Programación Numérica: punto flotante, raíces de ecuaciones y más

Miguel Angel Norzagaray Cosío

25 de octubre de 2007

Índice general

Índice de figuras

Índice de tablas

Listings

Pendientes:

- Preparar la simbología
- Terminar Preámbulo
- Capítulo de sumatorias
- En los lineamientos de depuración, aclarar el cambio de modo de redondeo para identificar programas sensibles
- Revisar ϵ y ε
- Comentar implicaciones de APF de los algoritmos
- Programar algoritmos faltantes
- Capítulo de programación de funciones
- Controlar las mayúsculas en las referencias
- Comprobar entrada por entrada del índice

Simbología

Los siguientes son símbolos que aparecen con frecuencia en el texto. Algunos se usan con un único significado, como $\mathbb{F}$, pero otros pueden usarse de dos maneras, como el caso de m, utilizado para la mantisa o para la multiplicidad de una raíz. La página indica el lugar donde se define, si es que ocurre.

No es del todo sencillo determinar aquello que el lector requiere, pero se espera que esta colección sea suficiente para facilitar la lectura fluida.

Símbolo	**Significado**	Pag.
APF	Aritmética de punto flotante	
β	La base del sistema numérico	26
$\mathbb{C}$	El conjunto de los números complejos	
ε_M	Épsilon de máquina	
e	El exponente (principalmente en la primera parte), pero también la base de los logaritmos naturales	
η	Menor número subnormal, $\eta = \beta^{1-e_{min}-p}$	
$\mathcal{F}$	La inversa de f, es decir, f^{-1}	
$\mathbb{F}$	El conjunto de los números de punto flotante	
f, f'	Una función y su derivada	
$f^{(n)}$	La derivada n-ésima de f	
$fl(x)$	El NPF más cercano al número real x	
$L_f(x)$	La convexidad logarítmica de f en x	
m	La mantisa, pero también la multiplicidad de una raíz (en el capítulo 6)	
μ	La unidad de redondeo, $\mu = \frac{1}{2}\varepsilon_M$	
NaN	Not a Number, cantidades que no se pueden calcular	
NPF	Número de punto flotante	
$\mathbb{R}$	El conjunto de los números reales	
p	Normalmente un polinomio o la precisión	
ufp	Unidad en la primera posición	
ulp	Unidad en la última posición	
$u(x)$	La corrección de Newton $\frac{f(x)}{f'(x)}$	
$\underline{x}$ y $\overline{x}$	Números de punto flotante que rodean a x	
$[x]$	El intervalo que contiene a x, es decir, $[\underline{x}, \overline{x}]$. También la diferencia dividdad de orden 0, $[x] = f(x)$.	
$[x_k, ..., x_{k+j}]$	La diferencia dividida de orden k, $\frac{[x_{k+1},...,x_{k+j}]-[x_k,...,x_{k+j-1}]}{x_{k+j}-x_k}$	

Preámbulo

En 1993 impartí por primera vez un curso de Métodos Numéricos para estudiantes de Ingeniería en Sistemas Computacionales, en la Escuela Superior de Cómputo (ESCOM) del Instituto Politécnico Nacional. Me di cuenta que el temario era en escencia el mismo que cursé con el nombre de Análisis Numérico en la Licenciatura en Matemáticas de la Universidad de Sonora, pese a la diferencia de perfiles de egreso. Igual que con carreras como Ingeniería Civil, Electrónica y otras.

Una de las características más distintivas de los profesionales de la computación es la programación de computadoras y eso debiera causar distinción en algunos programas de asigntura como los mencionados en el párrafo anterior. Los cursos de Análisis o Métodos Numéricos no explotan la ventaja de conocimientos de programación de los estudiantes. Simplemente se continúa impartiendo los problemas de matemáticas acostumbrados con los algoritmos de siempre.

Prefiero llamarlos *Programación Numérica*.

Creo firmemente que, en las carreras de computación, el curso de Programación Numérica debiera impartirse con un contenido que haga mejor uso de las habilidades que el estudiante está desarrollando y que además lo prepare para hacer utilizar con mayor conciencia su herramienta principal de trabajo: la computadora. Los contenidos actuales tendrían que ser modificados, determinando lo que sí es indispensable, tema para otra ocasión.

Si se lograra o no resolver el problema de fondo, que es la adecuada influencia de la asignatura en la formación profesional, faltaría asegurar que se tienen suficientes referencias en espaol con que se apoye el curso con el nuevo contenido[1] o que al menos sea útil para los interesados en programación numérica seria. Mucho material se encuentra disperso en libros (algunos difíciles de conseguir), revistas y páginas web. La traducción de Casares del libro de Overton [147] es la única excepción.

Debo insistir: se cambie o no la idea de esos curso, este texto intenta llenar un vacío en las referencias en español. No está pensado como un texto que deba seguirse capítulo tras capítulo, pero si permite introducirse a una mayor comprensión de la manera en que las computadoras realizan las operaciones aritméticas y las técnicas de programación más adecuadas. En ese sentido, puede

[1] La triste realidad es que los estudiantes de nivel superior dificilmente leen materiales en inglés.

ser útil como apoyo a los cursos tradicionales, optativos, tesis o incluso de curos posgrado.

Análisis numérico Métodos numéricos Matemáticas numéricas Programación matemática Programación numérica Programación científica

En los preliminares mateáticos no pude evitar incluir temas como métricas, funciones de Lipschitz y otros temas pues sí sirven al lector no matemático a acostumbrarse a esa nueva terminología. De esa manera, leer un libro o artículo especializados no será tan complicado como cuando sólo dispone de los conceptos de un libro típico de métodos numéricos. La presentación intenta ser más intuitiva que formal en casi todo el texto.

Parte I

Fundamentos de programación numérica

Introducción

Ya son varias décadas desde que la humanidad comenzó a utilizar computadoras, al menos del tipo que hoy se conocen, para hacer cálculos científicos y financieros. Aunque desde el principio se diseñaron formas de resolver los problemas de falta de precisión, aún es muy común que los profesionales de la computación y áreas afines como matemáticas, física y algunas ingenierías ignoren la magnitud de los problemas que pueden surgir.

Ya es un problema nada fácil la construcción de un modelo matemático que representa un fenómeno físico o financiero. Lo verdaderamente perturbador es que la deficiente precisión y exactitud no solo no permita resultados correctos sino que en ocasiones la magnitud de los errores es excesiva. En el peor de los casos, no es fácil predecir el tamaño del error.

Precisión no es lo mismo que exactitud, según se define más adelante.

Computadoras y problemas numéricos

Los cálculos numéricos con computadora pueden acarrear una gran cantidad de errores de redondeo. Es imposible evitarlos pero se pueden programar las operaciones con cuidado para minimizar los errores.

Este tema es comunmente considerado como demasiado especializado y hasta "esotérico" por algunos profesionales de la computación y suele pensarse que para los sistemas comunes y corrientes no se necesita. Sin embargo, sólo hay que considerar que por algún motivo:

- casi todo lenguaje de programación dispone de tipos para punto flotante,
- todo tipo de computadoras incluye aceleradores para cálculos de punto fotante,
- todos los compiladores incluyen instrucciones y opciones especiales para punto flotante,
- todo procesador incluye excepciones de punto flotante que respeta el sistema operativo y

- cuando se sume una nómina o cuadre un proceso contable, no siempre estarán correctos los centavos.

Es entonces un recurso cuyas generalidades deben ser conocidas por todo analista, diseñador y programador de sistemas y los detalles finos por quienes abordan la nada sencilla tarea de hacer rutinas matemáticas robustas.

Una computadora no analiza, simplemente ejecuta lo que se le dice y los resultados de un mismo algoritmo pueden variar aún en condiciones muy similares. La intuición no es una buena herramienta en este tipo de sistemas. Como la aritmética nativa de la computadora comete errores, es necesario poder medirlos para controlarlos, prevenirlos y evitar al máximo su propagación.

Los errores que se consideran en diversas áreas de aplicación son de diferente magnitud, ya sea el investigador que experimenta con algoritmos de alta precisión en los límites de la capacidad del hardware o el técnico que va al campo a hacer mediciones en condiciones poco o nada controladas. El analista numérico espera que los errores de punto flotante sean pequeños y aún así se preocupa, como bien presenta Stewart en su reporte [174], por los errores de las variables reales, deficiencia de lectura de campo o equipo con poca precisión, que ocasionan gran diferencia en la salida de un programa.

Los errores en las cantidades iniciales no se pueden controlar, pero hay programas donde no hay entrada de datos por parte del usuario sino un proceso matemático que origina una respuesta. Aún en este otro caso, lo errores de redondeo y conversión deben poder medirse.

Proceso típico del error en la computadora

El usuario promedio confía *a priori* en las computadoras. De manera simplista, regularmente ocurre lo siguiente.

1. El usuario captura un número exacto x, generalmente en base 10.
2. La computadora almacena x con su formato interno y redondea si no existe representación exacta. La base interna es generalmente 2.
3. Ya no hay x sino la aproximación $x+\delta$. Se espera que δ sea muy pequeña.
4. La computadora hace operaciones (inexactas) con $x+\delta$ y obtiene el resultado $f(x+\delta)$.
5. Para presentar el resultado al usuario, convierte $f(x+\delta)$ del formato interno a la salida que el usuario lee, redondeando de ser necesario.
6. El usuario lee el resultado $f(x+\delta)+\epsilon$, causado por el redondeo en la conversión.

El usuario normalmente cree que la computadora sí almacena sus datos originales.

Obsérvese que en elgunos casos, la representación puede ser exacta. Si x es un entero menor que 1000 y el programa lo multiplica por dos, no habrá error alguno. Pero por lo general, los problemas incluyen números fraccionarios para los que es común que no existe representación exacta en la computadora. Allí comienzan los problemas, en el paso 2, ya que se ocasiona lo que nos dice el paso 3: x se convierte en $x + \delta$.

Por ejemplo, en algunas computadoras, el .34 se almacena internamente como . 340000003576278686523437 5 y cuando se le presenta al usuario, se redondea de nuevo a .34, lo que significa que el usuario no ve realmente las cantidades con las que está trabajando[2]. En este caso no hay problema aparente, pero cuando la computadora comienza a hacer cálculos y más cálculos partiendo de las aproximaciones, el resultado, aún redondeado, puede ser incorrecto. ¿Qué tanto? Esa es la pregunta importante y requiere de análisis y diseño cuidadosos.

¿WYSIWYG?

De manera más sencilla, cuando un cálculo termina en una cantidad como 21.00000000001 es fácil suponer que el último dígito es incorrecto. Pero si el número es 3.394762376385945, no es obvio saber hasta dónde se puede confiar.

