[x_n + \frac{N}{x_n}\right], \qquad n = 0, 1, ...$,

entonces $\lim_{n\to\infty} x_n = \sqrt{N}$.

Es evidente que este método es infinito (la sucesión x_{n+1} no se hace constante porque $\sqrt{N}$ no es racional en la mayor parte de los casos), por lo que debemos indicar un criterio de parada para que el **algoritmo iterativo** sea efectivo. Usualmente el criterio es:

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

donde ε es la tolerancia permitida.

Si el algoritmo es visto como una serie temporal de operaciones, una pregunta fundamental es ¿cómo viene controlado el flujo de las operaciones? Cuando el programa en ejecución ha llegado a una instrucción particular ¿cómo determina el ordenador cuál es la próxima instrucción que tiene que ejecutar?

Se ha demostrado que sólo tres principios de control son suficientes para describir cualquier algoritmo.
El primer principio es la **noción de secuencia**; excepto que el ordenador sea instruido distintamente, él ejecuta las instrucciones de un programa secuencialmente.
El segundo principio es la **ejecución condicional** que se indica generalmente en el programa con una instrucción del tipo "*If ... then*" (si ... entonces). En la instrucción *if B then S*, B es una expresión boleana, que puede producir sólo los valores verdadero o falso, y S es una instrucción cualquiera o grupo de instrucciones. Se evalúa B y se ejecuta S sólo si el resultado es verdadero.
El tercer principio es la **repetición** que puede ser indicado con una instrucción "*While ... do*" (mientras ... ejecuta). *While B do S* examina el valor de B y, si es verdadero, ejecuta S: los dos pasos se repiten hasta que una evaluación de B produce el valor falso. En la mayoría de los casos una evaluación de S determina el cambio del valor de B, de manera que el ciclo no continúe para siempre. Otra manera para indicar la repetición es el bucle *Do ... var = vari, varf, vars S continue*. El *Do ... continue* repite las instrucciones del bloque *S* para los valores de la variable *var* desde *vari* hasta *varf* con paso *vars*.

En cada lenguaje de programación, los valores sobre los cuales operan las instrucciones son las constantes y las variables (numéricas, lógicas o alfanuméricas). Además, las instrucciones fundamentales, que se pueden individualizar con un nombre o con un número (llamado *dirección*), son de los tipos siguientes:

a) **instrucciones de asignación**, que permiten asignar el valor de una expresión a una variable;

b) **instrucciones de salto incondicional**, que permiten interrumpir el orden normal de ejecución de las instrucciones de un algoritmo;
c) **instrucciones de condición**, que comparando dos valores, condicionan la ejecución de unas instrucciones en lugar de otras;
d) **instrucciones de transmisión**, que permiten transferir valores entre el mundo externo y el ordenador;
e) **instrucciones de principio de ejecución y de fin de ejecución**, que comandan el inicio o fin de la ejecución de instrucciones del algoritmo.

Como se ha dicho antes, excepto las instrucciones de salto incondicional, todas las otras se ejecutan en el orden en el cual están escritas, y la ejecución de una instrucción no empieza hasta que no haya acabado la ejecución de la instrucción previa.

La estructura de un algoritmo se puede representar gráficamente con un diagrama dinámico de líneas que conectan sucesiones de instrucciones del algoritmo. Cada una de esa sucesión de instrucciones es incluida en una figura y las líneas indican la interconexión entre las sucesiones. Conviene dar forma distinta a las figuras dependiendo del tipo de instrucciones que contenga. El diagrama dinámico así realizado se llama **diagrama de flujo** (flow chart).

A menudo es conveniente que un problema caracterizado por un algoritmo A sea dividido en un número finito de problemas más sencillos, llamados subrutinas (**subroutines**). Uno de los motivos principales por los cuales es conveniente efectuar esa división en problemas más sencillos es que si se necesita resolver el mismo problema en más de un lugar del algoritmo principal, con diferentes datos, no es muy eficiente repetir las mismas instrucciones que tienen sólo nombres distintos por las variables sobre las cuales operan. Sin embargo, es más conveniente escribir un algoritmo separado que resuelva el problema parcial con **datos formales**, y organizar el problema principal originario de manera que las partes distintas se conectan a la subroutine, transmitiendo los **datos actuales**.

2. ORIGEN Y EVOLUCION DEL ANALISIS NUMERICO

Debido a la estrecha relación existente entre las diferentes ramas de la Ciencia (y en particular de las Matemáticas), no es fácil determinar dónde acaba una y empieza otra. Por ello la extensión exacta del Análisis Numérico no es conocida. De hecho, el concepto de **Análisis Numérico** no fue creado hasta 1947 en que se fundó el Instituto de Análisis Numérico en la Universidad de California. Sin embargo, el nombre parece estar asociado a aquellos temas que requieran un *procesamientos de datos*. Como la extensión de estos temas es considerable (puede ir, por ejemplo, desde la interpretación de datos médicos hasta la reserva automática de plazas de avión o gestión de una biblioteca), nos limitaremos a ciertos aspectos matemáticos de la idea.

Al principio, la mayor parte del trabajo que se efectuaba en el campo de las **Matemáticas**, inspirado por cuestiones y problemas concretos, se basaba en métodos constructivos para determinar la solución (predicciones sobre eclipses, aparición de un cometa, etc...).

El punto culminante de la utilización de los algoritmos está en Euler (1707–1783), que en los 70 volúmenes que comprenden sus trabajos incluye gran número de algoritmos y fórmulas. Los algoritmos infinitos que presenta, aparecen, normalmente, como desarrollos en serie.

Posteriormente, la perfección de los conocimientos matemáticos y la generalización de los problemas hacen que se sustituyan los razonamientos constructivos por otros de tipo lógico. Así, interesa más determinar si existe la solución a un determinado problema, que calcularlo de forma efectiva. Este proceso sigue hasta aproximadamente el año 1950. La razón del proceso de abstracción era que los algoritmos para el cálculo de las soluciones de los problemas eran, aunque finitos, irrealizables por la gran cantidad de cálculos que exigían. A partir de la segunda mitad del siglo XX, la aparición de las computadoras liberan al algoritmo de la pesadez del cálculo, lo que supone un nuevo auge para los métodos constructivos. Podríamos decir que si desde la antigüedad hasta 1945 la velocidad de cálculo se había multiplicado por 10 mediante rudimentarios artefactos (como el ábaco), desde entonces hasta ahora se ha multiplicado por un millón o más. Esto supone que 1 hora de trabajo de ordenador equivale a 200 años de trabajo de una persona, lo que permite realizar tareas inalcanzables en otros tiempos. Esto no significa que todos los algoritmos puedan ser tratados por un ordenador, pues algunos exigen más de 100 años de trabajo del ordenador actual más potente para poder ser llevados a cabo.

Como la eficiencia de un método depende de su facilidad de implementación, la elección del método apropiado para aproximar la solución de un problema está influenciada significativamente por los cambios tecnológicos en calculadoras y computadoras. El factor limitante en la actualidad es generalmente la capacidad de almacenamiento de la computadora, a pesar de que el costo asociado con los tiempos de cómputo es, desde luego, también un factor importante.

3. OBJETIVOS

El *Análisis Numérico* es *Matemática Aplicada* en el sentido de que toca problemas concretos, reales, de aplicación práctica, pero aprovechando los potentes métodos de la Matemática Pura. Por tanto no son materias opuestas, sino complementarias, lo que hace que la importancia de ambas sea cada vez mayor.

Algunos de los problemas que toca el Análisis Numérico son los siguientes:

a) **Problemas de interpolación**, en los que se sustituye una función poco manejable por otra más sencilla que cumple ciertas condiciones de coincidencia con la primera;
b) Problemas derivados de los anteriores, como pueden ser la **integración aproximada** (cuadratura, cubatura), o **derivación aproximada** de funciones poco manejables;
c) **Problemas de aproximación**, análogos a los anteriores, pero en los que se sustituye una función por otra que sea "próxima", en cierto sentido, a la primera;
d) **Resolución aproximada de ecuaciones diferenciales** tanto ordinarias como en derivadas parciales;

e) Los problemas presentados anteriormente producen, en muchos casos, **sistemas de ecuaciones lineales** con gran número de ecuaciones e incógnitas que por su coste de cálculo son irresolubles por métodos clásicos como la regla de Cramer;

f) **Problemas de tipo matricial**, (hallar valores propios, invertir matrices, etc...) relacionados con los anteriores;

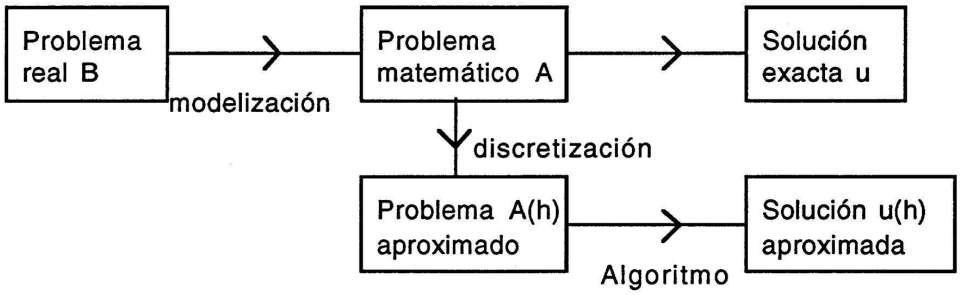
g) **Problemas de optimización**, en los que se maximiza o se minimiza un funcional;

h) **Resolución aproximada de ecuaciones algebráicas y sistemas de ecuaciones no lineales**.

CAPITULO II. ANALISIS DE LOS ERRORES

1. ESQUEMA DE RESOLUCION NUMERICA DE UN PROBLEMA

Si se desea resolver un problema físico B, lo primero que se suele hacer es traducirlo al lenguaje matemático para dar un problema matemático A. Se estudia la existencia y unicidad de la solución u de este problema, pero en la mayor parte de los casos y después de probado esto, no se sabe cómo determinar la solución de forma efectiva. Por ello, se sustituye el problema matemático A por un problema proximo a él, A_h, en el que aparecerá algún parámetro h que se va a hacer tender hacia un cierto valor (normalmente 0). Se exige que este problema tenga solución única, u_h, y se espera que al tender h hacia el valor elegido, u_h converja hacia u. Esquemáticamente este tratamiento típico (pero no único), es el siguiente:



De este planteamiento surgen algunos problemas interesantes:

a) ¿Cuál es la **velocidad de convergencia** de u_h hacia u?

b) **Problemas de estabilidad**; es inevitable cometer errores en el cálculo, debido a los redondeos que efectúan los computadores. Interesa que pequeños errores cometidos en los cálculos que conducen a u_h hagan que el resultado no difiera mucho de u; (de eso hablaremos más en el siguiente párrafo).

c) **Coste del proceso**. ¿Cuántas operaciones deben realizarse? ¿Cuánto tiempo se precisará para realizarlas?

Veamos ahora unos ejemplos que muestran la importancia de esas últimas cuestiones.

A. Supongamos que se necesita evaluar el polinomio

$$p(x) = a_0\ x^n + a_1\ x^{n-1} + ... + a_{n-1}\ x + a_n\ ,$$

que es equivalente a:

$$p(x) = ((...((a_0\ x + a_1)\ x + a_2)\ x + ... + a_{n-1})\ x + a_n)\ .$$

El número de operaciones para evaluarlo en el primer caso es de:

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

n Sumas,

mientras que en el segundo se requieren solamente

n Multiplicaciones

n Sumas.

Se comprende pues, que es preferible usar el segundo método porque exige menos operaciones (y por lo tanto existen menos posibilidades de que se propaguen los errores de redondeo, lo que dará lugar a una solución más exacta). El algoritmo que lleva a evaluar el polinomio con el segundo método se denomina **algoritmo de Horner** y es:

$b_0 = a_0$

$b_i = a_i + b_{i-1}\ x, i = 1, ..., n$.

B. Para resolver sistemas de ecuaciones de orden n con el método de Cramer se precisa un total de $(n+1)!\ (n-1)$ operaciones (multiplicaciones) (cada determinante exige $n!\ (n-1)$ multiplicaciones $a_{p(1)}\ a_{p(2)}...a_{p(n)}$ y hay $n+1$ determinantes a calcular).

El método de Gauss (que explicaremos en un capítulo posterior) exige, sin embargo, sólo $\frac{n^3}{3}$ operaciones. Así, una tabla comparativa de estos métodos sería:

n	5	10	50	100
Cramer	2880	$3.99*10^6$	$7*10^{67}$	10^{160}
Gauss	42	334	42000	334000

Por ejemplo, para $n = 5$, haciendo una operación cada medio minuto (manualmente) se tardarían 24 horas en resolver el sistema por el método de Cramer, mientras que por el de Gauss se tardarían sólo 21 minutos.

Si se intentase utilizar el método de Cramer para resolver un sistema de orden 15 en un ordenador que efectuase 10^6 operaciones por segundo, tardaría más de 9 años en obtener la solución, que además posiblemente no se parecería en nada a la solución verdadera debido a los errores de redondeo que se hubieran producido. ¡Con el método de Gauss, el mismo ordenador tardaría centésimas de segundo!

Este último ejemplo justifica suficientemente la necesidad de buscar algoritmos que sean prácticos.

2. DISTINTOS TIPOS DE ERRORES

Generalmente el resultado de un cálculo numérico es aproximado (sólo en casos excepcionales es exacto), y por eso necesitamos conocer la precisión.

Si p y p^* son dos números reales y p^* se considera como aproximación de p, una medida de la precisión de p^* es

$$E = |p - p^*| \ .$$

De costumbre el conocimiento de E no basta para establecer si p^* es una aproximación buena de p. Por ejemplo:

$p_1 = 5.1346$, $p_1^* = 5.1345$

$E = |p_1 - p_1^*| = 10^{-4}$,

y

$p_2 = 0.0005$, $p_2^* = 0.0004$
$E = |p_2 - p_2^*| = 10^{-4}$.

En los dos casos E es igual a 10^{-4}, pero sólo en el primer caso pensamos que p_1^* es una buena aproximación de p_1. En el segundo caso, p_2 y E son del mismo orden de magnitud, y entonces nos parece mejor considerar su razón.

Damos entonces la siguiente definición: si p^* es una aproximación de p, el **error absoluto** está dado por $E_a = |p - p^*|$, y el **error relativo** está dado por $E_r = \frac{|p-p^*|}{|p|}$, siempre y cuando $p \neq 0$.

Muchas son las causas que pueden interferir en la precisión de un cálculo, y generar errores. Esos errores se pueden clasificar en:

a) errores iniciales;
b) errores de redondeo;
c) errores de truncamiento;
d) errores de propagación.

Los **errores iniciales** no se pueden evitar si, por ejemplo, son el resultado de medidas de precisión limitada. Supongamos que debemos calcular $f(x)$ en un cierto punto x. Puede ocurrir que estemos obligados a sustituir x por x', con lo cual se calculará $f(x')$ en vez de $f(x)$. Se llama error inicial al valor $f(x') - f(x) = \varepsilon_i$.

Los **errores de redondeo** son debidos a redondeos en los cálculos porque están hechos con un número finito de cifras significativas. Entonces, y continuando con el ejemplo previo, no calcularemos $f(x')$ sino $f_1(x')$. El valor $f_1(x') - f(x') = \varepsilon_r$ se llama error de redondeo.

Los **errores de truncamiento** generalmente corresponden a truncamientos de procedimientos infinitos (desarrollos en serie, etc.). En el ejemplo previo puede ocurrir que f (y f_1) sea poco manejable y estamos obligados a sustituirla por otra función próxima a ella, f_2. El valor $f_2(x') - f_1(x') = \varepsilon_t$ es llamado error de truncamiento o de discretización.

Aquí es útil, por ejemplo, recordar el **Teorema de Taylor**: supongamos que $f \in C^n[a,b]$ y $f^{(n+1)}$ existe en [a, b). Sea $x_0 \in [a,b]$. Para toda $x \in [a,b]$, existe $\xi(x)$ entre x_0 y x tal que

$$f(x) = P_n(x) + R_n(x)$$

donde

$$P_n(x) = f(x_0) + f'(x_0)\ (x - x_0) + \frac{f''(x_0)}{2!}\ (x - x_0)^2 + ... + \frac{f^{(n)}(x_0)}{n!}\ (x - x_0)^n$$

$$= \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}\ (x - x_0)^k$$

y

$$R_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}\ (x - x_0)^{(n+1)} .$$

A $P_n(x)$ se le llama el polinomio de Taylor de grado n para f alrededor de x_0 y a $R_n(x)$ se le llama el residuo (o error de truncamiento) asociado con $P_n(x)$. La serie infinita que se obtiene tomando el límite de $P_n(x)$ cuando $n \to \infty$ se denomina Serie de Taylor para f alrededor de x_0. En el caso de que $x_0 = 0$, el polinomio de Taylor se conoce frecuentemente como polinomio de MacLaurin, y la serie de Taylor se denomina serie de MacLaurin.

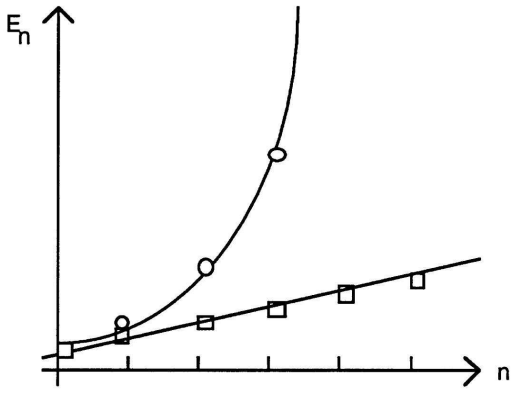
Los **errores de propagación** son debidos a la propagación de errores previos en el algoritmo.

3. CONVERGENCIA

Hemos dicho ya que los cálculos que involucran aproximaciones en la máquina pueden resultar en el crecimiento de los errores de redondeo. Por supuesto, estamos interesados en escoger métodos que produzcan resultados fiables en su precisión. Un criterio que impondremos en un algoritmo, cuando sea posible, es que cambios pequeños en los datos iniciales produzcan correspondientemente cambios pequeños en los resultados finales. Un algoritmo que satisfece esta propriedad se llama **estable**. Es **inestable** cuando este criterio no se cumple. Algunos algoritmos serán estables para ciertos grupos de datos iniciales pero no para todos. Se tratará, siempre que se pueda, de caracterizar las propiedades de estabilidad de los algoritmos.

Para considerar un poco más el tema del crecimiento del error de redondeo y su conexión con la estabilidad de los algoritmos, supongamos que se introduce un error ε en alguna etapa de los cálculos y que el error después de n operaciones subsecuentes se denota por E_n. Los dos casos que se presentan más frecuentemente en la práctica se definen a continuación.

Definición. Supongamos que E_n representa el crecimiento del error después de n operaciones subsecuentes. Si $|E_n| \approx C\ n\ \varepsilon$, donde C es una constante independiente de n, se dice que el crecimiento del error es **lineal**. Si $|E_n| \approx k^n\ \varepsilon$, para algún $k > 1$, el crecimiento del error es **exponencial**.



El crecimiento lineal del error es usualmente inevitable, y cuando C y ε son pequeños los resultados son generalmente aceptables. El crecimiento exponencial del error debe ser

evitado, ya que el término k^n será grande aún para valores pequeños de n. Esto lleva a imprecisiones inaceptables, no importando la magnitud de ε. Como consecuencia, un algoritmo que exhibe crecimiento lineal del error es estable, mientras que un algoritmo en el que el crecimiento del error es exponencial es inestable.

Como ejemplo consideremos la sucesión $p_n = (\frac{1}{3})^n,\ n > 0$, que puede generarse recursivamente tomando $p_0 = 1$ y definiendo $p_n = (\frac{1}{3})\ p_{n-1}$, para $n > 1$. Si obtenemos la sucesión de esta manera, usando aritmética de redondeo a cinco dígitos, los resultados vienen dados en la tabla 1.
El error de redondeo introducido en reemplazar $\frac{1}{3}$ por 0.33333 produce un error de sólo $(0.33333)^n \times 10^{-5}$ en el n-ésimo término de la sucesión. Este método de generar la sucesión es claramente estable.

Tabla 1

n	p_n
0	0.10000×10^1
1	0.33333×10^0
2	0.11111×10^0
3	0.37036×10^{-1}
4	0.12345×10^{-1}

Otra manera de generar la sucesión es definiendo $p_0 = 1$, $p_1 = \frac{1}{3}$, y calculando para cada $n \geq 2$,

$$p_n = (\frac{10}{3})\ p_{n-1} - p_{n-2}\ .$$

La tabla 2 muestra los resultados tanto exactos como redondeados a cinco dígitos usando esta fórmula.

Tabla 2

n	p_n calculado	p_n exacto
0	0.10000×10^1	0.10000×10^1
1	0.33333×10^0	0.33333×10^0
2	0.11111×10^0	0.11111×10^0
3	0.37000×10^{-1}	0.37037×10^{-1}
4	0.12230×10^{-1}	0.12346×10^{-1}
5	0.37660×10^{-2}	0.41152×10^{-2}
6	0.32300×10^{-3}	0.13717×10^{-2}
7	-0.26893×10^{-2}	0.45725×10^{-3}
8	-0.92872×10^{-2}	0.15242×10^{-3}

Este método es obviamente inestable.

Nótese que la fórmula dada, $p_n = (\frac{10}{3})\ p_{n-1} - p_{n-2}$, se satisface si p_n es de la forma

$$p_n = C_1(\frac{1}{3})^n + C_2 3^n$$

para cualquier par de constantes C_1 y C_2. Para verificar esto, notemos que

$$\begin{aligned}\frac{10}{3}\ p_{n-1} - p_{n-2} =& \frac{10}{3}[C_1(\frac{1}{3})^{n-1} + C_2 3^{n-1}] - [C_1(\frac{1}{3})^{n-2} + C_2 3^{n-2}] \\ =& C_1[\frac{10}{3}(\frac{1}{3})^{n-1} - (\frac{1}{3})^{n-2}] + C_2[\frac{10}{3}3^{n-1} - 3^{n-2}] \\ =& C_1(\frac{1}{3})^n + C_2 3^n = p_n\ .\end{aligned}$$

Para tener $p_0 = 1$ y $p_1 = \frac{1}{3}$, las constantes C_1 y C_2 deben escogerse como $C_1 = 1$ y $C_2 = 0$. Sin embargo, en la aproximación de cinco dígitos, los dos primeros términos son $p_0 = 0.10000 \times 10^1$ y $p_1 = 0.33333 \times 10^0$, los cuales requieren una modificación de estas constantes a $C_1 = 0.10000 \times 10^1$ y $C_2 = -0.12500 \times 10{-}5$. Este pequeño cambio en C_2 da lugar a un error de redondeo de $3^n(-0.12500 \times 10{-}5)$ al producir p_n. Como consecuencia resulta un crecimiento exponencial del error, lo cual se refleja en la pérdida extrema de exactitud encontrada en la tabla 2.

Para reducir los efectos del error de redondeo, podemos usar una aritmética de un orden grande de dígitos, como las opciones de doble o múltiple precisión, disponibles en la mayoria de las computadoras digitales. Una desventaja del uso de la aritmética de doble precisión es que toma mucho más tiempo de computadora. Por otro lado, no se elimina completamente el crecimiento serio del error de redondeo, sino que sólo se postpone si es que se realizan un gran número de cálculos posteriores. Hay también otros métodos para estimar el error de redondeo (aritmética de intervalo, métodos estadísticos, etc.) que no estudiaremos.

CAPITULO III. SISTEMAS DE NUMERACION

1. REPRESENTACION DE LA INFORMACION

El sistema de numeración usado habitualmente es el decimal, de base 10, que no es adecuado para ser manejado por el ordenador, fundamentalmente porque es más sencillo construir un elemento con únicamente dos posibles estados que uno con 10 estados. Y por otra parte como los componentes electrónicos envejecen es más dificil mantener correctamente un dispositivo con 10 estados que uno con dos.

Una información dada al ordenador es, generalmente, representada con una sucesión de caracteres escogidos desde un alfabeto compuesto sólo de dos caracteres, representados, respectivamente, por los simbolos "0" y "1", llamados cifras binarias o **bits** (binary digits).

Cada uno de los caracteres es físicamente representado por uno de los posibles estados de los componentes del ordenador: un núcleo magnético es imanado en una de las dos posibles direcciones de magnetización, un circuito puede ser abierto o cerrado.

El problema de representar los caracteres de un alfabeto hecho con más de dos caracteres se resuelve uniendo más cifras binarias; por ejemplo, los agrupamientos hechos con dos cifras binarias (00, 01, 10, 11) dan la posibilidad de distinguir cuatro caracteres diversos, los obtenidos con tres cifras binarias se pueden usar para distinguir ocho caracteres distintos, y en general los agrupamientos obtenidos con n cifras binarias pueden representar 2^n caracteres distintos.

El usuario de un ordenador no tiene que conocer necesariamente la representación de los datos en la máquina, porque los lenguajes comunes de programación permiten especificar datos e instrucciones con los caracteres y las cifras usadas comunmente por el hombre, y son los compiladores los que convierten al sistema de representación propio del ordenador.

Sin embargo, para la solución de muchos problemas es conveniente que el usuario de un ordenador conozca el sistema binario y los principios fundamentales de la aritmética de un ordenador, aunque nunca contará en la aritmética binaria.

2. INTRODUCCION A LOS SISTEMAS NUMERICOS

La representación común de los números es constituida por sucesiones de los símbolos "$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$". Tales sucesiones pueden ser precedidas por los símbolos "+" y "-" (para indicar un número positivo o negativo), y pueden tener el símbolo "." (**punto raíz**, que separa la parte entera del número, a la izquierda, de su parte decimal, a la derecha). Nuestro sistema decimal es un sistema **posicional**, es decir cada cifra tiene un *peso*. La posición ocupada por cada dígito tiene un significado exacto y determina la contribución de la cifra al valor numérico de la sucesión.