En algunos textos de programación, se indica a los princiantes que los variables flotantes de precisión simple son más rápidas pero menos precisas. La sorpresa es cuando el estudiante diligente hace una prueba y sí son menos precisas, pero no más rápidas en muchas ocasiones, pese a que muchos libros de texto y manuales de programación así lo afirman.

Lo mejor para el programador es interesarse en comprender las fuentes del error y su correcto uso en el desarrollo de programas. Decidirse por aprender sólo unas cuantas recetas para resolver los errores más comunes es práctico pero riesgoso. Con este texto ambas cosas son posibles y el lector decidirá el que le es más útil. En el caso de estudiantes, lo mejor es cuidar la parte formativa y comenzar con las bases.

En los preliminares matemáticos, aritmética de punto flotante y el estudio de la norma de IEEE está toda la información necesaria para entender ejemplos y programar rutinas con cuidado. Se recomienda especial atención a la aritmética de punto flotante, donde se estudian las propiedades del conjunto de números que la computadora utiliza para representar a los números reales.

Toda la exposición supone que se trabaja con escalares, aunque es posible desarrollar la teoría para vectores. Cuando el tema lo amerite, se comentarán las extensiones que complementen el caso escalar, ya sea para números complejos o para espacios vectoriales.

En la segunda parte se presentan aplicaciones de la programación numérica que ilustran el uso y efectos de la aritmética de punto flotante. Primeramente el caso de la suma de números para pasar a la aproximación de raíces de ecuaciones y terminar con la programación de funciones elementales, donde se toma el caso de la función exponencial como ejemplo.

[2] El cómputo numérico no es un sistema WYSIWYG (*w*hat you see is what you get).

Este es una panorama muy amplio de programación numérica que pretende informar y sensibilizar principalmente a estudiantes sobre los cuidados, dificultades y soluciones al desarrollar programas donde la aritmética de punto flotante sea importante.

De cualquier manera, ya es ventaja tener reunida toda esta información en un sólo documento. El lector podrá encontrar la misma información, pero distribuida en libros, artículos de revistas y no publicados y una gran cantidad de páginas web. La mayor parte se encuentra en inglés, lo que ocasiona problemas muchos estudiantes.

Capítulo 1

Preliminares matemáticos

Este texto es autocontenido en cuanto a la aritmética de punto flotante, pero sí se requiere de algunos conocimientos previos tanto de programación como de matemáticas.

Los siguientes son sólo síntesis de temas que debieran ser conocidos por cursos anteriores y algunos conceptos básicos nuevos, con unas cuantas anotaciones relacionadas con los detalles de su programación.

1.1. Error absoluto y relativo

Al resolver algunos problemas, comunmente no hay fórmulas aritméticas que den un resultado exacto, sino que van siendo aproximados poco a poco. Este proceso se detiene cuando ya no hay mejoras evidentes en la aproximación de la solución, aunque no estemos en un resultado exacto. Lo que interesa es un resultado suficientemente bueno, es decir, se tolera cierto error. Por ejemplo si se estuviéra calculando $\sqrt{2} = 1.414\,213\,562\,373\,095...$ se podrían tener aproximaciones sucesivas como las de la tabla 1.1.

Iteración	Aproximación
1	1.41
2	$1.414\,21$
3	$1.414\,213\,56$
4	$1.414\,213\,562\,373$
$\vdots$	$\vdots$

Tabla 1.1: La precisión va aumentando al mismo tiempo que la exactitud (estos conceptos se definen en esta sección), pero no siempre ocurre así.

De aquí que es necesario poder medir la disminución paulatina del error en cada etapa y así saber cuándo es tolerable detenerse.

Denominemos con x^* el número que queremos aproximar, es decir, la solución exacta de un problema. Si x es la aproximación que tenemos en un momento dado, entonces a la cantidad

$$E_{abs} = |x - x^*|$$

se le conoce como *error absoluto* y es simplemente la diferencia numérica entre la aproximación y el número exacto. Esta es la primera magnitud que se calcula al evaluar aproximaciones.

No se puede decir que el error sea el mismo.

Si se aproxima el 100 con el 99, el error absoluto es 1. Si se aproxima el 1000 con el 999, el error absoluto sigue siendo 1. Esto significa que el error absoluto no nos indica la magnitud del problema que tenemos al aproximar y por ello debe ser utilizado con cuidado.

Si bien resulta muy natural medir el error de esta manera, el resultado puede ser poco representativo. Por ejemplo, un error de 1mm puede ser despreciable si se está construyendo una carretera, pero es inaceptable si se está diseñando un circuito integrado. Muchas situaciones similares están al alcance de la mano.

Resulta más representativo o significativo si se calcula la proporción del error de la siguiente manera. Calculando

$$E_{rel} = \frac{|x - x^*|}{|x|}$$

se obtiene una medida de la relación del error considerando la magnitud de las cantidades. Este número se conoce como *error relativo*. Si se multiplica por 100 al error relativo tenemos el concepto de *porcentaje de error*.

Aquí ya se puede hablar con mayor claridad.

Si se aproxima el 100 con el 99, el error relativo es $1/99 = .\widehat{01}$ o $1\,\%$. Si se aproxima el 1000 con el 999, el error relativo es $1/999 = .\widehat{00\,1}$ o $.1\,\%$. Entonces, el error relativo va desde 0 para una aproximación exacta a 1 en el caso de una pésima aproximación donde x^* se aproxima con $2x^*$ (error del $100\,\%$).

En un caso más real, si se aproxima π con 3.1416, el error relativo es $E_{rel} = \frac{|\pi - 3.1416|}{\pi} \approx 2.338\,434 \times 10^{-6}$, lo que significa un error aproximado de $0.000002\,\%$. El error absoluto es $E_{abs} \approx 7.346\,410 \times 10^{-6}$.

1.1.1. Consideraciones de programación

Si estos conceptos van a utilizarse en un programa para computadora, debe notarse que en ambos casos, se hace uso de una cantidad no conocida que es x^*. Esto no debe confundir: si $x_1, x_2, ...$ son las aproximaciones sucesivas, el error en cada paso puede calcularse tomando las dos aproximaciones más recientes. La iteración $i + 1$, las aproximaciones más recientes corresponden a las iteraciones

i e $i+1$ del algoritmo, con valores x_i y x_{i+1}. Los errores absoluto y relativo se calculan como

$$E_{abs} = |x_i - x_{i+1}| \qquad \text{y} \quad E_{rel} = \frac{|x_i - x_{i+1}|}{|x_{i+1}|}$$

respectivamente. También es posible sustituir por x_i el denominador del error relativo, pero interesa E_{rel} para la variable más reciente, que es la que tiene más posibilidades de usarse como resultado. La evaluación de estas cantidades merece atención.

El mayor inconveniente de cualquiera de los dos errores es que se necesita calcular la diferencia entre dos números que pueden ser muy cercanos. Se detallará cuando se revise la pérdida de precisión y la llamada exageradamente cancelación catastrófica en la sección 2.3.2.

Por otra parte, siempre que una expresión aritmética aparece un cociente, es necesario verificar que el denominador no sea cero o muy pequeño en valor absoluto para evitarse problemas. En ese caso, utilizar el error absoluto es más seguro si la aproximación ocurre en una vecindad de 0.

En ocasiones, es útil el criterio combinado

No es raro que la mayor parte de los casos sólo un criterio se utilice.

$$\varepsilon_1 |x| + \varepsilon_2 \ < \ \delta$$

donde ε_1 es el error relativo, ε_2 es el absoluto y δ la tolerancia aceptable de fallo, lo que es admisible para aceptar un resultado como suficientemente aproximado. Si se desconoce el orden de magnitud de x, debe tenerse cuidado al determinar ε_1.

Una observación importante es que en muchos cálculos, donde las magnitudes son enormes o muy pequeñas, el error relativo siempre queda en la misma escala. De hecho, si se calcula el error relativo entre kx_i y kx_{i+1}, el error relativo se mantiene igual.

Además del error relativo, una manera más intuitiva de medir lo bien que una cantidad x aproxima a x^* es ver cuántas *cifras significativas* tienen en común. Por ejemplo: 3.141592 tiene 7 dígitos en común con π. Formalmente se dice que x coincide con x^* en p cifras significativas si $|x - x^*|$ es menor que media unidad (.5) de la p-ésima cifra de x. Por ejemplo, .55733 y .55734 coinciden en 4 cifras pues la diferencia es de 1 en la última cifra, menos de la mitad que la mayor diferencia posible[1], mientras que .55733 y .55739 coinciden en menos de 4 cifras, pues difieren en la última cifra por 6.

Algunos autores calculan las cifras que tienen en común dos números reales como $C_{a,b} = \log_{10} \left| \frac{a+b}{2(a-b)} \right|$ como una aproximación para el orden de magnitud de la diferencia y $C_{a,a} = \infty$.

Una asociación precisa entre el concepto de error relativo y la cantidad de cifras significativas correctas la presenta Higham en [88]. De cualquier forma, el

[1] La mayor diferencia posible es $10 - 1 = 9$: 1 menos que la base.

concepto de error relativo es por sí solo suficiente como medida de lo correcta que es una aproximación y sólo se usará el error absoluto cuando no genere problemas en cuanto a proporción.

1.1.2. Precisión, exactitud y otros conceptos

Los conceptos de error absoluto y relativo están íntimamente relacionados a otras formas de medir lo eficaz de una aproximación, como la precisión y la exactitud.

Se llama *precisión* al hecho de obtener casi los mismos resultados a partir de condiciones muy similares. Se le llama *exactitud* a la posibilidad de conseguir una buena aproximación de la solución que se desea. Ambos conceptos son fáciles de formalizar.

Sea y^* el valor que interesa aproximar y que es el resultado exacto de aplicar el método f al valor x^*. El método f recibe como entrada un valor inicial cercano a x^*. Se dispone de una colección de tales valores, representados por $\{x_i\}_{i=1}^n = \{x_1, x_2, ..., x_n\}$.

Cada evaluación $f(x_i)$ será exacta si el error $|y^* - f(x_i)|$ es pequeño[2], lo que significa que se puede medir para cada valor x_i. En caso de un conjunto de evaluaciones, se dirá que es exacto si su media aritmética $\overline{y_i}$ es exacta, es decir, $|y^* - \overline{y_i}|$ es un valor pequeño.

El conjunto de evaluaciones $\{y_i\}_{i=1}^n$ es preciso si los valores están poco dispersos. Otra forma de medir esto es con el estadístico desviación estándar $\sigma = \sqrt{\sum(\overline{y_i} - y_i)^2}$, que es una medida común de evaluación de la dispersión. Consultar algún libro de estadística para más detalles.

Un gráfico vale más que mil palabras.