Por ejemplo, 35 y 53 están constituidos por las mismas cifras 3 y 5, pero tienen significados distintos.

Cada cifra de la sucesión es multiplicada por una potencia de 10, con el exponente determinado por la posición de la cifra con respecto al punto raíz. El valor 10 es la **base**

de nuestro sistema de numeración, que por esta razón se denomina **sistema posicional en base diez**. Los exponentes de la parte entera son positivos y crecen en unidades a partir de cero, que corresponde al exponente de la potencia de 10 que multiplica la cifra más a la derecha de la parte entera, los exponentes de la parte fraccionaria del número son negativos y disminuyen en unidades a partir de -1, que corresponde al exponente de la potencia de 10 que multiplica la primera cifra de la parte fraccionaria.

Un ejemplo de sistema no posicional lo constituye la numeración romana. En ella, al valor 5 le corresponde el símbolo V, mientras que al valor 50 le corresponde el símbolo L. Para pasar de 5 a 50 no basta con cambiar la posición del símbolo 5 (V), sino que hay que introducir uno nuevo (L).

La descripción hecha del sistema posicional decimal sugiere la posibilidad de usar un sistema de numeración en una base distinta de 10. Con el sistema en base 10 se usan 10 cifras para representar cada número; en general, para representar un número en una base b, se necesitan b símbolos. Entonces, cuando se considera como base un entero $b > 10$, las 10 cifras del sistema decimal no bastan y es necesario usar símbolos nuevos.

Las bases más usadas, además de la 10, son 2, 8 y 16. El sistema en base dos, llamado **sistema binario**, usa las cifras "0, 1". El sistema en base ocho, llamado **sistema octal**, usa las cifras "0, 1, 2, 3, 4, 5, 6, 7". Y el **sistema hexadecimal**, en base 16, usa las cifras "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F".

3. CONVERSION DESDE EL SISTEMA DECIMAL AL SISTEMA NUMERICO EN BASE b

La existencia de sistemas de numeración en base distintas de 10, hace que nos planteemos el problema de la representación de los números en una nueva base, y la definición de las reglas y de las propiedades formales de las operaciones a ejecutar en la nueva aritmética.

Supongamos que conocemos la representación decimal de un número cualquiera entero positivo, y queremos construir la correspondiente representación en base b. Cada entero n se expresa en la base b en la forma

$$n = d_p * b^p + d_{p-1} * b^{p-1} + ... + d_1 * b^1 + d_0 * b^0 \ , \qquad (III.1)$$

donde p y $d_p, d_{p-1}, ..., d_1, d_0$ son enteros que tenemos que determinar para obtener la representación

$$n_b = d_p \ d_{p-1} \ ... \ d_1 \ d_0 \ .$$

La ecuación (III.1) se puede escribir de la forma

$$n = b * (d_p * b^{p-1} + d_{p-1} * b^{p-2} + ... + d_1) + d_0,$$

por lo cual se deduce que d_0 es el resto de la división de n entre b. Si denotamos con n_1 el cociente de esa división, tenemos

$$n_1 = d_p * b^{p-1} + d_{p-1} * b^{p-2} + ... + d_1$$

que podemos escribir

$$n_1 = b * (d_p * b^{p-2} + d_{p-1} * b^{p-3} + ... + d_2) + d_1.$$

Entonces, se deduce que d_1 es el resto de dividir n_1 entre b. Si denotamos con n_2 el cociente de esa división, tenemos

$$n_2 = d_p * b^{p-2} + d_{p-1} * b^{p-3} + ... + d_2$$

que podemos escribir como

$$n_2 = b * (d_p * b^{p-3} + d_{p-1} * b^{p-4} + ... + d_3) + d_2.$$

Entonces, se deduce que d_2 es el resto de dividir n_2 entre b. Procediendo de manera parecida se llega a determinar d_{p-1} como el resto de la división de n_{p-1} entre b, y donde

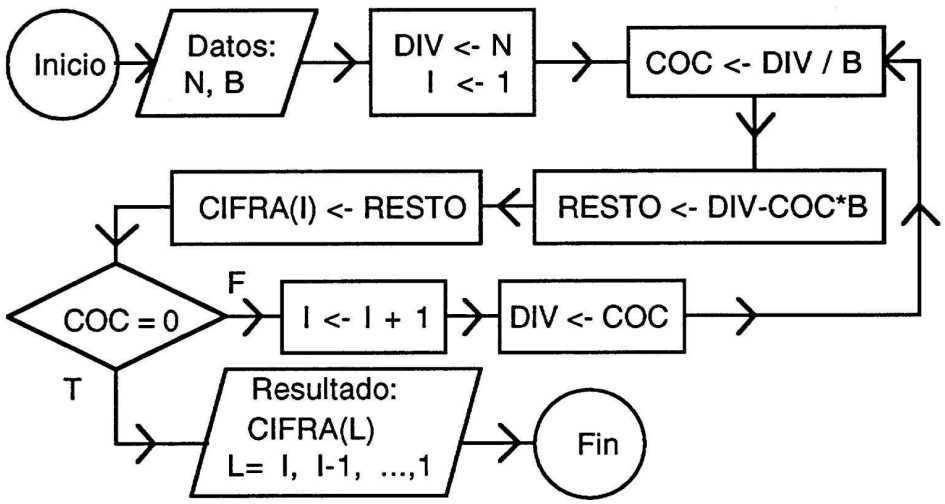
$$n_{p-1} = b * d_p + d_{p-1} \ .$$

Entonces, d_p es el cociente de la última división.

Dado que el valor de p no es conocido, y no se conoce el número de veces que tenemos que dividir entre b, el último cociente d_p se determina de esa manera (conocida como el **método de las divisiones sucesivas**), como el resto de la división por b que da un cociente cero.

En la figura 1 está representado el diagrama de flujo para el método de las divisiones sucesivas. En ese diagrama, N es el número entero para el cual queremos la representación en base B, $CIFRA$ es el vector con las cifras en la representación en base B, COC es el cociente de la división del dividendo DIV entre la base B. $RESTO$ es el resto de la división, e I es un índice con el cual se cuentan las cifras en la representación en base B.

Figura 1



Siguiendo el esquema de la figura 1, es fácil verificar que

$$1972_{10} = 11110110100_2 = 3664_8 = 7B4_{16}.$$

Construyamos, como ejemplo, la representación en **base 8**:

$1972/8 = 246$ resto 4
$246/8 = 30$ resto 6
$30/8 = 3$ resto 6
$3/8 = 0$ resto 3

Entonces $1972_{10} = 3664_8$, y viceversa:

$$3664_8 = 3 * 8^3 + 6 * 8^2 + 6 * 8^1 + 4 * 8^0$$
$$= 1536_{10} + 384_{10} + 48_{10} + 4_{10} = 1972_{10}$$

Nótese que el primer resto obtenido en las divisiones es la cifra que tendrá su posición a la inmediata izquierda del punto raíz, así como el último resto obtenido es la cifra que tendrá su posición más a la izquierda del punto raíz.

Consideremos ahora el problema de construir la representación en base b de un número z real y positivo, menor que 1, del cual conocemos la representación decimal. Cada número menor que 1 se expresa en la base b en la forma:

$$z = q_1 * b^{-1} + q_2 * b^{-2} + q_3 * b^{-3} + ... \qquad (III.2)$$

donde $q_1, q_2, q_3, ...$ son enteros que tenemos que determinar para crear la representación

$$z_b = 0.\ q_1\ q_2\ q_3....$$

Supongamos que realizamos la multiplicación de z por b; entonces, de (III.2) obtenemos:

$$z * b = q_1 + q_2 * b^{-1} + q_3 * b^{-2} + ... = q_1 + z_1$$

por lo cual se deduce que q_1 es la parte entera de $z * b$, es decir

$$q_1 = [z * b],$$

y que z_1 es un número menor que 1, obtenido como diferencia del producto $z * b$ menos su parte entera. Del producto $z_1 * b = q_2 + z_2$ se deduce que

$$q_2 = [z_1 * b].$$

De manera análoga se sigue que

$$q_3 = [z_2 * b]$$

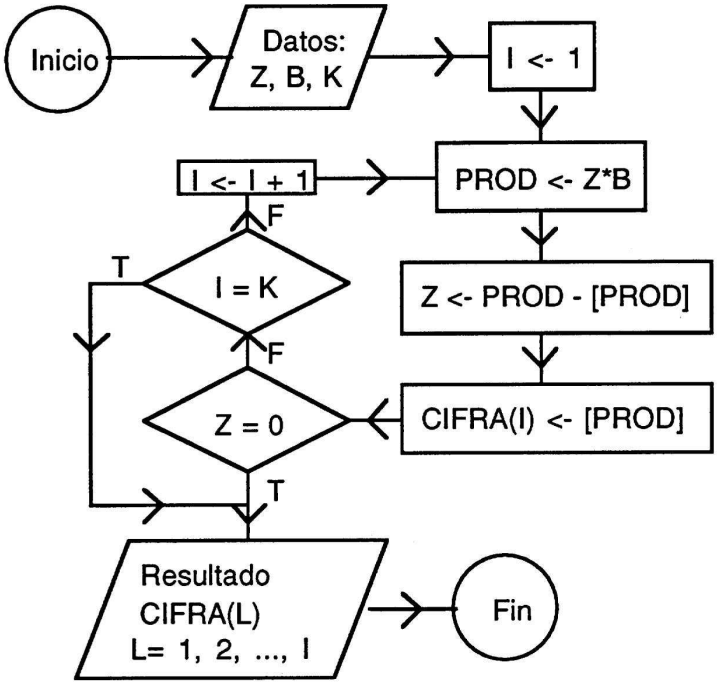
$$....$$

$$q_i = [z_{i-1} * b].$$

Este método es conocido con el nombre de **método de las multiplicaciones sucesivas**.

En la figura 2 está representado el diagrama de flujo para el método de las multiplicaciones sucesivas. En ese diagrama, Z es el número real positivo menor que 1, para el cual queremos la representación en base B, con no más de K cifras. $CIFRA$ es el vector con las cifras en la representación en base B, y $[PROD]$ es la parte entera del $PROD$.

Figura 2



Siguiendo el esquema de figura 2, es fácil verificar que

$$0.828125_{10} = 0.110101_2 = 0.65_8 = 0.D4_{16} \ .$$

Construyamos, como ejemplo, la representación en **base 8**:

$0.828125 * 8 = 6.625$ parte entera 6

$0.625 * 8 = 5.0$ parte entera 5

$0.0 * 8 = 0.0$

Entonces $0.828125_{10} = 0.65_8$, y viceversa:

$$0.65_8 = 6 * 8^{-1} + 5 * 8^{-2}$$
$$= 6 * 0.125_{10} + 5 * 0.015625_{10} = 0.828125_{10}$$

Nótese que la primera parte entera obtenida en las multiplicaciones es la cifra que tendrá su posición más a la inmediata derecha del punto raíz, así como la última parte entera obtenida es la cifra que tendrá su posición más a la derecha del punto raíz.

Si aplicamos el método de las multiplicaciones sucesivas para construir la representación en base 2 del número $z_{10} = 0.1_{10}$, resulta:

$$0.1_{10} = 0.000110011..._2 = 0.0\overline{0011}_2 \ .$$

Con ese ejemplo, hemos demostrado que un mismo número puede tener, en una base, una representación con un número finito de cifras, mientras que en otra base, puede tener una representación con un número infinito de cifras.

4. LAS OPERACIONES ARITMETICAS EN BASE b

Supongamos que tenemos que calcular una determinada operación aritmética. Una manera de proceder es convertir los sumandos en base 10, ejecutar la suma, y después convertir el resultado en base 2. Ese método, sin embargo, no siempre es conveniente, sobre todo si pensamos que para las operaciones aritméticas de los números en base b valen las mismas reglas y propiedades formales conocidas en la aritmética en base 10. En el sistema de numeración binario existen sólo dos símbolos: 0_2 y 1_2. Entonces, cuando se efectua la operación $1_2 + 1_2$, no tenemos un único símbolo para representar el resultado, y el resultado es 0 con el reporte de 1, es decir 10_2.

Las reglas para efectuar la suma y la multiplicación de dos cifras binarias están resumidas en la tabla 1.

Tabla 1

+	0	1	*	0	1
0	0	1	0	0	0
1	1	10	1	0	1

Damos también las tablas de la suma y la multiplicación en base 8 (tabla 2). De manera parecida se pueden construir las tablas en base 16.

Tabla 2

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

*	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

5. CONVERSION DESDE UN SISTEMA NUMERICO EN BASE b_1 A UN SISTEMA EN BASE b_2

El problema de la conversión de la representación de un mismo número real (no cero) desde una base b_1 a otra base b_2 se puede resolver de distintos modos.