La figura 1.1 se muestran los puntos x_i que son mapeados a los y_i con precisión por f pero no con exactitud. En este sentido, exactitud sí implica precisión, pero no al revés, como en la gráfica. La región sombreada es el mapeo que hace f sobre el conjunto de valores. Nótese que se ha utilizado el error absoluto. Es mas sencillo entender el concepto de esta manera.

Otra manera de definir los mismos conceptos es pensar que la precisión es una medida de la cercanía de la representación interna que la computadora hace de un número, mientras que la exactitud mide la cercanía del procedimiento de cálculo. Si los cálculos de conversión son exactos, la precisión se preserva.

Ejemplo. Un usuario, pretende calcular el área de un círculo. Aunque el radio sea exacto (un entero, por ejemplo), se necesita usar la constante π, que la computadora no puede almacenar con precisión infinita. Por ello, el programador escribe una aproximación truncada como $\pi^* = 3.141592653589793238$, para usar más de 15 cifras significativas. Aquí hay un **error de exactitud** en la representación de π, de la que es conciente el programador.

[2]Por *pequeño* se hace referencia la idea intuitiva de una cantidad cuya magnitud es mucho menor en proporción a los valores que interesan. Más adelante se definirá con formalidad.

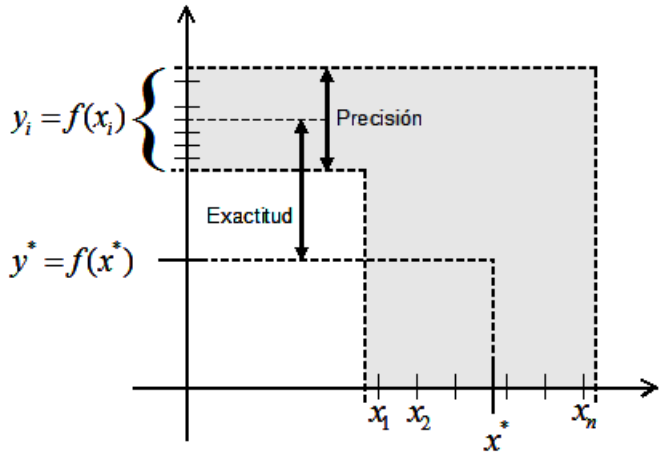


Figura 1.1: Precisión y exactitud.

Esta aproximación π^* es convertida por la computadora (por el compilador, de hecho) a un formato interno en binario, con lo cual se hace una conversión de base que no siempre puede llevarse a cabo sin pérdida de cifras significativas, por lo que el programa almacena una aproximación de π^*. Ahora tenemos un **error de precisión**, normalmente inadvertido por el programador promedio. Numéricamente ocurre lo siguiente:

Cada compilador utiliza su propio algoritmo de conversión, aunque es de esperarse que sean igual de exactos.

$$\begin{aligned} \pi &= 3.14159265358979323846264338327950288419 7... \\ \pi^* &= 3.141592653589793238 \\ \pi - \pi^* &= 4.626433832795028841 97... \times 10^{-19} \quad \text{Error de exactitud con } \pi \end{aligned}$$

En formato interno (en base dos) es

$$\begin{aligned} \pi^* &= 1.10010010000111111011010101000100010000101101000110 00 \times 2^1 \\ &= 3.141592653589793 \quad \text{Error de precisión con } \pi^* \end{aligned}$$

Hay ejemplos de este tipo para casi todo problema que se resuelve con computadora. En ocasiones, ambos términos son intercambiables, pero su diferencia es útil en muchas situaciones.

1.1.3. Estabilidad y condicionamiento

El concepto de *estabilidad* se refiere al posible crecimiento de los errores que se van produciendo durante las operaciones. Un algoritmo es *estable* si los errores no crecen considerablemente y es *inestable* cuando en etapas posteriores, los errores son muy significativos[3]. Más formalmente, si en la iteración i se tiene

[3] Qué tan significativos son, depende de lo crítico de las respuestas esperadas.

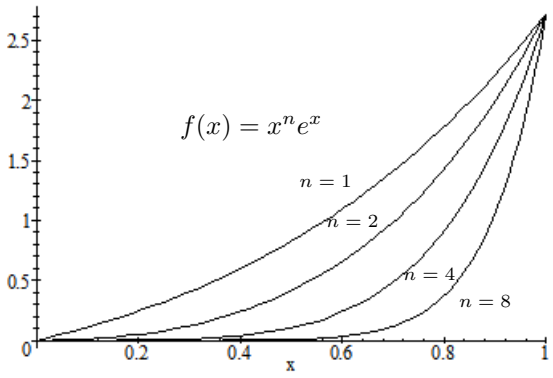


Figura 1.2: El área bajo la curva va disminuyendo en $[0, 1]$.

el resultado $x_i + \delta_i$ con δ_i el error en esa iteración y a partir de estos valores se produce el resultado $x_{i+1} + \delta_{i+1}$, el algoritmo será estable si la diferencia de los errores de la primera a la última iteración n es poca, es decir, $|\delta_n - \delta_1|$ es pequeño. Pero si $\delta_n \gg \delta_1$ entonces el algoritmo es inestable.

Los siguientes son dos ejemplos famosos de procesos inestables, que con frecuencia aparecen en libros de análisis numérico. Primero considérese la sucesión recursiva1.1, matemáticamente equivalente a $\frac{1}{3^n}$:

$\lim_{n\to\infty} \frac{1}{3^n} = 0$

$$\begin{cases} x_0 = 1,\ x_1 = \frac{1}{3} \\ x_{n+1} = \frac{13}{3}x_n - \frac{4}{3}x_{n-1},\ \text{para } n \geq 1. \end{cases} \tag{1.1}$$

Aunque sabemos que su límite es cero, esta sucesión diverge y se puede analizar para ver la razón. Lo que ocurre es que el error incluido en el término n-ésimo $x_n + \delta_n$ es multiplicado por $\frac{13}{3}$, y cada iteración los errores crecen poco más de cuatro veces. Se deja al lector el experimento de programación.

Otro ejemplo lo tenemos en la sucesión y_n:

$$\begin{aligned} y_n &= \int_0^1 x^n e^x \, dx \\ &= \frac{e}{n+1} - \int_0^1 e^x \frac{x^{n+1}}{n+1} \, dx \\ &= \frac{1}{n+1}(e - y_{n+1}) \end{aligned}$$

Conforme la n aumenta, la integral (el área bajo la curva) debiera tender a cero, como dejan ver los gráficos de y_n para $n = 1, 2, 4, 8$ de la figura 1.2.

Despejando se tiene que, $y_{n+1} = e - (n+1)y_n$. En este caso, el primer término es $y_0 = e - 1$. A partir de la figura 1.2 debiera ser obvio que $y_n \to 0$. Sin

embargo, esta sucesión también diverge al calcularla en computadora, debido a que el error h_n es multiplicado por un número cada vez más grande (n). Se deja al lector la programación.

Estos dos cálculos se dice que son *inestables* porque los errores producidos en una iteración son aumentados en iteraciones posteriores, degradando seriamente los resultados.

La forma de determinar si un cálculo es inestable debe ser por medio del error relativo. Es importante mencionar que no es una idea que el programador principiante utiliza empíricamente, requiriéndose algo de experiencia para estar completamente sensibilizado.

Entre los posiblemente diversos algoritmos para resolver un problema $\mathcal{P}$ pueden existir algunos más veloces o más exactos, así como también más o menos inestables. La estabilidad hace referencia a una característica del algoritmo empleado, no del problema que se está resolviendo.

Otro concepto relacionado es el de *condicionamiento*, que indica la proporción esperada de cambio en la salida de un algoritmo cuando se hacen pequeños cambios en la entrada. Si se evalúa $x + \delta$, con δ un valor pequeño, se obtiene $f(x)+\varepsilon$, y si ε también es pequeña (aproximadamente del mismo orden de magnitud que δ) se dice que f está *bien condicionado*, al menos en una vecindad de x. Se dice que está *mal condicionado* si $\delta \ll \varepsilon$.

Con la misma idea de pequeño de la página 10.

A diferencia del caso de la estabilidad, el condicionamiento es una característica del problema, no del algoritmo. Si un problema $\mathcal{P}$ está mal condicionado por naturaleza, por más efeciente que sea el algoritmo con que se intente resolverlo, debe tenerse cuidado con la exactitud de las respuestas. Por ejemplo, no se espera gran cosa de ningún algoritmo que se enfrente a la solución de un sistema de ecuaciones con la matriz de Hilbert porque se sabe que es una matriz sumamente mal condicionada.

La variación en la salida de un algoritmo puede escribirse como

$$f(x+h) - f(x) \approx hf'(x)$$

lo que es el error absoluto. De nuevo, el error relativo resulta más significativo, por lo que una buena forma de medir el *número de condición* de un problema es

$$\text{cond}(f) = \frac{f(x+h) - f(x)}{f(x)} \approx \frac{hf'(x)}{f(x)}.$$

En el caso de polinomios, se acostumbra calcular el número de condición como

$$\text{cond}(p, x) = \frac{\sum_{i=0}^{n} |a_i| \, |x|^i}{|\sum_{i=0}^{n} a_i x^i|}$$

donde $p(x)$ es el valor aproximado. Para matrices y producto de vectores cond

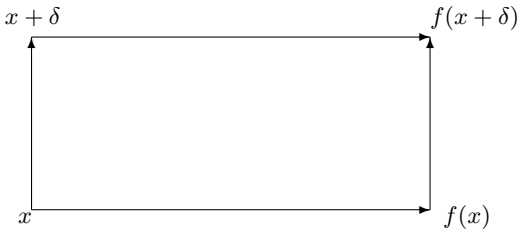


Figura 1.3: Backward y fordward error.

se aproxima con

$$\begin{array}{cc} \text{Producto de vectores} & \text{Matrices} \\ 2\dfrac{|x|\,|y|}{|x^T y|} & \|A\|\left\|A^{-1}\right\|. \end{array}$$

El número de condición es una forma de medir por adelantado la magnitud de los problemas numéricos que el programador tendrá.

Otras dos definiciones importantes para los analistas numéricos son las siguientes. A δ (o la versión relativa $|\delta/x|$) se le llama *backward error*, mientras que a $|\varepsilon| = |y - f(x+\delta)|$ se le llama *forward error*. El análisis completo de un algoritmo requiere que se den cotas para estos dos errores, lo que significa analizar el error tanto en los datos de entrada como en la salida. La figura 1.3 ilustra estas ideas.

En este caso, $f^*(x) = f(x+\delta)$ es procedimiento que se aplica en la realidad, que es el original f perturbado por razones externas (como el modo de redondeo o la precisión del hardware) o internas (como la estabilidad del algoritmo o el condicionamiento del problema).

¿Qué problema se resolvió realmente?

El backward error tiene la ventaja de interpretar los errores de redondeo como perturbaciones en los datos de entrada, es decir, responde la pregunta ¿qué valores próximos a x tienen como solución a $f(x)$? Esto es conocido también como la sensibilidad del problema. Con los polinomios de Taylor se presentará un ejemplo.

1.2. Notación asintótica

Siempre es un número natural.

Al comparar algoritmos, generalmente se mide la cantidad de recursos que utilizan. Los dos recursos básicos son el tiempo (de procesador) y el espacio (la memoria). A esto se le conoce como *complejidad algorítmica* y generalmente se expresa entérminos de n, que puede ser la cantidad de datos, el tamaño de la matriz, el grado del polinomio o la cantidad de bits, entre otras.

Casi siempre interesa el comportamiento cuando la complejidad aumenta arbitrariamente, por lo que interesa el comportamiento asintótico y en particular el peor caso. Para ello, se define la notación O, llamada notación-O u O grande, que es el conjunto de funciones $g(n)$ que acotan superiormente a una función dada $f(n)$. Por ejemplo, x^2 está acotada superiormente por x^3, a partir de cierta x_0 que en este caso es 1. Formalmente,

$$O(g(n)) = \{f(n) \text{ tales que existen constantes } c_1 \text{ y } n_0 \text{ que cumplen con } 0 \leq f(n) \leq cg(n) \text{ para toda } n \geq n_0\}. \tag{1.2}$$

Además de determinar una función que acota a otra, se utiliza para agrupar términos. Por ejemplo,

$$f(n) < 4n^2 + O(n)$$

significa que f tiene comportamiento cuadrático + términos de orden lineal cuya expresión es poco relevante. En ese sentido, deir que un algoritmo tiene complejidad temporal $O(n^2)$ significa que el tiempo de ejecución aumenta cuadráticamente conforme n aumenta, que es comunmente la cantidad de datos.

Una operación atómica como una suma o una asignación de un valor a una variable tienen complejidad $O(1)$, que es tiempo constante. Es importante mencionar que la notación asintótica puede usarse de manera general, suponiendo que cada paso de un algoritmo cuenta igual en términos del tiempo, pero también puede especificarse que algunas operaciones son realmente más cotosas. Por ejemplo, sumar o restar cuesta mucho menos que multiplicar.

EL análisis completo de un algoritmo puede consistir en determinar la complejidad mediante una expresión como $O(3n_1+5n_2)$, donde cada n pesa distinto e incluir el peor caso, el mejor caso y el caso medio con su desviación estándar. Más comunmente se simplifica considerando todas las operaciones al mismo costo a menos que se especifique lo contrario.

Por lo anterior, decir que un algoritmo es $O(4n^3 + n^2 + 2n\log n)$ es útil para conocer el detalle del tiempo, en términos de la cantidad de operaciones, pero es equivalente a decir $O(n^3)$ pues el término cúbico domina a los otros. Formalmente, $4n^3 + n^2 + 2n\log n \in O(n^3)$ pues es una función que pertenece a ese conjunto.

En ocasiones, si $|h| \ll 1$ es un término de error, expresiones como

$$f(x) < 4x^3 - \frac{hx^3}{3} + O(h^2)$$

significa que hay términos que incluyen errores de orden cuadrático y superior que no influyen en la cota.

Una excelente exposición de este tema se encuentra en el libro [40][4], donde además de O se presentan o, ω, Ω y Θ, para acotar por debajo o por ambos lados.

[4]Todo libro serio sobre algoritmos o estructuras de datos define estos conceptos.

1.3. Polinomios de Taylor

Brook Taylor (1685-1731) formalizó el concepto a partir de trabajos incompletos de Newton, Halley, Bernoulli, Leibniz y otros.

Los *polinomios de Taylor* se utilizan para aproximar funciones trascendentes para las que no existe alguna fórmula explícita calculable en una cantidad finita de pasos, como radicales o funciones trigonométricas. Su deducción puede encontrarse en muchos libros de cálculo.

Además de los polinomios de Taylor, existen otros como los de Chebyshev o las cocientes de polinomios de Padé, pero sólo se presentan los de Taylor, que son más ampliamente utilizados.

De manera general, si se conoce el valor de una función f continua en un punto x^* y f es infinitamente diferenciable en una vecindad de x^*, entonces se puede aproximar a f en valores vecinos a x^* con la *Serie de Taylor*:

$$f(x) = f(x^*) + hf'(x^*) + \frac{1}{2}h^2 f''(x^*) + \frac{1}{3!}h^3 f'''(x^*) + \cdots \tag{1.3}$$

donde $h = x - x^*$. Este resultado es conocido como el teorema de Taylor. Por ejemplo, para la función seno, se conoce con exactitud $\text{sen}(k\frac{\pi}{2})$ para k entera. En general cada función trascendente posee algún valor donde es conocida.

Evidentemente, los términos son cada vez más insignificantes por el factorial del denominador, por lo que esta serie se trunca cuando se obtiene una aproximación aceptable, quedando el polinomio de Taylor.

Colin Maclaurin (1698-1746).

Cuando x^* es el 0, obtenemos el polinomio de *McLaurin*. No es difícil suponer que este polinomio es utilizado comunmente bibliotecas de funciones comerciales, como las que acompañan a muchos compiladores, pero no es así exactamente. Los polinomios de grado alto son mal condicionados, lo que impone algunas restricciones en su uso, presentadas con mayor detalle en el capítulo 7.

Por ejemplo, estas son las series de Taylor (McLaurin en estos casos) para algunas funciones.

$$\begin{aligned}
\text{sen}(x) &= x - \frac{1}{6}x^3 + \frac{1}{120}x^5 - \frac{1}{5040}x^7 + \frac{1}{362\,880}x^9 + \cdots \\
\log(x+1) &= x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \frac{1}{4}x^4 + \frac{1}{5}x^5 - \frac{1}{6}x^6 + \cdots \\
e^{x^3} &= 1 + x^3 + \frac{1}{2}x^6 + \frac{1}{6}x^9 + \frac{1}{24}x^{12} + \frac{1}{120}x^{15} + \cdots
\end{aligned}$$

Si se trunca el polinomio (lo que en la práctica es necesario) es posible agregar un término que represente al error de truncamiento, de la siguiente manera:

$$f(x) = f(x^*) + hf'(x) + \frac{1}{2}h^2 f''(x) + \cdots + \frac{1}{n!}h^n f^{(n)}(\xi), \tag{1.4}$$

donde ξ es algún número en el intervalo $(x^* - h, x^* + h)$. El teorema de Taylor garantiza la existencia de tal número. Este último término es conocido como *residuo* y no es la única forma de expresarlo. Para mayor información debe consultarse algún libro de cálculo.

Determinar el valor máximo del residuo en cierto intervalo permite conocer la exactitud de la aproximación del polinomio de Taylor. También puede usarse la notación asintótica y escribir

$$f(x) = f(x^*) + hf'(x) + \frac{1}{2}h^2 f''(x) + \cdots + O(h^n)$$

que es más compacta. La primera expresión aparece en libros de matemáticas, mientras la segunda es más común en libros de computación y análisis numérico.

Si se considera la serie de Taylor de e^x como el polnomio

$$p(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots$$

y se trunca a los 5 primeros términos (hasta el término de grado 4), el fordward error analíticamente es

$$|e^x - p(x)| = \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!} + \frac{x^8}{8!} + \cdots$$

El backward error se calcula como el valor h tal que $e^{x+h} = p(x)$. Es inmediato que este caso, $h = |\ln p(x) - x|$. Numéricamente, si se opera con una precisión de 9 decimales, en $x = 2$ se obtiene

En este ejemplo hay precisión pero no exactitud.

$$\begin{aligned} e^2 &\approx 7.389\,056\,098 \\ p(2) &= 7 \end{aligned}$$

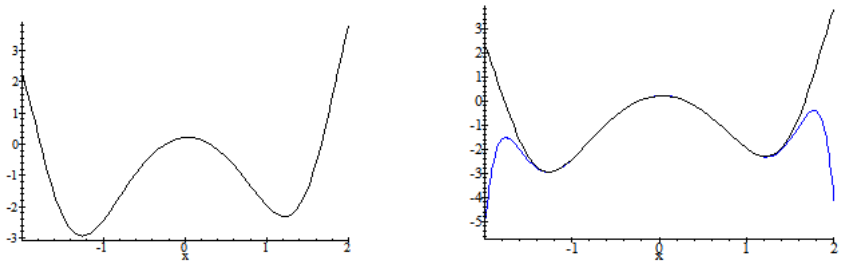
con un forward error de $0.389\,056\,098$ y un backward error de $\ln 7 - 2 \approx -.0$ $540\,898\,509$. Normalmente obtener las expresiones analíticas es lo que interesa y por lo general no resulta tan simple como en este caso.

Nótese que estos errores están calculados como absolutos, que es lo más común en estos casos.

1.3.1. Interpretación geométrica

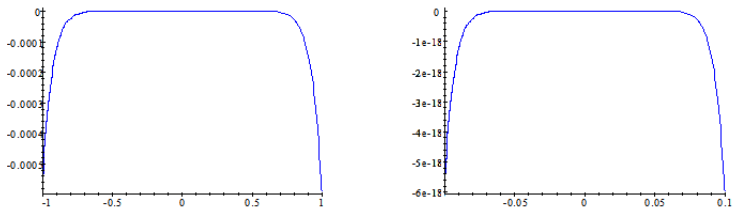
Considérese la función $\frac{e^x}{5} - 3\operatorname{sen} x^2$. Su gráfica en el intervalo $[-2, 2]$ se muestra en la figura 1.4. La figura 1.4b muestra superpuestas la función f y el polinomio aproximante

$$\begin{aligned} p(x) &= \frac{e^x}{5} - 3\operatorname{sen} x^2 \\ &= \frac{1}{5} + \frac{x}{5} - \frac{29x^2}{10} + \frac{x^3}{30} + \frac{x^4}{120} + \frac{x^5}{600} + \frac{1801x^6}{3600} + \\ &\quad \frac{x^7}{25\,200} + \frac{x^8}{201\,600} + \frac{x^9}{1814\,400} - \frac{453\,599x^{10}}{18\,144\,000} + \\ &\quad \frac{x^{11}}{199\,584\,000} + \frac{x^{12}}{2395\,008\,000} + \frac{x^{13}}{31\,135\,104\,000} + \cdots \end{aligned}$$



a. Función $f(x)$. **b.** Aproximación con $p(x)$.