Primer método. Una manera es convertir la representación del número, primero desde la base b_1 a la base 10, y después desde la base 10 a la nueva base b_2. Por ejemplo, para convertir el número X que en base 2 tiene la representación -11101.101_2 desde la base $b_1 = 2$ a la base $b_2 = 8$, se puede proceder de la manera siguiente. Construyamos la representación decimal de X, que se obtiene expresando X en la forma

$$X = -(1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-3}) = -29.625_{10} \ ,$$

y luego se determina la representación de X en la base $b_2 = 8$ en la manera que ya conocemos, obteniendo

$$-11101.101_2 = -29.625_{10} = -35.5_8 \ .$$

Segundo método. Otra manera para obtener la conversión de las representaciones de X desde la base b_1 a la base b_2, no usa la representación decimal intermedia, sino aplica directamente los algoritmos usados para la conversión de un número decimal a la base b, con la condición que las operaciones sean ejecutadas en base b_1.

Por ejemplo, si queremos representar en base $b_2 = 8$ el número X que en base $b_1 = 2$ tiene la representación -11101.101_2, se procede de la forma siguente. Se convierte antes la parte entera del número desde la base 2 a la base 8, recordando que $b_2 = 8$ tiene la representación 1000_2, en base 2. Entonces,

$11101_2/1000_2 = 11_2$ resto 101_2

$11_2/1000_2 = 0$ resto 11_2 .

Dado que $101_2 = 5_8$ y $11_2 = 3_8$, obtenemos $11101_2 = 35_8$. Después se convierte desde base 2 a la base 8 la parte fraccionaria de X, obteniendo

$0.101_2 * 1000_2 = 101.0_2$ parte entera 101_2

$0.000_2 * 1000_2 = 0.0_2$

Dado que $101_2 = 5_8$, obtenemos $0.101_2 = 0.5_8$, y, concluyendo, resulta que

$$-11101.101_2 = -35.5_8 \ .$$

Tercer método. Otro método para obtener la conversión de un número X desde la base b_1 a la base b_2 es aplicable cuando $b_2 = b_1^k$, siendo k un entero mayor o igual que 2. La conversión se obtiene entonces de la manera siguiente. Sea n un entero positivo del cual tenemos la representación en base b_1. Entonces, descomponemos el alineamiento de los caracteres en agrupamientos, cada uno de k caracteres, partiendo de la derecha hacia la izquierda. Si el agrupamiento más a la izquierda tiene menos de k caracteres se añaden ceros para obtener un agrupamiento de k caracteres. Dado que la tabla:

$$
\begin{array}{c}
\overset{k}{} \\
000\ ...\ 000_{b_1} = 0_{b_1^k} \\
000\ ...\ 001_{b_1} = 1_{b_1^k} \\
..... \\
111\ ...\ 111_{b_1} = (b_1^k - 1)_{b_1^k}
\end{array}
$$

da la representación en base b_1 de los enteros $0, 1, ..., (b_1^k - 1)$ representados en base b_1^k, se asocia a cada agrupamiento el correspondiente entero representado en base b_1^k, y el nuevo alineamiento es la representación en base b_1^k de n.

Las tablas que dan las correspondencias entre las bases 2 y 8 y las bases 2 y 16, permiten la inmediata conversión de las representaciones de un número entre las bases 2, 8 y 16. Por ejemplo:

$$11101.101_2 = 011\ \ 101.101_2 = 35.5_8$$
$$11101.101_2 = 0001\ \ 1101.1010_2 = 1D.A_{16}$$

Y viceversa, si tenemos la representación del número X en base $b_2 = b_1^k$, obtenemos su representación en base b_1, sustituyendo cada cifra representada en base b_1^k, con el correspondiente agrupamiento de k caracteres obtenido de la tablas de correspondencia. Por ejemplo:

$$5FF.FB1_{16} = 0101\ \ 1111\ \ 1111.1111\ \ 1011\ \ 0001_2 = 10111111111.11111011_2\ .$$

Para convertir un número desde su representación octal a su representación hexadecimal, se ejecuta un proceso con dos pasos. Antes el número octal se convierte a binario, y después los dígitos binarios se dividen en agrupamientos de cuatro dígitos y convertidos a las cifras hexadecimales. Por ejemplo:

$$14057_8 = 001\ \ 100\ \ 000\ \ 101\ \ 111_2 = 0001\ \ 1000\ \ 0010\ \ 1111_2 = 182F_{16}\ .$$

Un procedimiento analogo se usa si se quiere convertir un número hexadecimal a su representación octal. Antes se escribe el número en su representación hexadecimal y se convierten los dígitos a los correspondientes agrupamientos de cuatro cifras binarias. Después esos dígitos binarios se dividen en agrupamientos de tres dígitos, y esos agrupamientos se escriben en cifras octales. Por ejemplo:

$$
\begin{aligned}
1F34B_{16} &= 0001\ \ 1111\ \ 0011\ \ 0100\ \ 1011_2 = \\
&= 011\ \ 111\ \ 001\ \ 101\ \ 001\ \ 011_2 = 371513_8\ .
\end{aligned}
$$

CAPITULO IV. ARITMETICA DEL COMPUTADOR

1. REPRESENTACION DE LOS NUMEROS

Cuando se usa una calculadora o una computadora digital para realizar cálculos numéricos, se debe considerar un error inevitable, el llamado error de redondeo. Este error se origina porque la aritmética realizada en una máquina involucra números con sólo un número finito de dígitos, con el resultado de que muchos cálculos se realizan con representaciones aproximadas de los números verdaderos.

El ordenador recibe, normalmente, información en decimal, que es transformada a binario por un programa interno. Posteriormente efectúa las operaciones pertinentes, pasa el resultado a decimal e informa al usuario de este resultado.

Así pues, en principio deberíamos hablar de las representación de los números en binario (la forma usual de trabajar del ordenador), pero para facilitar la comprensión, usaremos la representación decimal.

La representación de los números en el sistema decimal no es única (considerar que $0.999... = 1.000...$). Esto también es cierto para otros sistemas de numeración, y en particular para el sistema binario. Para evitar estas ambiguedades, siempre nos referiremos a la representación finita (1 en vez de 0.999...).

En general los ordenadores digitales trabajan con un número fijo (finito) de posiciones, **la longitud de palabra**, cuando representan un número internamente. Esta longitud n depende de la máquina, y además algunas permiten extensiones a múltiplos enteros de n ($2n$, $3n$, ...) que posibilitan una mayor exactitud si se necesita. Una palabra de longitud n se puede utilizar de distintas formas para representar un número:

- la **representación de punto fijo** especifica un número fijo n_1 de lugares enteros, y un número fijo n_2 de decimales, de modo que $n = n_1 + n_2$. En esta representación, la posición del punto decimal está fija y son pocos los dispositivos que la utilizan (ciertas máquinas de calcular, o máquinas de tipo comercial).

- Más importante, sobre todo en el cálculo cientifico, es la **representación en punto flotante**. La posición del punto decimal con respecto al primer dígito se expresa con un número separado, denominado exponente. Así se obtiene la notación científica:

$$x = a * b^t, \qquad \text{con} \qquad |a| < 1, \qquad b \in \mathcal{N}, \qquad t \in \mathcal{Z}$$

donde b es la **base** del sistema de numeración, t es un exponente llamado **característica** y a se llama la **mantisa**. Además si $|a| \geq b^{-1}$, es decir que el primer dígito despues del punto raiz no es cero, se dice que la representación es **normalizada**.

Naturalmente, en un ordenador digital sólo se dispone de un número finito de posiciones para representar un número (n, la longitud de la palabra), por lo que cada ordenador tendrá reservadas m posiciones para la mantisa y c posiciones para la característica ($n = m + c$). Por ejemplo, sea $m = 5$, $c = 2$ ($\Rightarrow n = 7$), y $b = 10$. El número 5420.0 se representaría:

$$0.54200 * 10^4 \longrightarrow \underline{| \ 5 \ 4 \ 2 \ 0 \ 0 \ | \ 0 \ 4 \ |} \ .$$

Esta notación no es única porque ese número se podría haber expresado también como:

$$0.05420 * 10^5 \longrightarrow \underline{| \ 0 \ 5 \ 4 \ 2 \ 0 \ | \ 0 \ 5 \ |} \ .$$

La primera notación es la representación normalizada.

Son **dígitos significativos** de un número todos los dígitos de la mantisa sin contar los primeros ceros.

Los números m y c, junto con la base b de la representación de un número, determinan un conjunto $\mathcal{A} \subset \mathcal{R}$ de números reales que se pueden representar de forma exacta en una máquina; estos números se denominan **números de la máquina**. Como ejemplo, imaginemos que una computadora pueda representar exactamente el número decimal 179.015625 , y que el siguiente número de máquina más pequeño sea 179.015609 , mientras que el número de máquina más grande siguiente es 179.015640 . Esto significa que nuestro número de máquina original debe representar no solamente a 179.015625, sino a un número infinito de números reales que estén entre este número y su número de máquina más cercano.

Hay diversos conceptos, relacionados con estas representaciones, que describiremos a continuación: truncamiento, redondeo, underflow y overflow.

Los dos primeros conceptos, **truncamiento** y **redondeo**, son relativos a la forma de representar los números que no pertenecen al conjunto $\mathcal{A}$ definido anteriormente. Supongamos que tenemos una máquina con m dígitos de mantisa y c dígitos de característica. Ya sabemos que el conjunto $\mathcal{A}$ de los números reales que se pueden representar exactamente es un conjunto finito. Sea entonces $x \in \mathcal{R}$, $x \notin \mathcal{A}$ (por ejemplo, por el mayor número de dígitos de mantisa),

$$x = a * b^t, \qquad \text{donde} \qquad b^{-1} \leq |a| < 1$$

$$a = 0.\alpha_1 \ \alpha_2 \ ...\alpha_m \ \alpha_{m+1} \ ..., \qquad 0 \leq \alpha_i \leq b-1, \qquad \alpha_1 \neq 0 \ .$$

Consideremos ahora

$$a' = \begin{cases} \pm 0.\alpha_1 \ \alpha_2 \ ...\alpha_m & 0 \leq \alpha_{m+1} \leq \frac{b}{2} - 1 \\ \pm 0.\alpha_1 \ \alpha_2 \ ...\alpha_m + b^{-m} & \frac{b}{2} \leq \alpha_{m+1} \leq b-1 \end{cases} \qquad (IV.1a)$$

esto es, se suprimen los dígitos que sigan al último representable si $0 \leq \alpha_{m+1} \leq \frac{b}{2} - 1$, y se aumenta en una unidad α_m si $\alpha_{m+1} \geq \frac{b}{2}$, y después se suprimen los dígitos que sigan al último representable. Entonces, está claro que

$$fl(x) = a' * b^t \in \mathcal{A} \ . \qquad (IV.1b)$$

Por ejemplo, supongamos que se desease utilizar el número $0.34826 * 10^4$ en una máquina en que $m = 4$ y $c = 2$. El truncamiento consiste en suprimir todos los dígitos que existen tras el último representable, sin mirar cuál es el dígito siguiente; por el contrario, el redondeo suprime los dígitos que sigan al último representable si el siguiente es menor o

igual que 4 o aumentan en una unidad el último representable, suprimiendo los restantes, si el que sigue es mayor o igual que 5. Así el número anterior se representaría como

$$\underline{|\ 3\ 4\ 8\ 2\ |\ 0\ 4\ |} \quad \text{con truncamiento ,}$$
$$\underline{|\ 3\ 4\ 8\ 3\ |\ 0\ 4\ |} \quad \text{con redondeo .}$$

La forma usual de utilizar los números es la última, porque se usa el número de la máquina que está más próximo al que se necesita.

Entonces, desde un número $x \notin \mathcal{A}$ se puede construir otro número $fl(x) \in \mathcal{A}$, naturalmente haciendo un error de redondeo. Para el error relativo de $fl(x)$ se tiene

$$\Big|\frac{fl(x)-x}{x}\Big| = \Big|\frac{a' * b^t - a * b^t}{a * b^t}\Big| = \Big|\frac{a'-a}{a}\Big| < \frac{\frac{b}{2}\ b^{-(m+1)}}{|a|} \leq \frac{b}{2} b^{-m} \ ,$$

siendo $|a| \geq b^{-1}$. En el caso de $b = 10$

$$\Big|\frac{fl(x)-x}{x}\Big| < \frac{5.0 * 10^{-(m+1)}}{|a|} \leq 5.0 * 10^{-m} \ ,$$

y si ponemos $\nu = 5.0 * 10^{-m}$, se puede poner que

$$fl(x) = x\ (1+\varepsilon) \ , \qquad \text{donde} \ \ |\varepsilon| \leq \nu \ .$$

El valor ν se llama **precisión de la máquina**.

En ocasiones, el número x no puede ser representado por la máquina al efectuar el redondeo, como indicamos en los cuatro casos siguientes:

$m = 4, c = 2, b = 10$
$fl(0.31794 * 10^{110}) = 0.3179 * 10^{110} \notin \mathcal{A}$
$fl(0.99997 * 10^{99}) = 0.1000 * 10^{100} \notin \mathcal{A}$
$fl(0.012345 * 10^{-99}) = 0.1235 * 10^{-100} \notin \mathcal{A}$
$fl(0.54321 * 10^{-110}) = 0.5432 * 10^{-110} \notin \mathcal{A}$.