Figura 1.4: Gráfica de $f(x) = \frac{e^x}{5} - 3\operatorname{sen} x^2$ y su aproximación con $p(x)$.



a. En $(-1, 1)$. **b.** En $(-0.1, 0.1)$.

Figura 1.5: Diferencia $f - p$ en intervalos dstintos. Poseen la misma forma pero la escala del eje de las ordenadas habla por sí solo.

y su valor en cero es $1/5$.

Si se trunca a 14 términos y se grafica junto con la función f se obtiene una aproximación que al alejarse del punto de aproximación toma dirección opuesta a la de la función original, como se muestra en la figura 1.4b.

Como puede verse, el polinomio se acerca en una vecindad del cero, aunque se aleja de la función cuando ampliamos el intervalo. Como de –.5 a .5 se confunde por completo con la función original, es más representativo graficar la diferencia entre f y la aproximación polinomial p.

En la gráfica de la diferencia (figura 1.5) se aprecia mejor que en la vecindad del 0 la aproximación es mejor, aún teniendo la misma forma, la escala del eje y habla por sí sola: en el intervalo [–.1, .1] (figura 1.5b) se tiene una precisión de más de 17 decimales.

Una función como $\sqrt{x}$ no posee polinomio de McLaurin pues no es continua en cero. Alejándose del cero, se puede desarrollar a

$$\sqrt{x+1} = 1 + \frac{1}{2}x - \frac{1}{8}x^2 + \frac{1}{16}x^3 - \frac{5}{128}x^4 + \frac{7}{256}x^5 - \frac{21}{1024}x^6 + \cdots$$

pues esta función sí es contínua y diferenciable en 1. De cualquier manera, el método de Newton es mejor en el caso de esta función y es el que se utiliza en las bibliotecas de funciones, luego de la etapa de reducción de rango y búsqueda en tabla[5].

1.3.2. Consideraciones de programación

Los polinomios de Taylor son entidades matemáticas muy prácticas por consistir sólo en operaciones elementales. Por ello no es de extrañar que sean de los primeros candidatos para aproximar funciones más complicadas, como las trascendentes. Utilizadas con cuidado, dan aproximaciones buenas. Lo que se necesita es determinar hasta qué término detener el cálculo de la serie y en qué intervalo se aplicarán. Aunque se darán muchos más detalles en el capítulo de evaluación de funciones, se presentan a continuación algunos lineamientos básicos para comenzar a comprender las ideas fundamentales de la programación numérica.

Entiéndase exponenciales, trigonométricas, ...

Cuando se programa, es posible detener la evaluación de los términos utilizando diversos criterios, no necesariamente excluyentes. En todos los casos, decir que algo es muy pequeño significa que es menor en valor absoluto que cierta cantidad fija que es la tolerancia aceptable.

- **Cantidad de términos evaluados**. No es recomendable como criterio principal porque en general se ignora el término a partir del cual se corta la aproximación. Este es más bien un criterio de seguridad para evitar que los cálculos se ciclen y es común que se combine con otros criterios.

- **El siguiente término del polinomio es despreciable**. Tiene el inconveniente de que puede ocurrir que antes de ser despreciable, su tamaño ya no modifique el resultado parcial anterior, con lo que se hacen operaciones ociosas. En las siguientes secciones se justificará este hecho.

  Por ejemplo, si la suma parcial hasta $n = 20$ es de $4\,851\,651\,95.4\,097$ y el siguiente término es $.000\ 000\,000\,797\,246$, es posible que en la aritmética de la computadora este pequeño valor ya no altere la suma parcial, aún cuando no es despreciable. También es conveniente aplicar la teoría y anticipar cuántos términos se requieren para lograr la precisión requerida en el intervalo de interés[6].

  Entre más se analice mejores resultados se obtendrán.

  Para no dejar una idea incorrecta, es necesario aclarar que no se acostumbra programar un polinomio de Taylor general y verificar en tiempo de ejecución cuántes términos se requieren. Se hace un trabajo de análisis previo para determinar cuántos se requierirán para la precisión deseada y se deja fijo en el código.

[5] Consultar el capítulo de Programación de Funciones en la segunda parte.
[6] El Teorema del Residuo es un buen inicio.

- **El error absoluto entre las dos aproximaciones más recientes es muy pequeño**. Es sencillo pero no indica la relación entre el error y la magnitud calculada. Un error de .001 puede ser muy grande, pero no si se aproxima un número del orden de 10^{20}. Es equivalente al criterio anterior.

- **El error relativo entre las dos aproximaciones más recientes es muy perqueño**. Un criterio práctico. En este caso debe cuidarse que la aproximación **NO** cause un poblema de división entre cero. Al dividir el error absoluto entre una de las aproximaciones, puede usarse la mayor en valor absoluto para evitar problemas.

No en todos los casos los polinomios de Taylor son la única opción. Existen otras familias de polinomios o funciones cuya combinación también puede aproximar a otras funciones, como las trigonométricas en las series de Fourier. En el capítulo 7 de aproximación de funciones se presentan más detalles.

1.4. Distancias, normas y otros conceptos

Cada vez que se resuelva un problema numérico con la computadora, la solución encontrada es una aproximación de la solución matemáticamente exacta. La forma de medir esta diferencia entre la realidad y el resultado es con el término de error, que es la diferencia numérica en el caso del conjunto de los números reales $\mathbb{R}$.

En muchas ocasiones, el conjunto $\mathbb{R}$ es insuficiente, como en el caso de las raíces del polinomio $x^2+1=0$, que son los números imaginarios $i, -i$, con parte real 0. Algunos métodos para el cálculo de raíces utiliza notación matricial, por lo que también debe ser posible medir el error entre vectores e incluso matrices.

Por lo anterior es necesario extender nuestro concepto de error para poder incluir cualquier entidad matemática: escalares, complejos, vectores, matrices, funciones y cualquier otro que sea necesario.

Esta rama de las matemáticas se profunduza en cualquier texto (y curso) de análisis funcional, aquí sólo se presentan los conceptos básicos que den el vocabulario y notación suficientes como para facilitar el desarrollo de los temas de los capítulos siguientes.

Se acostumbra llamar *espacios* en general a los conjuntos y *puntos* a sus elementos. Así, $\mathbb{R}$ es un espacio compuesto por una infinidad de puntos llamados números reales. El lector encontrar los encontrará referidos así en casi todo texto de matemáticas y por lo tanto se usarán aquí también.

No debe causar confusión con la idea geométrica de $\mathbb{R}^3$, el espacio más conocido por estudiantes.

1.4.1. Espacios normados

El término y concepto más básico es el de *norma*. Una norma es una función[7] denotada por $\|x\|$ que mide el "tamaño" o magnitud del punto x. Toda norma siempre está asociada a un espacio X en el que debe cumplir las siguientes propiedades. Sean x y y elementos arbitrarios de X y a cualquier escalar.

1. $\|x\| > 0$ si $x \neq 0$
2. $\|ax\| = |a| \, \|x\|$
3. $\|x + y\| \leq \|x\| + \|y\|$

La norma es una función que asocia un número real a cada punto del espacio X, es decir, $\|\cdot\| : X \to \mathbb{R}$. En el caso de que X sea el conjunto de los números complejos $\mathbb{C}$, los puntos son de la forma $x = a + ib$ donde a es la parte real y b la parte imaginaria. La norma en este caso particular se conoce como *módulo* y se define como

Como de costumbre, $i^2 = -1$.

$$\|x\| = \sqrt{a^2 + b^2} \tag{1.5}$$

y el lector puede comprobar sin problemas que se cumplen las tres condiciones de la norma. Además, esta definición es útil para el caso de vectores en $\mathbb{R}^2$ pero no es la única norma que se puede definir, sólo la convencional.

Equivale también a la hipotenusa.

Cuando existe tal norma asociada a X, se dice que X es un *espacio normado*. Una vez definida la manera de medir la magnitud de cada punto en el espacio, ahora falta definir la distancia entre cada par de puntos, lo que es sencillo a partir de su norma.

La *distancia* entre dos puntos x y y del espacio normado X es llamada *métrica* y se define como la norma de su diferencia

$$d(x, y) = \|x - y\| . \tag{1.6}$$

Las propiedades relevantes de d son las siguientes. Sea z otro punto de X además de x y y.

1. $0 \leq d(x, y) < \infty$
2. $d(x, y) = 0$ si y solo si $x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, y) \leq d(x, z) + d(z, y)$

[7] Se acostumbara usar de manera indistinta las palabras función, aplicación, mapeo y transformación.

La cuarta propiedad es la famosa *desigualdad del triángulo*. A X también se le llama también *espacio métrico*. Cuando se distingue un espacio métrico de otro por la distancia asociada a cada uno, se acostumbra escribir (X, d_X), que significa que el espacio métrico X tiene asociada la métrica o distancia d_X.

Así como se define el intervalo cerrado unitario $[0, 1]$ en $\mathbb{R}$, en un espacio métrico general se define una *bola unitaria cerrada* como el conjunto de todos los puntos con norma menor o igual a 1, es decir, $\overline{B}_1(0) = x \in X, \|x\| \leq 1$.

Algunos autores escriben $\overline{B_1(0)}$ en vez de $\overline{B}_1(0)$

Igualmente se define la bola unitaria abierta $B_1(0) = x \in X, \|x\| < 1$, que es como la $\overline{B}_1(0)$ pero sin la corteza o cáscara. La pura corteza serían todos los puntos cuya norma es igual a 1. Esto equivale en $\mathbb{R}$ al intervalo $(0, 1)$, donde la cáscara serían los números 0 y 1.

Se puede definir una bola de cualquier radio, abierta o cerrada, por ejemplo la bola abierta de radio r

$$B_r(0) = \{x \in X,\ \|x\| < r\} \tag{1.7}$$

y si se desea centrar la bola en el punto z en vez del cero se acostumbra escribir

$$B(z, r) = \{x \in X,\ \|x - z\| < r\} \tag{1.8}$$

cuya concepción es importante cuando se trata de la convergencia de diversas sucesiones. Quitando a z de esa bola, lo que queda es la *vecindad abierta de radio r de z*.

1.4.2. Espacios de Banach

Una extensión muy importante es el concepto de espacio de Banach, para lo que se requiere definir una *sucesión de Cauchy*. Una sucesión de puntos $\{x_n\}$ es de Cauchy si satisface

$$\lim_{\min(m,n)\to\infty} d(x_m, x_n) = 0 \tag{1.9}$$

lo que significa que la sucesión converge. Un espacio es *completo* si toda sucesión de Cauchy converge.

Ahora ya se puede definir que un *espacio de Banach* es un espacio normado completo. Con definiciones adecuadas de norma, los reales, complejos, $\mathbb{R}^n$ y las matrices son espacios de Banach. Por ello es que el estudio de las propiedades generales de estos espacioes es tan importante en análisis numérico.