En los dos primeros casos, el exponente es demasiado grande para caber en los lugares reservados para él, y se produce un **overflow** (rebasamiento del valor máximo), y en los dos últimos casos el exponente es demasiado pequeño para caber en los lugares reservados para él, y se produce un **underflow** (rebasamiento del valor mínimo). Los dos últimos casos tienen una posible solución, y es la de prevenirlos definiendo:

$fl(0.012345 * 10^{-99}) = 0.0123 * 10^{-99} \in \mathcal{A}$, (no normalizado)
$fl(0.54321 * 10^{-110}) = 0.0 \in \mathcal{A}$,

pero ahora el redondeo puede no verificar que

$$fl(x) = x\ (1+\varepsilon) \ , \qquad \text{con} \ \ |\varepsilon| \leq \nu \ .$$

Los ordenadores digitales tratan los fenómenos de overflow y de underflow de forma diferentes, y siempre como irregularidades del cálculo. En cierto casos, al producirse alguno de los rebasamientos, el ordenador continúa los cálculos con el mayor valor permitido (o cero

si se trata de un underflow) mostrando o no un mensaje de aviso de lo que ha ocurrido; en otros se muestra un mensaje de error y detiene el programa. Los rebasamientos pueden ser evitados si se hacen escalados adecuados de los datos, y si durante los cálculos se hacen chequeos, efectuando reescalados si fuese preciso.

El uso frecuente de la aritmética de redondeo en computadoras lleva a la siguiente definición: se dice que **el número p^* aproxima a p con m dígitos significativos** (o cifras) si m es el entero más grande no negativo para el cual

$$\left|\frac{p^* - p}{p}\right| \leq 5.0 * 10^{-m} \ .$$

La razón por la cual se usa el error relativo en la definición es que se desea obtener un concepto continuo. Por ejemplo, para que p^* aproxime a 1000 con cuatro cifras significativas, p^* debe satisfacer

$$\left|\frac{p^* - 1000}{1000}\right| \leq 5.0 * 10^{-4} \ , \qquad \text{y eso implica que} \qquad 999.5 \leq p^* \leq 1000.5 \ .$$

Pasamos ahora a ver brevemente algunas **representaciones internas** usadas por el ordenador para almacenar los números enteros y los reales.

1. Representación interna de números enteros en "magnitud-signo".
La escritura de un número en el sistema binario es la manera más sencilla de representarlo mediante un patrón de bits. La idea más simple para representar el signo menos es reservar un bit para ello, de forma que si dicho bit vale 0 el número es positivo, y si vale 1 es negativo. Normalmente se suele utilizar el bit situato más a la izquierda.

Ejemplo. Representar los número enteros 17 y -17 en magnitud-signo con 8 bits.

La representación binaria del número 17 es 10001. Entonces, en la representación magnitud-signo con 8 bits del número positivo 17 tendremos que el primer bit, que es el que denota el signo, es cero: 00010001. Para el número negativo -17 se obtiene 10010001.

Ejemplo. ¿Qué números decimales representan las series de 8 bits 11100010 y 00111011 codificados en magnitud-signo?

La primera serie 11100010 tiene un uno en el primer bit, indicando que el número representado es un número negativo. Los demás dígitos son la representación binaria del número

$$1100010 = 2 + 32 + 64 = 98 \ .$$

Entonces la primera serie representa al número entero -98.

La segunda serie 00111011 tiene un cero en el primer bit, indicando que el número representado es un número positivo. Los demás dígitos son la representación binaria del número

$$111011 = 1 + 2 + 8 + 16 + 32 = 59 \ .$$

Entonces la segunda serie representa al número entero 59.

2. Representación interna de números enteros en "notación en exceso".
La representación en magnitud-signo es una manera muy natural de codificar números en binario. Sin embargo, hay otras formas de codificación que permiten diseñar circuitos electrónicos más simples para interpretarlas. Una de éstas es la **notación en exceso**. La representación en exceso con p bits de un número decimal entero N consiste en codificar N como el equivalente binario del número $N + 2^{p-1}$, que se denomina **característica** de N con p bits.

Por ejemplo, la tabla siguente muestra la notación en exceso con 4 bits

$0000 = -8$	$1000 = 0$
$0001 = -7$	$1001 = 1$
$0010 = -6$	$1010 = 2$
$0011 = -5$	$1011 = 3$
$0100 = -4$	$1100 = 4$
$0101 = -3$	$1101 = 5$
$0110 = -2$	$1110 = 6$
$0111 = -1$	$1111 = 7$

El nombre "notación en exceso" se debe a la diferencia que hay entre el número codificado y el número binario directo que representa el patrón de bits. Nótese que a diferencia de la codificación en magnitud-signo, en la notación en exceso los números positivos tienen el primer bit igual a 1, mientras que los números negativos tienen el primer bit igual a 0.

Ejemplo. Representar en exceso con 8 bits los números enteros 23 y -49.

Para dar la representación en exceso del número 23 tenemos que hallar la representación binaria del número $23 + 2^{8-1} = 23 + 2^7 = 23 + 128 = 151$. Tal representación es 10010111 que coincide con la representación en exceso con 8 bits de 23. De manera parecida, para hallar la representación en exceso del número -49 tenemos que hallar la representación binaria del número $-49 + 2^{8-1} = -49 + 2^7 = -49 + 128 = 79$. Tal representación es 1001111, y ahora para tener la representación en exceso se necesitan añadir ceros a la izquierda; entonces, la representación en exceso con 8 bits de -49 es 01001111.

Ejemplo. ¿Qué número decimale representa el código en exceso 10010011?

Tenemos: $10010011 = 1+2+16+128 = 147$, entonces $N+128 = 147$ lo cual implica $N = 19$.

3. Representación interna de números enteros en "complemento a dos".
La representación en **complemento a dos** es una manera muy útil de codificar un número debido a que facilita enormemente las operaciones algebraicas. Para obtener el complemento a dos de un número binario hay que considerar en primer lugar el complemento a uno cuya definición es la siguiente: el **complemento a uno** de un número binario es el número que se obtiene al cambiar los ceros por unos y los unos por ceros. Conocido el complemento a uno, el complemento a dos se obtiene fácilmente: el **complemento a dos** de un número binario se obtiene sumando 1 al complemento a uno.

Ahora, la **representación en complemento a dos** de un número consiste en escribir los números positivos como su equivalente en el sistema binario, y los números negativos como el complemento a dos del equivalente en el sistema binario de su valor absoluto.

Para decodificar un número decimal representado en complemento a dos se procede del modo siguiente:

- si el primer bit de la izquierda es 0 el número es positivo. Entonces, el número representado es el equivalente del número binario que forma el resto de los bits.
- si el primer bit de la izquierda es 1 el número es negativo. Entonces el número representado es el opuesto del equivalente decimal del número binario que forma su complemento a dos.

Ejemplo. Representar con 8 bits en complemento a dos los números decimales 17 y -17.

La representación binaria del número 17 es 10001, entonces la representación con 8 bits en complemento a dos de 17 se obtiene añadiendo ceros a la izquierda: 00010001.

Para la representación con 8 bits en complemento a dos de -17 tenemos que realizar el complemento a dos de la representación binaria del su valor absoluto 17. Primero se pasa de 00010001 a su complemento a uno: 11101110. Ahora, se hace el complemento a dos, es decir se le suma 1: $11101110 + 1 = 11101111$. Esta es la representación con 8 bits en complemento a dos de -17.

Ejemplo. ¿Qué números decimales representan las series de 8 bits 00101011 y 10101011 codificadas en complemento a dos?

La primera serie 00101011 tiene un cero en el primer bit, indicando que el número representado es un número positivo. Los demás dígitos son la representación binaria del número 43 ($101011 = 1+2+8+32 = 43$). Entonces la primera serie representa al número entero 43.

La segunda serie 10101011 tiene un uno en el primer bit, indicando que el número representado es un número negativo y que entonces tenemos que realizar la operación de complemento a dos. Es decir, primero tenemos que pasar la representación 10101011 a complemento a uno: 01010100 y ahora a complemento a dos añadiendo uno: $01010100 + 1 = 01010101$. Finalmente el número buscado es el opuesto del equivalente decimal: $01010101 = 1 + 4 + 16 + 64 = 85$, es decir -85.

4. Representación interna de números reales en "punto flotante".

Los números fraccionarios y reales se introducen en el ordenador **en punto flotante**. Esta representación consiste en escribirlos en forma exponencial binaria normalizada y codificar tres campos: el signo, el exponente y la matisa. Cada uno de los campos se codifica de la manera siguiente:

1. El bit de signo se pone a 0 cuando el número es positivo y a 1 cuando el número es negativo.
2. El campo exponente se codifica usualmente mediante la notación en exceso.
3. El campo mantisa se codifica como el equivalente binario directo del número decimal dado.

Si se usan 32 bits para la representación pueden dividirse del modo siguiente: 1 bit para el signo, 7 bits para el exponente y 24 bits para la mantisa:

1 bit (signo) | 7 bits (exponente) | 24 bits (mantisa)

Ejemplo. Representar en punto flotante con 32 bits, 1 de signo, 7 de exponente y 24 de mantisa, los números decimales 104.3125 y -13506.96875.

El primer número 104.3125 es positivo, entonces el primer bit será un cero. La representación binaria del número es: 1101000.0101, cuya forma exponencial normalizada es 0.11010000101×2^7. El exponente (7) se codifica en exceso con 7 bits: $7+2^{7-1} = 7+2^6 = 7+64 = 71$ cuya represenatción binaria es 1000111. Finalmente, la mantisa tiene 11 bits (11010000101) y se completa con 13 ceros a la derecha. Entonces, la representación en punto flotante con 32 bits del número 104.3125 es

$$0 \mid 1000111 \mid 11010000101000000000000 \ .$$

De manera parecida, para el segundo número -13506.96875, notamos que es negativo, entonces el primer bit será un uno. La representación binaria del valor absoluto del número es: 11010011000010.11111. Su forma exponencial normalizada es $0.11010011000010111111 \times 2^{14}$. El exponente (14) se codifica en exceso con 7 bits: $14 + 2^{7-1} = 14 + 2^6 = 14 + 64 = 78$ cuya represenatción binaria es 1001110. Finalmente, la mantisa tiene 19 bits (11010011000010111111) y se completa con 5 ceros a la derecha. Entonces, la representación en punto flotante con 32 bits del número 104.3125 es

$$1 \mid 1001110 \mid 11010011000010111111100000 \ .$$

2. INTRODUCCION A LA ARITMETICA DE PUNTO FLOTANTE

Además de dar una representación inexacta de los números, la aritmética realizada en la computadora no es exacta. Sin embargo, usando números con representación en punto flotante con m dígitos de mantisa, las operaciones aritméticas elementales no se pueden siempre ejecutar de manera exacta, y los resultados de las operaciones no necesariamente son números de la máquina aunque los operandos lo sean. Por ello no se puede esperar reproducir de forma exacta las operaciones aritméticas en un ordenador digital. Deberemos contentarnos con sustituirlas por otras $(\oplus, \ominus, \otimes, \oslash)$ llamadas **operaciones de punto flotante**, que las aproximen tanto como sea posible. Esto se puede conseguir, por ejemplo, definiéndolas con la ayuda del redondeo:

suma: $x \oplus y = fl(fl(x) + fl(y))$
resta: $x \ominus y = fl(fl(x) - fl(y))$
multiplicación: $x \otimes y = fl(fl(x) * fl(y))$
división: $x \oslash y = fl(fl(x)/fl(y))$.

Esta aritmética idealizada corresponde a efectuar la aritmética exacta en la representación del punto flotante de x e y, y luego a la conversión del resultado exacto a su representación de punto flotante. Se pueden probar las relaciones

$$x \oplus y = (x + y)\ (1 + \varepsilon_1) \qquad (IV.2a)$$
$$x \ominus y = (x - y)\ (1 + \varepsilon_2) \qquad (IV.2b)$$
$$x \otimes y = (x * y)\ (1 + \varepsilon_3) \qquad (IV.2c)$$
$$x \oslash y = (x/y)\ (1 + \varepsilon_4)\ , \qquad (IV.2d)$$

donde $|\varepsilon_i| \leq \mu_i\ \nu$, y μ_i es un entero $\mu_i \geq 1$, que depende del tipo de máquina usada.

Podemos comprobar que las operaciones en punto flotante no verifican las reglas aritméticas normales:

a) $x \oplus y = x$ **no implica que** $y = 0$. Esta igualdad es cierta para todo y tal que $|y| < \frac{\nu}{b}|x|$ (b es la base del sistema de numeración usado). Según esto, la precisión de la máquina, ν, debería definirse como el menor número positivo g de la máquina, para el cual se cumple $1 \oplus g > 1$, $\nu = \min\{g \in \mathcal{A}\ /\ 1 \oplus g > 1 \ \text{ y } \ g > 0\}$.

Ejemplo.

$m = 3, c = 2, b = 10$

$x = 0.123 * 10^0$

$y = 0.000061 = 0.61 * 10^{-4} \leq 0.5 * 10^{-3} * 0.123 = 0.615 * 10^{-4}$

Entonces

$x \oplus y = x$

b) **no asociatividad**: $a \oplus (b \oplus c)$ puede ser diferente de $(a \oplus b) \oplus c$.