Lo que se conoce como *endomorfismo*.

Sea $f : X \to X$ una función de un espacio métrico en sí mismo, es decir, toma un punto de X y su resultado es también un punto de X. Se dice que f es una *función contractiva* si existe una constante k que cumple con $0 \leq k < 1$ y

$$d(f(x), f(y)) \leq k\, d(x, y) \tag{1.10}$$

para toda x y y puntos de X. Esta definición es de gran importancia para el capítulo 6, donde en particular interesa que una función sea de Lipschitz.

Una función $f : X \to Y$ de un espacio métrico (X, d_X) a otro (Y, d_Y), se dice que es de *Lipschitz* cuando

$$d_Y(f(a), f(b)) \leq L\, d_X(a, b) \tag{1.11}$$

con a y b elementos de X. A L se le llama *constante de Lipschitz*[8]. El caso particular cuando $X = Y$ es de gran importancia práctica.

Es común que para aplicar con éxito algunos algoritmos sea necesario que se cumplan ciertas condiciones. Un caso común es cuando para una función $f : X \to Y$ se requiere que sea 1-a-1 y que tanto f como su inversa f^{-1} sean continuas. Toda función que cumple con esos requisitos es llamada *homeomorfismo*.

1.4.3. Convexidad

Hay dos maneras básicas de aplicar este concepto: a conjuntos y a funciones. La mejor forma de comprenderlos es con la idea geométrica. Un subconjunto $C \subset X$ es *convexo* si dados dos puntos cualesquiera x y y, la recta que los une está completamente contenida en C, es decir,

$$\lambda x + (1 - \lambda)y \in C, \qquad \text{con } 0 \leq \lambda \leq 1. \tag{1.12}$$

Una función f continua en el intervalo $[a, b]$ con primera y segunda derivadas continuas es *convexa* si para todo $x \in [a, b]$ se cumple que $f''(x) > 0$. La parábola definida por x^2 es una función convexa pues su segunda derivada es 2 y además define un conjunto convexo, que es la región que queda entre las dos ramas de la parábola que se muestra en la figura 1.6. La región externa es un conjunto cóncavo.

Puede verse al conjunto completo de puntos generados por f en cualquier intervalo real como el conjunto de las coordenadas cartesianas $(x, f(x))$ mapeadas dentro de la parábola, como $f([a, b]) \subset \mathbb{R}^2$. Toda la región sombreada en la figura 1.6 es $f(\mathbb{R})$.

Queda responder la pregunta ¿Qué tan convexa es la función f? Dentro de las medidas aceptadas se encuentra

$$L_f(x) = \frac{f(x)f''(x)}{f'(x)^2} \tag{1.13}$$

para la convexidad de f en x, conocida como *convexidad logarítmica*. Para esto se requiere que $f'(x) \neq 0$ y que f tenga hasta la segunda derivada continua.

Otros conceptos de utilidad serán definidos al momento de necesitarse. Los anteriores son una base firme tanto conceptual como de vocabulario para poder

[8] Normalmente se toma la menor L que cumpla con la condición (1.11), pero no se necesitará tanto en este texto.

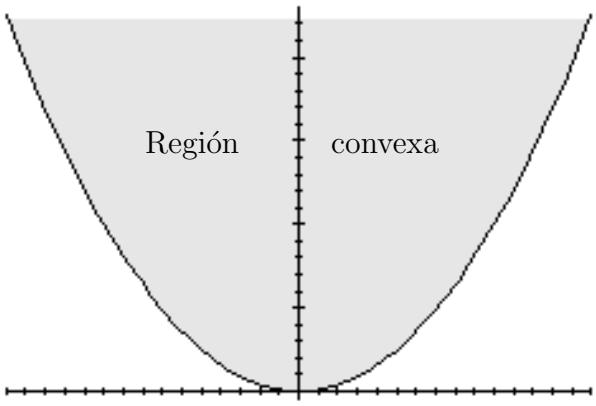


Figura 1.6: La función convexa define un conjunto convexo. Cualquier segmento de recta con ambos extremos dentro de la región sombreada queda completamente contenido allí mismo.

explicar algunos temas de manera clara y concisa. Debe entenderse que la presentación anterior es sencilla pero suficientemente completa para los fines de este texto. Los conceptos pueden formalizarse más aún, pensando en espacios topológicos, pero no es realmente necesario.

Capítulo 2

Aritmética de Punto Flotante

Las primeras investigaciones conocidas de Aritmética de Punto Flotante (APF) provienen de von Neumann y principalmente Householder, como el trabajo [183] de errores en la inversión de matrices, continuado en [184] y [89]. Otros ejemplos son Brown [20] y John Carr publicó [95] donde compara dos formatos distintos para los números de punto flotante, referidos como NPF de aquí en adelante.

John von Neumann, (1903-1957).

Alston Scott Householder, (1904-1993),.

Esta es una rama muy joven de las matemáticas, con sus origenes consolidados sistemáticamente en el trabajo de Wilkinson [187], aunque también Forsythe [68], Golub y Van Loan [128] son referencia excelente.

James Hardy Wilkinson, (1919-1986).

Hay dos formas de representa números que requieren punto decimal: con punto fijo o con punto flotante. Los números de *punto fijo* son sencillos de entender a partir de la tabla 2.1.

Los números que se utilizan normalmente para la mayoría de las personas son de este tipo y también se le llama sistema posicional. El punto se mantiene con su significado convencional de separar la parte fraccionaria de la parte entera[1].

[1]Las primeras computadoras realizaban todas las operaciones con números enteros y el

550043.4238901
8123.456
3.14159265...
0.0000122
0.00000000000232

Tabla 2.1: Números en notación de punto fijo. Cada posición indica su magnitud u orden (`..., decenas, unidades, punto, décimas,...`)

$$\begin{aligned} 123.456 &= .123456 \times 1000 \\ &= 123456 \times \frac{1}{1000} \\ &= 1.23456 \times 100. \end{aligned}$$

Tabla 2.2: Números en punto flotante.

En la práctica, los números tenían que tener una cantidad fija de cifras tanto para la parte entera como la fraccionaria, lo que limitaba considerablemente el rango de los números que se podían representar.

Los números de *punto flotante* son todos aquellos en los que se selecciona dónde poner el punto, agregando un factor a la representación para que el significado numérico sea el mismo. Por ejemplo en la tabla 2.2

De esta manera, el punto "flota".

Por convención la posición del punto se selecciona para mantener la parte entera dentro de algún rango predefinido, como se muestra a continuación.

2.1. Notación científica

Al escribir números de magnitudes muy diversas, es costumbre utilizar una notación que sea homogénea y fácil de entender. Se muestran a continuación varios números y su *notación científica* correspondiente en base 10.

$$\begin{gathered} 6000000000000000000 = 6.0 \times 10^{18} \\ 23\,000\,000\,000\,000\,000\,000 = 2.3 \times 10^{19} \\ .00000000000000000012 = 1.2 \times 10^{-20} \\ 1.0 = 1 \times 10^{0} \end{gathered}$$

Cuando se utiliza un sistema numérico con una base distinta a 10, se acostumbra indicarlo con un subíndice[2], de tal manera que usando las bases 10, 16, 8 y 2 respectivamente, se escribe $15 = f_{16} = 17_8 = 1111_2$. Como el sistema en base 2 se usará muy frecuentemente, se omitirá el subíndice excepto cuando pueda ser confuso. La base puede ser cualquier número $\beta > 1$.

También es común que se utilice notación científica para redondear algunas cantidades o simplemente por ahorro de espacio, por ejemplo en el siguiente caso[3]: $200000000000000000000.00000000000000000001 = 2.0 \times 10^{21}$.

usuario escalaba resultados posteriormente. Al poco tiempo se comenzó a utilizar un sistema de punto fijo no como 2.1, sino uno donde todos los números estaban en el intervalo $(-1, 1)$. Era tan poco flexible que a los pocos años se comenzó a utilizar el sistema de punto flotante, que ya se utilizaba en máquinas calculadoras.

[2]Tambien se usa `fh` = `xf` = 1111b = 1111B para al 15 en hexadecimal y binario, pero en este texto se utilizará la convención del subíndice.

[3]¿El redondeo es con fines estéticos o prácticos?

Como se ve, un número en notación científica se forma con un número real en el intervalo $[1, 10)$ y una potencia de alguna base, en este caso se usó el 10. En general es de la forma

$$d_0.d_1d_2...d_p \times 10^e = d_0 \times 10^e + d_1 \times 10^{e-1} + \cdots + d_p \times 10^{e-p} \tag{2.1}$$

con notación expandida, donde d_0 es distinto de cero y cada otro d_i cumple con $0 \leq d_i < 10$. A la cadena de dígitos $d_0.d_1d_2...d_p$ se le llama mantisa. Cualquier cantidad que se pueda descomponer como la suma de potencias de 10 no tendrá problemas de representación, pero cantidades como $1/3$ son un caso distinto, requiriéndose una infinidad de términos.

Con las restricciones anteriores, la notación es única para cada número. Por ejemplo:

$$\begin{aligned} 3.1416 \times 10^0 &= 3 \times 10^0 + 1 \times 10^{-1} + 4 \times 10^{-2} + 1 \times 10^{-3} + 6 \times 10^{-4} \\ 7.56 \times 10^{12} &= 7 \times 10^{12} + 5 \times 10^{11} + 6 \times 10^{10}. \end{aligned}$$

En computación, la notación corta $d_0.d_1d_2...d_p \times 10^e$ se prefiere a la expandida, más utilizada en textos de aritmética. La cantidad p de decimales después del punto es comunmente referida como precisión y se fija en vez de considerar números arbitrariamente precisos como $d_0.d_1d_2...d_pd_{p+1}... \times 10^e$ pues es la mejor manera de modelar la representación limitada de las computadoras.

El exponente e es un número entero (positivo o negativo) finito, aunque de nuevo en las computadoras tiene restricciones.

Obsérvese que dos números que difieran sólo en el último dígito, tienen una diferencia que es múltiplo de 10^{e-p}. A esta cantidad genéricamente se la llama *ulp* por sus siglas en inglés *units in last position*. Por ejemplo, 3.141592653589 y 3.141592653590 difieren en .000000000001 o 1 ulp:

Acuñado en 1960 por Kahan, para referirse a la exactitud de la librería matemática de las IBM 7090 y 7094.

$$\begin{array}{rl} & 3.141592653590 \\ - & \underline{3.141592653589} \\ & 0.000000000001 \end{array}$$

Con este nuevo concepto, puede redefinirse el de cifras significativas. Ahora se puede decir que deben coincidir en .5 ulp para tener p dígitos significativos en común, suponiendo que p es la cantidad de cifras utilizadas. p es un parámetro importante en la computadora pues los números disponen de espacio limitado, por lo que la cantidad de cifras es escencial.