Ejemplo.

$m = 8, c = 2, b = 10$

$a = 0.23371258 * 10^{-4}$

$b = 0.33678429 * 10^2$

$c = -0.33677811 * 10^2$

Entonces

$a \oplus (b \oplus c) = 0.23371258 * 10^{-4} \oplus 0.61800000 * 10^{-3} = 0.64137126 * 10^{-3}$

$(a \oplus b) \oplus c = 0.33678452 * 10^2 \ominus 0.33677811 * 10^2 = 0.64100000 * 10^{-3}$

y el resultado exacto es

$a + b + c = 0.64137126 * 10^{-3}$.

Esto ha ocurrido porque cuando se restan dos números del mismo signo, se produce un efecto de cancelación si ambos coinciden en uno o más dígitos con respecto al mismo exponente (los dígitos comunes desaparecen). A pesar de que, cuando $x, y \in \mathcal{A}$, su diferencia también es un elemento de $\mathcal{A}$, y por lo tanto no hay errores de redondeo adicionales, veremos que la cancelación es un efecto peligroso cuando se trata de propagación de errores previos (cuando x e y provienen de cálculos que han necesitado redondeo).

c) **no distributividad**: $a \otimes (b \oplus c)$ puede ser diferente de $(a \otimes b) \oplus (a \otimes c)$.

Ejemplo.

$m = 2, c = 2, b = 10$

$a = 0.94 * 10^2$

$b = 0.33 * 10^2$

$c = -0.32 * 10^2$

Entonces

$a \otimes (b \oplus c) = 0.94 * 10^2 \otimes 0.1 * 10 = 0.94 * 10^2$

$(a \otimes b) \oplus (a \otimes c) = 0.31 * 10^4 \ominus 0.30 * 10^4 = 0.1 * 10^3$

y el resultado exacto es

$a * (b + c) = 0.94 * 10^2$.

Las operaciones aritméticas $+, -, *, /$, junto a las funciones para las que se hayan especificado sustituciones (por ejemplo raíz cuadrada, funciones trigonometricas, etc...) se llaman **funciones elementales**.

3. PROPAGACION DEL ERROR

Hemos comprobado que la no associatividad de la suma y la no distributividad del producto en un ordenador pueden provocar la obtención de resultados diferentes dependiendo de la técnica que se utilice para efectuar las operaciones. Entonces la propagación del error es un efecto muy importante a tener en cuenta, y se debe evitar en lo posible.

Generalmente, un problema matemático puede ser esquematizado en la manera siguiente: con un número finito de datos iniciales $x_1, x_2, ..., x_n \in \mathcal{R}$ queremos calcular un número finito $y_1, y_2, ..., y_m \in \mathcal{R}$ de resultados. Eso corresponde a asignar una función

$$\phi^{(i)}: D_i \to D_{i+1}, \qquad i = 0, ..., r, \qquad D_j \subseteq \mathcal{R}^{n_j} \tag{IV.3}$$

donde $\phi = \phi^{(r)} \circ \phi^{(r-1)} \circ ... \circ \phi^{(0)}$ y $D_0 = D \subset \mathcal{R}^n$, $D_{r+1} \subseteq \mathcal{R}^{n_r+1} \equiv \mathcal{R}^m$.

Entonces el problema es analizar como un error Δx y los errores de redondeo que se hacen en el cálculo se propagan y cambian el resultado final $y = \phi(x)$. Consideremos

$$\phi: D \subset \mathcal{R}^n \to \mathcal{R}^m, \qquad \phi(x) = \begin{pmatrix} \phi_1(x_1, \ldots, x_n) \\ \vdots \\ \phi_m(x_1, \ldots, x_n) \end{pmatrix}$$

con las funciones componentes continuas y con derivadas primeras continuas. Para hacer los calculos más sencillos, analicemos antes sólo la propagación del error sobre el dato inicial Δx, con un procedimiento del primer orden (si ε y η son dos números muy pequeños, entonces el producto $\varepsilon\ \eta$ se puede despreciar con respecto a ε y η).

Si x^* es una aproximación de x, los errores absolutos serán

$$\Delta x_i = x_i^* - x_i, \qquad \Delta x = x^* - x, \qquad \Delta y_i = \phi_i(x^*) - \phi_i(x)\ .$$

Si usamos el desarrollo en serie de Taylor hasta al primer orden

$$\Delta y_i = y_i^* - y_i = \phi_i(x^*) - \phi_i(x) \approx \sum_{j=1}^{n} (x_j^* - x_j)\ \frac{\partial \phi_i(x)}{\partial x_j} = \sum_{j=1}^{n} \Delta x_j\ \frac{\partial \phi_i(x)}{\partial x_j}\ , \tag{IV.4a}$$

ó en notación matricial

$$\Delta y = \begin{pmatrix} \Delta y_1 \\ \vdots \\ \Delta y_m \end{pmatrix} \approx \begin{pmatrix} \frac{\partial \phi_1(x)}{\partial x_1} & \cdots & \frac{\partial \phi_1(x)}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial \phi_m(x)}{\partial x_1} & \cdots & \frac{\partial \phi_m(x)}{\partial x_n} \end{pmatrix} \begin{pmatrix} \Delta x_1 \\ \vdots \\ \Delta x_n \end{pmatrix} = D\phi(x) \cdot \Delta x \ , \qquad (IV.4b)$$

con $D\phi(x)$ la matriz Jacobiana. Aquí el factor de proporcionalidad $\frac{\partial \phi_i(x)}{\partial x_j}$ mide la sensibilidad con la cual y "reacciona" a las variaciones absolutas Δx_j de x_j. La fórmula análoga para la propagación de los errores relativos es:

$$RE_{y_i} \approx \sum_{j=1}^{n} \frac{x_j}{\phi_i(x)} \frac{\partial \phi_i(x)}{\partial x_j} RE_{x_j} = \sum_{j=1}^{n} \frac{\Delta x_j}{\phi_i(x)} \frac{\partial \phi_i(x)}{\partial x_j} \ . \qquad (IV.5)$$

Aquí el factor $\frac{x_i}{\phi(x)} \frac{\partial \phi(x)}{\partial x_i}$ (a menudo se le conoce como **índice de condicionamento**) indica cómo el error relativo en x_i repercute en el error relativo de y. Si el índice de condicionamento es de valor absoluto suficientemente grande, errores relativos pequeños en los datos iniciales producen errores relativos muy grandes en los resultados. En ese caso se dice que el problema está **mal planteado**.

La propagación del error relativo en las operaciones elementales viene dada por:

1. $\phi(x,y) = x * y \quad \Rightarrow \quad RE_{x*y} \approx RE_x + RE_y$
2. $\phi(x,y) = x/y \quad \Rightarrow \quad RE_{x/y} \approx RE_x - RE_y$
3. $\phi(x,y) = x \pm y \quad \Rightarrow \quad RE_{x\pm y} \approx \frac{x}{x\pm y} RE_x \pm \frac{y}{x\pm y} RE_y$
4. $\phi(x) = \sqrt{x} \quad \Rightarrow \quad RE_{\sqrt{x}} \approx \frac{1}{2} RE_x \ .$

Consideremos $\phi(x) = \sqrt{x}$. Entonces $\phi'(x) = \frac{1}{2\sqrt{x}}$ y el error relativo es:

$$\frac{|\phi(x) - \phi(x^*)|}{|\phi(x)|} \approx |\phi'(x^*)| \left| \frac{x - x^*}{\phi(x)} \right| = \frac{1}{2} \left| \frac{x - x^*}{\sqrt{x x^*}} \right| \approx \frac{1}{2} \left| \frac{x - x^*}{x} \right| ,$$

por lo que el error relativo en $\phi(x^*)$ es aproximadamente la mitad del error relativo en x^*, y por lo tanto, la operación de calcular la raíz cuadrada es, desde el punto de vista del error relativo, una operación segura.

Es más, también en la multiplicación, división y extracción de raíz, los errores relativos en los datos iniciales no se notan de manera fuerte en el resultado. Eso pasa también en la suma si los operandos x e y tienen el mismo signo: los indices de condicionamento $x/(x+y)$, $y/(x+y)$ tienen un valor entre cero y uno, y su suma es uno, luego

$$|RE_{x+y}| \leq \max(RE_x, RE_y) \ .$$

Si en la operación de suma los operandos x e y tienen signo contrario, por lo menos uno de los factores $x/(x+y)$, $y/(x+y)$, es mayor que uno, y entonces, por lo menos uno de los errores relativos RE_x, RE_y es mayor. Esa amplificación del error es todavía mayor si $x \approx -y$, porque en ese caso en la expresión de $x+y$ los dos terminos se cancelan.

Ejemplo. Queremos estudiar el error obtenido para hallar la suma

$$\phi(\alpha, \beta, \gamma) = \alpha + \beta + \gamma$$

con $\phi : \mathcal{R}^3 \to \mathcal{R}$.

Para el calculo de ϕ se pueden usar los dos algoritmos:

Algoritmo 1	Algoritmo 2
$\eta = \alpha + \beta$	$\eta = \beta + \gamma$
$y = \phi(\alpha, \beta, \gamma) = \eta + \gamma$	$y = \phi(\alpha, \beta, \gamma) = \alpha + \eta$.

Las decomposiciones (IV.3) de ϕ en este caso son:

$$\phi^{(0)} : \mathcal{R}^3 \to \mathcal{R}^2 \ , \qquad \phi^{(1)} : \mathcal{R}^2 \to \mathcal{R} \ .$$

Entonces los algoritmos son:

Algoritmo 1	Algoritmo 2
$\phi^{(0)}(\alpha, \beta, \gamma) = \begin{pmatrix} \alpha + \beta \\ \gamma \end{pmatrix} \in \mathcal{R}^2$	$\phi^{(0)}(\alpha, \beta, \gamma) = \begin{pmatrix} \beta + \gamma \\ \alpha \end{pmatrix} \in \mathcal{R}^2$
$\phi^{(1)}(u, v) = u + v \in \mathcal{R}$	$\phi^{(1)}(u, v) = u + v \in \mathcal{R}$.

Usando el cálculo en punto flotante, (IV.2), se obtiene para el primer algoritmo:

$\eta = fl(\alpha + \beta) = (\alpha + \beta)\ (1 + \varepsilon_1)$

$\overline{y} = fl(\eta + \gamma) = (\eta + \gamma)\ (1 + \varepsilon_2) = [(\alpha + \beta)\ (1 + \varepsilon_1) + \gamma]\ (1 + \varepsilon_2)$

$= \alpha + \beta + \gamma + (\alpha + \beta)\ \varepsilon_1 + (\alpha + \beta + \gamma)\ \varepsilon_2 + (\alpha + \beta)\ \varepsilon_1\ \varepsilon_2 =$

$= (\alpha + \beta + \gamma)\big[1 + \frac{(\alpha+\beta)}{\alpha+\beta+\gamma}\ \varepsilon_1\ (1 + \varepsilon_2) + \varepsilon_2\big]$

Y para el error relativo

$$RE_y = \Big|\frac{y - \overline{y}}{y}\Big| = \Big|\frac{\alpha + \beta}{\alpha + \beta + \gamma}\ \varepsilon_1\ (1 + \varepsilon_2) + \varepsilon_2\Big|$$

Y despreciando los términos de orden superior (procedimento del primer orden):

$$RE_y \approx \Big|\frac{\alpha + \beta}{\alpha + \beta + \gamma}\ \varepsilon_1 + \varepsilon_2\Big| \ .$$

Si hubiésemos usado el segundo algoritmo, tendríamos:

$$RE_y \approx \Big|\frac{\beta + \gamma}{\alpha + \beta + \gamma}\ \varepsilon_1 + \varepsilon_2\Big| \ .$$

Los factores de amplificación $\frac{\alpha+\beta}{\alpha+\beta+\gamma}$ y $\frac{\beta+\gamma}{\alpha+\beta+\gamma}$, respectivamente, y 1, indican cómo los errores de redondeo ε_1 y ε_2 influyen sobre el error relativo RE_y del resultado. Dependiendo de cuál de las dos cantitades $(\alpha + \beta)$ o $(\beta + \gamma)$ es menor, se prefiere uno u otro algoritmo. En el caso del ejemplo visto para comprobar la no asociatividad:

$$\frac{\alpha + \beta}{\alpha + \beta + \gamma} \approx 0.5 * 10^5 \qquad \frac{\beta + \gamma}{\alpha + \beta + \gamma} \approx 0.97 \ .$$

Y eso explica la mayor precisión del segundo algoritmo.

Por lo que concierne a la propagación del error relativo, desde la relación $(IV.5)$, se tiene:

$$RE_y \approx \frac{\alpha}{\alpha+\beta+\gamma}\ RE_\alpha + \frac{\beta}{\alpha+\beta+\gamma}\ RE_\beta + \frac{\gamma}{\alpha+\beta+\gamma}\ RE_\gamma\ .$$

Y se puede decir que el problema está bien planteado si cada sumando α, β, γ es pequeño con respecto a $(\alpha+\beta+\gamma)$.

Ejemplo. Sabemos que las raices de $a\ x^2 + b\ x + c = 0$, cuando $a \neq 0$, son:

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \qquad x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}\ .$$

Consideremos la ecuación cuadrática

$$x^2 + 62.10\ x + 1 = 0$$

con raíces aproximadas:

$$x_1 = -0.01610723 \qquad \text{y} \qquad x_2 = -62.08390\ .$$

Para esta ecuación, b^2 es mucho mayor que $4ac$, así que en el cálculo de x_1 y x_2 el numerador involucra la sustracción de números casi iguales. Supongamos que efectuamos los calculos para x_1 usando aritmética de redondeo con cuatro dígitos.

$$\sqrt{b^2-4ac} = \sqrt{(62.10)^2 - 4.000} = \sqrt{3856. - 4.000} = \sqrt{3852.} = 62.06\ ,$$

así que

$$fl(x_1) = \frac{-b+\sqrt{b^2-4ac}}{2a} = \frac{-62.10+62.06}{2.000} = \frac{-0.040}{2.000} = -0.020$$

es una representación bastante pobre de $x_1 = -0.01611$ $(RE_{x_1} \approx 0.2415)$. Por otro lado, los cálculos para x_2 implican la adición de dos números casi iguales, $-b$ y $-\sqrt{b^2-4ac}$, y no presentan ningún problema.

$$fl(x_2) = \frac{-b-\sqrt{b^2-4ac}}{2a} = \frac{-62.10-62.06}{2.000} = \frac{-124.2}{2.000} = -62.10$$

es una aproximación precisa de $x_2 = -62.08$ $(RE_{x_2} \approx 0.0003222)$.

Para obtener una aproximación más exacta de x_1, aún con redondeo de cuatro dígitos, cambiamos la forma de la fórmula cuadrática **racionalizando** el numerador. Entonces:

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}\Big(\frac{-b-\sqrt{b^2-4ac}}{-b-\sqrt{b^2-4ac}}\Big) = \frac{-2c}{b+\sqrt{b^2-4ac}}$$

y desde luego

$$fl(x_1) = \frac{-2.000}{62.10+62.06} = \frac{-2.000}{124.2} = -0.0161\ .$$

La técnica de racionalización se puede aplicar para obtener una forma alternativa también para x_2

$$x_2 = \frac{-2c}{b - \sqrt{b^2 - 4ac}} \ .$$

Esta sería la expresión a usar si b fuera un número negativo. En nuestro problema, sin embargo, el uso de esta fórmula resulta no sólamente en la sustracción de números casi iguales, sino también en la división entre el resultado pequeño de esta sustracción. La inexactitud que esto produce es dramática:

$$fl(x_2) = \frac{-2c}{b - \sqrt{b^2 - 4ac}} = \frac{-2.000}{62.10 - 62.06} = \frac{-2.000}{0.040} = -50.00 \ .$$

Para comprender mejor ese "mal" resultado, hagamos

$$y = \phi(p, q) = p - \sqrt{p^2 + q}, \qquad p > 0$$

y determinemos el error relativo que se propaga en y. Dado que

$$\frac{\partial \phi}{\partial p} = 1 - \frac{p}{\sqrt{p^2 + q}} = \frac{-y}{\sqrt{p^2 + q}} \qquad \text{y} \qquad \frac{\partial \phi}{\partial q} = \frac{-1}{2\sqrt{p^2 + q}}$$

se sigue que

$$\begin{aligned} RE_y &\approx \frac{p}{y}\Big(\frac{-y}{\sqrt{p^2 + q}}\Big)\ RE_p + \frac{q}{y}\Big(\frac{-1}{2\sqrt{p^2 + q}}\Big)\ RE_q = \\ &= \frac{-p}{\sqrt{p^2 + q}}\ RE_p - \frac{q}{2y\sqrt{p^2 + q}}\ RE_q = \\ &= \frac{-p}{\sqrt{p^2 + q}}\ RE_p + \frac{p + \sqrt{p^2 + q}}{2\sqrt{p^2 + q}}\ RE_q \ . \end{aligned}$$

Dado que, si $q \geq 0$:

$$\Big|\frac{p}{\sqrt{p^2 + q}}\Big| \leq 1 \qquad \text{y} \qquad \Big|\frac{p + \sqrt{p^2 + q}}{2\sqrt{p^2 + q}}\Big| \leq 1 \ ,$$

entonces ϕ está bien planteada si $q > 0$, y mal planteada si $q \approx -p^2$. Además, si $|q|$ es muy pequeño con respecto a p^2, obtenemos el fenómeno de la cancelación, por el cual los errores de redondeo en el cálculo previo se amplifican notablemente.

Veamos ahora la forma en que se propagan los errores en el caso de la suma de varios términos, para poder deducir la forma correcta en que deberían realizarse las operaciones.

$$S = a_1 + a_2 + a_3 + a_4 + a_5$$

$$fl[((((a_1 + a_2) + a_3) + a_4) + a_5)] = fl(fl(fl(fl(a_1 + a_2) + a_3) + a_4) + a_5) =$$

$$= ((((a_1 + a_2)(1 + \varepsilon_2) + a_3)(1 + \varepsilon_3) + a_4)(1 + \varepsilon_4) + a_5)(1 + \varepsilon_5) =$$

$$(a_1 + a_2 + a_3 + a_4 + a_5)(1 + \delta)$$

y hay que acotar

$$|\delta| = \left| \frac{S - fl[...]}{S} \right| ,$$

sabiendo que $|\varepsilon_i| < \nu$.

$$\begin{aligned} fl[...] = {} & a_1\ (1+\varepsilon_2)\ (1+\varepsilon_3)\ (1+\varepsilon_4)\ (1+\varepsilon_5) + \\ & a_2\ (1+\varepsilon_2)\ (1+\varepsilon_3)\ (1+\varepsilon_4)\ (1+\varepsilon_5) + \\ & a_3\ (1+\varepsilon_3)\ (1+\varepsilon_4)\ (1+\varepsilon_5) + \\ & a_4\ (1+\varepsilon_4)\ (1+\varepsilon_5) + \\ & a_5\ (1+\varepsilon_5) \approx \\ \approx {} & a_1 + a_2 + a_3 + a_4 + a_5 + \\ & a_1\ (\varepsilon_2 + \varepsilon_3 + \varepsilon_4 + \varepsilon_5) + a_2\ (\varepsilon_2 + \varepsilon_3 + \varepsilon_4 + \varepsilon_5) + \\ & a_3\ (\varepsilon_3 + \varepsilon_4 + \varepsilon_5) + a_4\ (\varepsilon_4 + \varepsilon_5) + a_5\ \varepsilon_5\ , \end{aligned}$$

donde se han desestimado los sumandos $\varepsilon_i\ \varepsilon_j$ que son despreciables respecto a ε_i.

Si ahora consideramos la situación más desfavorable (todos los ε_i iguales en signo y con el mayor valor absoluto ν), tenemos la siguiente acotación:

$$|\delta| \leq \left|\frac{4\ \nu\ a_1}{S}\right| + \left|\frac{4\ \nu\ a_2}{S}\right| + \left|\frac{3\ \nu\ a_3}{S}\right| + \left|\frac{2\ \nu\ a_4}{S}\right| + \left|\frac{\nu\ a_5}{S}\right| =$$

$$= \frac{\nu}{S} \left[|4\ a_1| + |4\ a_2| + |3\ a_3| + |2\ a_4| + |a_5| \right] .$$

En general, al sumar progresivamente $a_1 + a_2 + ... + a_n$, el error relativo máximo que se comete es, aproximadamente:

$$\left| \frac{S - fl[...]}{S} \right| \leq \frac{\nu}{S} \left[(n-1)\ (|a_1| + |a_2|) + (n-2)\ |a_3| + ... + 2\ |a_{n-1}| + |a_n| \right] .$$

Está claro, pues, que esta acotación es menor si los números $a_1, ..., a_n$ se ordenan de menor a mayor antes de sumarlos. Obtenemos así la siguiente regla práctica:
si se desea hallar $\sum_{i=0}^{n} a_i$, con n grande, y se trata de una serie convergente, entonces $\lim_{n\to\infty} a_i = 0$, y debe efectuarse la suma en orden inverso.

Otro caso interesante es cuando se pretende calcular $\sum_{i=1}^{4} x_i\ y_i$. El resultado que se obtiene es:

$$\begin{aligned} fl\Big[\sum_{i=1}^{4} x_i\ y_i \Big] = {} & \Big(\Big\{ \big[x_1\ y_1\ (1+\delta_1) + x_2\ y_2\ (1+\delta_2) \big]\ (1+\delta_5) + \\ & x_3\ y_3\ (1+\delta_3) \Big\}\ (1+\delta_6) + x_4\ y_4\ (1+\delta_4) \Big)\ (1+\delta_7) = \\ = {} & x_1\ y_1\ (1+\delta_1)\ (1+\delta_5)\ (1+\delta_6)\ (1+\delta_7) + \\ & x_2\ y_2\ (1+\delta_2)\ (1+\delta_5)\ (1+\delta_6)\ (1+\delta_7) + \\ & x_3\ y_3\ (1+\delta_3)\ (1+\delta_6)\ (1+\delta_7) + \\ & x_4\ y_4\ (1+\delta_4)\ (1+\delta_7)\ . \end{aligned} \qquad (IV.6)$$

Observamos la falta de simetría en los resultados, debida a la no conmutatividad y no asociatividad de las operaciones de punto flotante.

Para simplificar la expresión anterior, vamos a obtener unas cotas manejables de los productos $(1+\delta_i)$.

Lema. Si $|\delta_i| \leq u, i = 1, ..., n$ y $n\ u \leq 0.01$, entonces

$$\prod_{i=1}^{n}(1+\delta_i) \leq 1 + 1.01\ n\ u\ .$$

Demostración. Antes, consideremos $0 \leq x \leq 0.01$, y entonces

$$1 + x \leq e^x \leq 1 + 1.01\ x\ .$$

La primera desigualdad es inmediata, así que sólo veremos la segunda:

$$\begin{aligned} e^x = \sum_{r=0}^{\infty} \tfrac{x^r}{r!} &= 1 + x\ \left(1 + \tfrac{x}{2} + \tfrac{x^2}{3!} + ... + \tfrac{x^n}{(n+1)!} + ...\right) \leq \\ &\leq 1 + x\ \left(1 + \tfrac{x}{2} + \tfrac{x^2}{4} + ... + \tfrac{x^n}{2^n} + ...\right) \leq \\ &\leq 1 + x\ \left(1 + x\ (\tfrac{1}{2} + \tfrac{x}{4} + ...)\right) \leq \\ &\leq 1 + x\ (1 + x) \leq 1 + 1.01\ x\ . \end{aligned}$$

Entonces, si $n \in \mathcal{N}$ y $0 \leq n\ u \leq 0.01$

$$(1+u)^n \leq (e^u)^n = e^{n\ u} \leq 1 + 1.01\ n\ u\ .$$

Ahora está claro que

$$\prod_{i=1}^{n}(1+\delta_i) \leq \prod_{i=1}^{n}(1+u) = (1+u)^n \leq 1 + 1.01\ n\ u\ .$$

c.q.d.

El resultado de este lema puede expresarse también como:

$$\prod_{i=1}^{n}(1+\delta_i) = 1 + 1.01\ n\ \theta\ u\ , \qquad \text{donde} \qquad |\theta| \leq 1.$$

Entonces, volviendo a $(IV.6)$, y suponiendo que $n\ u \leq 0.01$ (lo que se cumple en todas las situaciones reales):

$$\begin{aligned} fl\Big(\sum_{i=1}^{n} x_i\ y_i\Big) = {} & x_1\ y_1\ (1 + 4.04\ \theta_1\ u) + x_2\ y_2\ (1 + 4.04\ \theta_2\ u) + \\ & x_3\ y_3\ (1 + 3.03\ \theta_3\ u) + x_4\ y_4\ (1 + 2.02\ \theta_4\ u)\ , \qquad |\theta_i| \leq 1\ . \end{aligned}$$

En general se verifica:

Teorema: Si $n\ u \leq 0.01$ entonces

$$\begin{aligned} fl\Big(\sum_{i=1}^{n} x_i\ y_i\Big) = {} & x_1\ y_1\ (1 + 1.01\ n\ \theta_1\ u) + \\ & \sum_{i=2}^{n} x_i\ y_i\ (1 + 1.01\ (n+2-i)\ \theta_i\ u)\ , \qquad |\theta_i| \leq 1\ . \end{aligned}$$

Muchos computadores tienen la ventaja de que en la evaluación de productos escalares $\sum_{i=1}^{n} x_i\ y_i$, pueden ir acumulando los productos parciales $x_1\ y_1$, $x_1\ y_1 + x_2\ y_2$, ..., en doble precisión, de modo que la única vez que se redondea un número con precisión simple es cuando se da el resultado final. Con esta acumulación en doble precisión, el error en el cálculo del producto interno es aproximadamente el de una sola operación.