También es posible que el número que acompaña a la potencia no esté en el intervalo $[1, 10)$, con lo que tenemos números como 23.1×10^5 o $.012 \times 10^9$. Se llama entonces *notación científica desnormalizada*, que se utiliza en la representación de números en ciertas circunstancias.

Cuando $d_0 = 0$ y el primer dígito después del punto decimal es, en este caso un dígito del 1 al 9, entonces se tiene la *notación científica normalizada*. La ventaja es que la representación de cada número es única, mientras que con

la desnromalizada se pueden tener varias representaciones distintas del mismo número. Por ejemplo el $.231 \times 10^7$ puede representarse en notación desnormalizada como $23.1 \times 10^5 = 2.31 \times 10^6 = .00231 \times 10^9 = 231 \times 10^4$.

En algunas ocasiones, también se define ufp como la unidad en la primera posición. Su valor es $\text{ufp}(x) = 2^{\lfloor \log_2 |x| \rfloor}$.

2.1.1. Notación en base 2

Como las computadoras representan toda su información con ceros y unos, llamados *bits*, es normal que el sistema numérico base 10 se sustituya con el de base 2. Los conceptos del apartado anterior siguen siendo los mismos, sólo las restricciones a los dígitos cambian. Por ejemplo, la notación científica normalizada ahora es

$$0.d_1d_2d_3...d_p \times 2^e \tag{2.2}$$

donde d_1 es distinto de cero y cada otro d_i cumple con $0 \leq d_i < 2$. Así, podemos escribir

$$0.d_1d_2...d_p \times 2^e = d_1 \times 2^e + d_2 \times 2^{e-1} + \cdots + d_p \times 2^{e-(p-1)}$$

y será sencillo representar cualquier cantidad que resulte de la suma de potencias de 2. Por ejemplo:

$$\begin{aligned} 4.75 &= 4 + .5 + .25 \\ &= 2^2 + 2^{-1} + 2^{-2} \\ &= 100.11_2 \\ &= .10011 \times 2^3 \quad \text{(normalizado)}. \end{aligned}$$

El problema es con las cantidades que no son resultado de la suma de potencias de 2, como el 1/10, el 0.34 o 1/5. En estos casos, la representación en base dos requiere de una infinidad de términos, a diferencia de la notación en base 10. No hay una base que tenga representación finita para todos los números.

Regresando a la base 2, si $x = 0.d_1d_2d_3...d_p \times 2^e$, 1 $ulp(x) = d_p \times 2^{e-(p-1)}$. El número $x + 2^{e-p}$ difiere en .5 ulp de x:

$$\begin{aligned} x &= 0.d_1d_2d_3...d_p \times 2^e \\ x + 2^{e-p} &= 0.d_1d_2d_3...d_p\mathbf{1} \times 2^e \\ &= x + 1 \times 2^{e-p} \end{aligned}$$

y el último 1 equivale a $\frac{1}{2}d_p \times 2^{e-(p-1)}$, que es la última cifra de x. La medida .5 ulp es de gran importancia pues es el error máximo que se espera en las *operaciones elementales* $(+, -, \times, \div, \sqrt{})$ con una computadora, luego del redondeo final. La meta que se busca al programar rutinas matemáticas es que todo resultado tenga una diferencia menor que .5 ulp de lo matemáticamente correcto, aunque se sabe que no siempre se puede.

A diferencia del Gulp (Giga ulp), acuñado por Fred Gustavson para referirse a librerías matemáticas con inexactitudes enormes.

Lo mejor es acostumbrarse a esta notación pues es la forma adecuada de pensar en la representación de los números si se trata de trabajar con computadoras.

Una observación importante es que por 2.2 (la definición de notación científica normalizada en base 2), todo número comienza con el dígito 1 después del punto:

$$\begin{array}{l} .\mathbf{1} \times 2^{12} \\ .\mathbf{1}0001101 \times 2^{-9} \\ .\mathbf{1}111110101 \times 2^{0} \\ .\mathbf{1}100110011 \times 2^{-4} \end{array} \quad . \tag{2.3}$$

Esto es importante porque al almacenar un número en la computadora, este primer bit no requiere ser guardado. Como siempre es uno, se ahorra un bit en la representación y se le llama *bit implícito*[4]. Este es precisamente el ufp o unidad en la primera posición.

2.2. Números de punto flotante 𝔽

Es obvio que ya sea con base 2, 10 o cualquier otra, es matemáticamente posible representar cualquier número real, aunque se requiera de una infinidad de cifras. Pero como la precisión está restringida en una computadora, no es posible representar todo el conjunto de los números reales. No se usa 2 como base definitiva pues 10 sí es importante en algunas aplicaciones, así que β sigue representando la base numérica. Al cambiar de base se degrada la precisión sólo si la exactitud de los cálculos empeora.

Se presenta a continuación el conjunto de los NPF que contiene valores normales y especiales. Un poco más adelante, con la justificación adecuada, se presentan los llamados números subnormales, en la sección 2.2.2.

En "Finger or Fists" (dedos o puños), de Buchholz [24], hay una de los primeros estudios comparativos publicados sobre la eficiencia de las bases 2 y 10, que no deja mucho qué agregar en favor de la base 10, dejándola como segunda opción, pero necesaria. También Brown y Richman [23], que justifican formalmente las bondades de la base 2, son referencia obligada.

Se denota con 𝔽 al conjunto finito de números reales que tienen representación exacta con una computadora, llamados números de punto flotante[5]. Una primera definición (por completarse) de este conjunto es

$$\mathbb{F} = \{(-1)^s \times m \times \beta^e\} \cup \overbrace{\{-0, 0, -\infty, +\infty, \mathrm{NaN}\}}^{\text{Valores especiales}} \tag{2.4}$$

donde los parámetros son

[4] En algunos textos le llaman bit fantasma o perdido.

[5] Donald Knuth da una presentación axiomática más formal en [117].

- s el *signo*,
- m la *mantisa*, un número en el rango $\frac{1}{\beta} \leq |m| < 1 - \beta^{-p}$,
- β la *base* numérica seleccionada, comunmente 2, 10 o 16,
- p la *precisión* o cantidad de dígitos empleados, comunmente 24, 53, 64 o 112 y
- e el *exponente* en el rango $e_{\text{mín}} \leq e \leq e_{\text{máx}}$.

La base 2 es con mucho la más utilizada. La base 16 poco a poco se ha dejado de usar.

Por el rango de la mantisa, se trata de un sistema normalizado[6]. NaN representa el resultado de operaciones inválidas o no definidas, como $\sqrt{-2}$, $\log(-5)$, $\cos(\infty)$ o $\infty - \infty$. Por otra parte, los símbolos $\pm\infty$ no tienen el significado matemático convencional, sino que se emplean para cualquier número real que quede fuera del rango de $\mathbb{F}$ después del redondeo.

Con esta notación, el menor número es el $\beta^{e_{\text{mín}}}$ y el mayor el $\beta^{e_{\text{máx}}}(1-\beta^{-p})$ ya que $1-\beta^{-p}$ representa el mayor valor posible para la mantisa. Por ejemplo, en un sistema numérico en base 2 donde se tengan 3 dígitos de precisión y un rango de exponentes entre -2 y 2, tendríamos números de la forma $\pm.d_1d_2d_3 \times 2^p$. El menor es $2^{-2} = .1 \times 2^{-2} = .25$, mientras que el mayor es el $2^2(1-2^{-3}) = .111 \times 2^2 = 3.5$.

A partir de estos valores se pueden calcular los límites o el intervalo real al que pertenecen los NPF: $\mathbb{F} \subsetneqq [-\beta^{e_{\text{máx}}}(2-2^{1-p}), +\beta^{e_{\text{máx}}}(2-2^{1-p})]$. Se usa $\subsetneqq$ porque en ese intervalo existe una infinidad de números reales que no pertenecen a $\mathbb{F}$. Todo número fuera de ese intervalo quedará representado por $\pm\infty$, salvo posibles redondeos. Más específicamente, cuando el resultado de una operación resulta en un exponente mayor a $e_{\text{máx}}$, el resultado será $\pm\infty$, dependiendo del signo s de la mantisa y se dice que ocurre un *overflow* (desborde).

Con algunos lenguajes de programación, en los números enteros, al sumar uno al entero mayor se obtiene el entero menor, pero no en las variables de punto flotante.

Si por el contrario, la operación genera un exponente menor que $e_{\text{mín}}$ el resultado se hace 0 y y se dice que ocurre un *underflow*.

Obsérvese que si se usan los p dígitos de precisión en la notación normalizada, entonces $m\beta^p$ es un número entero y además $-\beta^p < m\beta^p < \beta^p$. Por ejemplo, en base 10, $.1234 \times 10^4 = 1234$ y

$$-10^4 < 1234 < 10000.$$

Para ser realistas con la representación de las computadoras, sólo falta considerar el bit implícito (ver 2.3) en la notación de $\mathbb{F}$ y los números subnormales, pero se hará en su momento.

[6] La definición de los NPF no es única. Es común ver que la mantisa se presenta como entero, por lo que los números toman la forma $m \times \beta^{e-p}$, pero es equivalente al presentado.

Por completitud, se define ulp(NaN) = NaN, ulp(∞) = $\beta^{e_{\text{máx}}}(1-\beta^{-p})$.

Es posible que las operaciones entre NPF produzcan resultados que no pertenecen a 𝔽. Por ejemplo, $x = 0.x_1x_2...x_p \times \beta^{e_x}$ y $y = 0.y_1y_2...y_p \times \beta^{e_y}$ son elementos de 𝔽 y sea $e_x - e_y = p+1$, es decir, $|x| > |y|$.

Para realizar la operación $x+y$ se requiere poner a y con el mismo exponente de x, lo que se conoce como *normalización*. El resultado requiere una mantisa de $2p$ dígitos en base β pues de lo contrario hay pérdida de cifras significativas.

$$\begin{aligned} x+y &= 0.x_1x_2...x_p \times \beta^{e_x} + 0.y_1y_2...y_p \times \beta^{e_y} = \\ &= 0.x_1x_2...x_p \times \beta^{e_x} + 0.\overbrace{000...0}^{p \text{ ceros}}y_1y_2...y_p \times \beta^{e_x} \\ &= 0.x_1x_2...x_py_1y_2...y_p \times \beta^{e_x} \\ &= 0.x_1x_2...x'_p \times \beta^{e_x} \qquad \text{¡Desapareció } y! \end{aligned} \tag{2.5}$$

y x'_p es posiblemente afectado por el redondeo, y de no ser así, $x + y = x$. Ejemplos similares pueden inventarse para cada operación elemental. Esa es una muestra de los problemas de exactitud de 𝔽.