Queremos ahora usar la fórmula matricial (*IV*.4*b*) para describir la propagación del error de redondeo en un algoritmo. Como hemos ya visto, un algoritmo para calcular una función $\phi\colon D \subset \mathcal{R}^n \to \mathcal{R}^m$, para un dado $x = (x_1, \ldots, x_n)^t \in D$ corresponde a una descomposición de la aplicación ϕ en aplicaciones elementales, diferenciables continuamente, $\phi^{(i)}$, (ver (*IV*.1)), y nos lleva desde x hasta y con resultados intermedios

$$x = x^{(0)} \to \phi^{(0)}(x^{(0)}) = x^{(1)} \to \ldots \to \phi^{(r)}(x^{(r)}) = x^{(r+1)} = y\ .$$

Denotamos con $\psi^{(i)}$ la **aplicación resto**

$$\psi^{(i)} = \phi^{(r)} \circ \phi^{(r-1)} \circ \ldots \circ \phi^{(i)} : D_i \to \mathcal{R}^m\ , \qquad i = 0, 1, 2, \ldots, r\ .$$

Entonces, $\psi^{(0)} \equiv \phi$. $D\phi^{(i)}$ y $D\psi^{(i)}$ son las matrices Jacobianas de las aplicaciones $\phi^{(i)}$ y $\psi^{(i)}$, respectivamente. Dado que las matrices Jacobianas son multiplicativa con respecto de la composición de funciones, tenemos, para $i = 0, 1, 2 \ldots r$

$$D(f \circ g)(x) = Df(g(x)) \cdot Dg(x)\ ,$$

$$D\phi(x) = D\phi^{(r)}(x^{(r)}) \cdot D\phi^{(r-1)}(x^{(r-1)}) \ldots D\phi^{(0)}(x^{(0)})\ ,$$

$$D\psi^{(i)}(x^{(i)}) = D\phi^{(r)}(x^{(r)}) \cdot D\phi^{(r-1)}(x^{(r-1)}) \ldots D\phi^{(i)}(x^{(i)})\ .$$

Con aritmética de punto flotante, los errores iniciales y de redondeo perturberán los resultados intermedios $x^{(i)}$, de manera que se obtendrá el valor aproximado $x^{*(i+1)} = fl(\phi^{(i)}(x^{*(i)}))$. Para los errores absolutos obtenemos

$$\begin{aligned} \Delta x^{(i+1)} =& x^{*(i+1)} - x^{(i+1)} = \\ =& [fl(\phi^{(i)}(x^{*(i)})) - \phi^{(i)}(x^{*(i)})] + [\phi^{(i)}(x^{*(i)}) - \phi^{(i)}(x^{(i)})]\ . \end{aligned} \tag{*IV*.7}$$

Desde (*IV*.4*b*) sigue

$$\phi^{(i)}(x^{*(i)}) - \phi^{(i)}(x^{(i)}) \approx D\phi^{(i)}(x^{(i)})\Delta x^{(i)} \tag{*IV*.8}$$

Nótese que la aplicación $\phi^{(i)}\colon D_i \to D_{i+1} \subseteq \mathcal{R}^{n_i+1}$ es un vector de funciones componentes $\phi_j^{(i)}\colon D_i \to \mathcal{R}$, $j = 1, \ldots, n_i + 1$. Entonces, podemos escribir

$$fl(\phi^{(i)}(u)) = (I + E_{i+1}) \cdot \phi^{(i)}(u)\ ,$$

con I la matriz identidad y E_{i+1} la matriz diagonal cuyos elementos son los errores $\varepsilon_j,\ j=1,\ldots,n_i+1,\ |\varepsilon_j|\leq\nu$. Entonces, para el primer paréntesis de la expresión $(IV.7)$ sigue

$$\begin{aligned} fl(\phi^{(i)}(x^{*(i)}))-\phi^{(i)}(x^{*(i)}) =& E_{i+1}\cdot\phi^{(i)}(x^{*(i)}) \\ \approx& E_{i+1}\cdot\phi^{(i)}(x^{(i)})=E_{i+1}\cdot x^{(i+1)}=\alpha_{i+1}\ . \end{aligned} \tag{IV.9}$$

La cantidad α_{i+1} se puede interpretar como el error absoluto de redondeo creado cuando $\phi^{(i)}$ es evaluada en aritmética de punto flotante, y los elementos diagonales de E_{i+1} se pueden interpretar como los correspondientes errores relativos de redondeo.

Unendo $(IV.7)$, $(IV.8)$ y $(IV.9)$, se puede expresar $\Delta x^{(i+1)}$ como aproximación del primer orden de la maniera siguiente

$$\Delta x^{(i+1)}\approx\alpha_{i+1}+D\phi^{(i)}(x^{(i)})\cdot\Delta x^{(i)}=E_{i+1}\cdot x^{(i+1)}+D\phi^{(i)}(x^{(i)})\cdot\Delta x^{(i)}\ .$$

Sigue entonces que

$$\begin{aligned} \Delta x^{(1)} \approx& D\phi^{(0)}(x)\cdot\Delta x+\alpha_1\ , \\ \Delta x^{(2)} \approx& D\phi^{(1)}(x^{(1)})[D\phi^{(0)}(x)\cdot\Delta x+\alpha_1]+\alpha_2\ , \\ & \cdots\cdots\cdots \\ \Delta y=\Delta x^{(r+1)} \approx& D\phi^{(r)}(x^{(r)})\ldots D\phi^{(0)}(x)\cdot\Delta x+ \\ & +D\phi^{(r)}(x^{(r)})\ldots D\phi^{(1)}(x^{(1)})\cdot\alpha_1+\ldots+\alpha_{r+1}\ , \end{aligned}$$

que se puede escribir como

$$\begin{aligned} \Delta y \approx& D\phi(x)\cdot\Delta x+D\psi^{(1)}(x^{(1)})\cdot\alpha_1+\ldots+D\psi^{(r)}(x^{(r)})\cdot\alpha_r+\alpha_{r+1} \\ \approx& D\phi(x)\cdot\Delta x+D\psi^{(1)}(x^{(1)})\cdot E_1x^{(1)}+\ldots+D\psi^{(r)}(x^{(r)})\cdot E_rx^{(r)}+E_{r+1}y\ . \end{aligned} \tag{IV.10}$$

Es entonces la medida de la matriz Jacobiana $D\psi^{(i)}$ de la aplicación resto $\psi^{(i)}$ que es critica para el efecto de los errores de redondeo intermedios α_i ó E_i sobre el resultado final. Está claro que si para hallar el mismo resultados $\phi(x)$ se usan dos algoritmos diferentes, $D\phi(x)$ queda igual mientras que las matrices Jacobianas $D\psi^{(i)}$ que miden la propagación del error de redondeo serán diferentes. Un algoritmo se dirá **numéricamente más fiable** que otro si, por un dado conjunto de datos, el efecto total de redondeo, dado por $D\psi^{(1)}(x^{(1)})\cdot\alpha_1+\ldots+D\psi^{(r)}(x^{(r)})\cdot\alpha_r+\alpha_{r+1}$ es menor por el primer algoritmo que por el segundo.

Ejemplo. Queremos estudiar el error obtenido para hallar la operacion

$$\phi(a,b)=a^2-b^2$$

con $\phi:\mathcal{R}^2\to\mathcal{R}$. Dado que $a^2-b^2=(a+b)(a-b)$, se pueden usar para el calculo de ϕ los dos algoritmos

Algoritmo 1	Algoritmo 2
$\eta_1=a\times a$	$\eta_1=a+b$
$\eta_2=b\times b$	$\eta_2=a-b$
$y=\phi(a,b)=\eta_1-\eta_2$	$y=\phi(a,b)=\eta_1\times\eta_2\ .$

Las correspondientes decomposiciones ($IV.3$) de ϕ en este caso son

Algoritmo 1	Algoritmo 2
$\phi^{(0)}(a,b) = \begin{pmatrix} a^2 \\ b \end{pmatrix} \in \mathcal{R}^2$	$\phi^{(0)}(a,b) = \begin{pmatrix} a+b \\ a-b \end{pmatrix} \in \mathcal{R}^2$
$\phi^{(1)}(u,v) = \begin{pmatrix} u \\ v^2 \end{pmatrix} \in \mathcal{R}^2$	$\phi^{(1)}(u,v) = u \cdot v \in \mathcal{R}$
$\phi^{(2)}(\alpha,\beta) = \alpha - \beta \in \mathcal{R}$.	

Para el algoritmo 1 obtenemos

$$x = x^{(0)} = \begin{pmatrix} a \\ b \end{pmatrix} \ , \quad x^{(1)} = \begin{pmatrix} a^2 \\ b \end{pmatrix} \ , \quad x^{(2)} = \begin{pmatrix} a^2 \\ b^2 \end{pmatrix} \ , \quad x^{(3)} = y = a^2 - b^2 \ ,$$

$$\psi^{(1)}(u,v) = u - v^2 \ , \quad \psi^{(2)}(u,v) = u - v \ ,$$

$$D\phi(x) = (2a, -2b) \ , \quad D\psi^{(1)}(x^{(1)}) = (1, -2b) \ , \quad D\psi^{(2)}(x^{(2)}) = (1, -1) \ .$$

Además, dado que $fl(\phi^{(0)}(x^{(0)})) - \phi^{(0)}(x^{(0)}) = \begin{pmatrix} a \otimes a \\ b \end{pmatrix} - \begin{pmatrix} a^2 \\ b \end{pmatrix}$, tenemos, con $|\varepsilon_i| < \nu$

$$E_1 = \begin{pmatrix} \varepsilon_1 & 0 \\ 0 & 0 \end{pmatrix} \ , \ \alpha_1 = \begin{pmatrix} \varepsilon_1 \ a^2 \\ 0 \end{pmatrix} \ , \ E_2 = \begin{pmatrix} 0 & 0 \\ 0 & \varepsilon_2 \end{pmatrix} \ , \ \alpha_2 = \begin{pmatrix} 0 \\ \varepsilon_2 \ b^2 \end{pmatrix} \ , \ \alpha_3 = \varepsilon_3(a^2 - b^2) \ .$$

Desde ($IV.10$) con $\Delta x = (\Delta a, \Delta b)^t$ sigue

$$\Delta y \approx 2a\Delta a - 2b\Delta b + a^2\varepsilon_1 - b^2\varepsilon_2 + (a^2 - b^2)\varepsilon_3 \ . \qquad (IV.11)$$

De la misma manera para el algoritmo 2 sigue

$$x = x^{(0)} = \begin{pmatrix} a \\ b \end{pmatrix} \ , \quad x^{(1)} = \begin{pmatrix} a+b \\ a-b \end{pmatrix} \ , \quad x^{(2)} = y = a^2 - b^2 \ ,$$

$$\psi^{(1)}(u,v) = u \cdot v \ , \quad D\phi(x) = (2a, -2b) \ , \quad D\psi^{(1)}(x^{(1)}) = (a-b, a+b) \ ,$$

$$E_1 = \begin{pmatrix} \varepsilon_1 & 0 \\ 0 & \varepsilon_2 \end{pmatrix} \ , \ \alpha_1 = \begin{pmatrix} \varepsilon_1(a+b) \\ \varepsilon_2(a-b) \end{pmatrix} \ , \ \alpha_2 = \varepsilon_3(a^2 - b^2) \ .$$

Y entonces desde ($IV.10$) sigue

$$\Delta y \approx 2a\Delta a - 2b\Delta b + (a^2 - b^2)(\varepsilon_1 + \varepsilon_2 + \varepsilon_3) \ . \qquad (IV.12)$$

Desde las ecuaciones ($IV.11$) y ($IV.12$) se obtienen los siguientes efectos totales de redondeo:

$$|a^2\varepsilon_1 - b^2\varepsilon_2 + (a^2 - b^2)\varepsilon_3| \leq (a^2 + b^2 + |a^2 - b^2|)\nu \ ,$$

para el algoritmo 1, y

$$|(a^2 - b^2)(\varepsilon_1 + \varepsilon_2 + \varepsilon_3)| \leq 3|a^2 - b^2|\nu \ ,$$

para el algoritmo 2. Entonces, podemos decir que el algoritmo 2 es numéricamente más fiable que el algoritmo 1 cada vez que $\frac{1}{3} < |\frac{a}{b}|^2 < 3$; en los otros casos el algoritmo 1 es más fiable. Esto sigue desde la equivalencia de las dos relaciones $\frac{1}{3} \leq |\frac{a}{b}|^2 \leq 3$ y $3|a^2 - b^2| \leq a^2 + b^2 + |a^2 - b^2|$.

Por ejemplo para $a = 0.3237$ y $b = 0.3134$, con aritmética de cuatro dígitos significativos, se obtienen los siguientes resultados:

Algoritmo 1: $a \otimes a = 0.1048$, $b \otimes b = 0.9822 \times 10^{-1}$

$a \otimes a - b \otimes b = 0.6580 \times 10^{-2}$.

Algoritmo 2: $a \oplus b = 0.6371$, $a \ominus b = 0.1030 \times 10^{-1}$

$a^2 - b^2 = 0.6562 \times 10^{-2}$.

Resultado exacto: $a^2 - b^2 = 0.656213 \times 10^{-2}$.