El valor de los parámetros β, p, $e_{\text{mín}}$ y $e_{\text{máx}}$ define el conjunto 𝔽 con el que cada computadora trabaja. Si dos computadoras se fabrican con valores distintos de los parámetros, el mismo programa no dará los mismos resultados como consecuencia de las diferencias. Esto llegó a ocasionar grandes problemas de portabilidad de código y de dificultad de elaborar librerías matemáticas robustas en el inicio de la programación de computadoras [100]. De hecho el problema persiste pero ya se dipone de convenciones que facilitan las cosas.

Esto implica que las computadoras tienen diferente arquitectura. También depende del objetivo de cada computadora.

2.2.1. Valores importantes y propiedades del redondeo

La distancia entre 1 y el siguiente NPF es β^{1-p} y es llamada *épsilon de máquina*, que se denotará por ε_M. En general, el espacio entre el NPF x y el siguiente es al menos $\frac{\varepsilon_M}{\beta}|x|$ y a lo más $\varepsilon_M|x|$, salvo para los valores especiales.

La cota máxima no es difícil de ver ya que si x tiene la menor mantisa ocurre $x = .1 \times \beta^e$ y la distancia al siguiente número de punto flotante es

$$x\beta^{1-p} = .\overbrace{000...01}^{p-1 \text{ceros}} \times \beta^{e+1-p}$$

que es precisamente ulp(x).

Es sencillo calcular ε_M en un programa:

Listing 2.1: Calculando ε_M (épsilon de máquina).

```
1 double e = 1.0;
2 while ( 1+e != 1 )
3       e = e/2;
4 e = e*2;
```

El final, el valor de `e` será ε_M para el tipo de dato de la variable `e`. En algunos ambientes de programación, esta es una constante ya definida. Por ejemplo, en Matlab es **eps**. A partir de ε_M es posible determinar la base numérica empleada en la representación interna por la computadora pues $\beta = \frac{1+\varepsilon_M}{1-.5\varepsilon_M}$. Si al NPF más pequeño se le divide entre $1 - .5\varepsilon_M$ se obtiene una buena cota para determinar qué tan significativo es el underflow.

A la cantidad

$$\mu = \frac{1}{2}\varepsilon_M$$

se le llama *unidad de redondeo* y tiene gran importancia en todo análisis de errores. También equivale a .5 ulp(1.0).

Cada número real $x \in \mathbb{R}$ tiene al menos un representante en $\mathbb{F}$ que se obtiene mediante la función $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$, como $\mathrm{fl}(x) \in \mathbb{F}$. Se dice que $\mathrm{fl}(\mathbb{R}) = \mathbb{F}$.

El concepto de redondeo es central en programación numérica

Cuando un número real puede ser representado con dos NPF con el mismo error, es decir, que queda a la misma distancia de los dos NPF más cercanos, el representante es seleccionando dependiendo de la regla de redondeo que se utilice, pudiendo ser siempre al mayor, siempre al menor, hacia aquel cuyo último dígito es 0 u otra.

El valor original x y el representante seleccionado $\mathrm{fl}(x)$ deben diferir en a lo más 1 ulp, por definición. La transformación de x a $\mathrm{fl}(x)$ se llama *redondeo* y están relacionados por:

$$\mathrm{fl}(x) = x(1 + \delta)$$

donde $|\delta| < \mu$. Esta propiedad (y notación) ha sido empleada desde los primeros años del análisis de errores (véase [187]), generalmente atribuida a Wilkinson y es de gran utilidad. Al despejar δ se obtiene el error relativo $|\mathrm{fl}(x) - x| / x$.

Si no se selecciona el número más cercano y simplemente se descartan las cifras excedentes dejando sólo las p del formato, entonces el proceso se llama *truncamiento*, que posee un error relativo mayor. El error relativo para el truncamiento está acotado por ε_M y no por μ. Las primeras computadoras utilizaron este esquema.

Las reglas de monotonicidad se cumplen, es decir, si $a \in \mathbb{F}$, $b \in \mathbb{F}$ y $x \in \mathbb{R}$ entonces

$$\begin{aligned} a \le r \le b \quad &\Rightarrow \quad a \le \mathrm{fl}(r) \le b \\ a < r < b \quad &\Rightarrow \quad a < \mathrm{fl}(r) < b. \end{aligned}$$

Es fácil ver que $\mathrm{fl}(-x) = -\mathrm{fl}(x)$, o económicamente, $\mathbb{F} = -\mathbb{F}$.

2.2.2. Distribución de los Números de Punto Flotante $\mathbb{F}$

A partir de la definición de $\mathbb{F}$ (ver 2.4) se ve que los mismos valores de mantisa m, desde $1/\beta$ hasta $1 - \beta^{-p}$, se repiten al pasar a otro exponente,

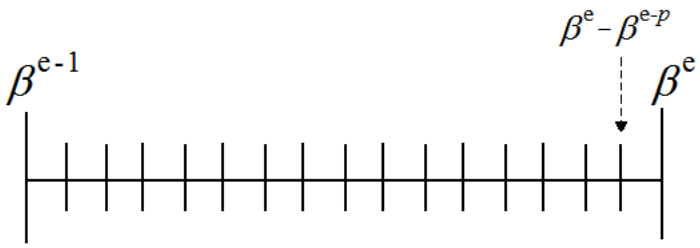


Figura 2.1: Distribución de 𝔽. Los NPF entre 2 potencias contiguas de β están uniformemente distribuidos, por lo que el error relativo va aumentando en ese intervalo.

representando otros números. Para un exponente fijo e, la menor mantisa representa al número β^{e-1} y la mayor mantisa al $(1-\beta^{-p})\beta^e = \beta^e - \beta^{e-p}$ que queda exactamente a la izquierda de β^e, la siguiente potencia. Esto se ilustra en la figura 2.1.

Esto significa que aunque las potencias de β se van separando, la cantidad de representantes entre cada par de potencias se mantiene constante: son β^p. El error absoluto entre dos NPF es el mismo (vale $(\beta^e - \beta^{e-1})/\beta^p$) entre cada par de potencias de β y se duplica al pasar a la siguiente potencia.

Este vale $(\beta^e - \beta^{e-1})/\beta^p$, es decir,

$$\frac{\beta^{e_{\text{mín}}+1} - \beta^{e_{\text{mín}}}}{\beta^p} \quad \text{mínimo y}$$
$$\frac{\beta^{e_{\text{máx}}} - \beta^{e_{\text{máx}}-1}}{\beta^p} \quad \text{máximo.}$$

Con el error relativo ocurre algo distinto. Para toda x, sin importar entre qué potencias de β^{e-1} y β^e se encuentre, se tiene que

$$\left|\frac{x - \text{fl}(x)}{x}\right| \approx \frac{1}{2}\beta^{1-p} \tag{2.6}$$

por lo que el error relativo es múltiplo de la base numérica empleada. Por ello, se prefiere utilizar la base 2 para reducir este fenómeno, conocido como *wobbling* (tambaleo), que consiste en el comportamiento no suave del error relativo en la representación de x con $\text{fl}(x)$. El error puede ser 0 (para las potencias de 2) o puede ser tan grande como la unidad de redondeo (en el caso del mínimo NPF mayor que x). En la figura 2.2, el wobbling (2.6) se ve en las marcas grandes que representan potencias de β, pues la distancia a la marca de la izquierda es la mitad de la distancia a la marca de la derecha.

Los NPF que representan números reales con menor error relativo para cada potencia son lo que tienen mantisa de mayor valor.

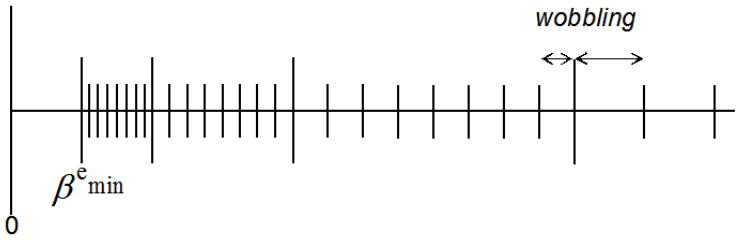


Figura 2.2: Wobbling o tambaleo. Consiste en la reducción abrupta del error relativo en la aproximación de x con $\text{fl}(x)$.

Observando de nuevo la figura 2.2, es evidente un vacío entre el 0 y $\beta^{e_{\text{mín}}}$. Ese intervalo no tiene representantes en $\mathbb{F}$ pues cualquier resultado con exponente menor a $e_{\text{mín}}$ sufre un underflow. Ese intervalo real $(0, \beta^{e_{\text{mín}}})$ no posee representantes en en $\mathbb{F}$, ocasionando problemas de precisión y dificultando el análisis de errores.

Por ejemplo, la comparación `if(x==y)` no es equivalente a `if(x-y==0)` con ese vacío alrededor del 0. Cualquier número en este intervalo se representará con 0 o con $\beta^{e_{\text{mín}}}$, con un error tan grande como $\frac{1}{2}\beta^{e_{\text{mín}}}$. Eso significa que el número real $\frac{1}{2}\beta^{e_{\text{mín}}}$ se representará con un porcentaje de error de 100 %.

El coprocesador 8087 de Intel fue el primero que los usó.

Desde antes de 1980 se tomaron medidas al respecto. La solución, que con los años resultó ser la mejor, fue la de agregar a $\mathbb{F}$ el conjunto de *números subnormales*[7]

$$\mathbb{U} = \{(-1)^s \times m \times \beta^{e-p}\}$$

donde ahora se relaja la restricción de la mantisa y $0 < m \leq \frac{1}{\beta}$, lo que significa que no está en notación científica normalizada. Esto permite que al intervalo $(0, \beta^{e_{\text{mín}}})$ se agreguen $\beta^p - 1$ valores igualmente distribuidos (sólo la combinaciones formada por puros 0s no se cuenta). A esto se le llama *underflow gradual*. Entre otras cosas, esto significa que no hay bit implícito en los números subnormales.

Como la distancia entre ellos es fija (vale $\beta^{e_{\text{mín}}}/(\beta^p - 1)$), el error relativo aumenta conforme los valores se acercan a 0. Ahora el conjunto $\mathbb{F}$ está completo y luce como la figura 2.3. La región sombreada representa los números subnormales. Además de rellenar ese espacio vacío, los subnormales tienen la ventaja de no sufrir de wobbling. Ahora que ya está completo el conjunto de los NPF $\mathbb{F}$, `if(x==y)` ya es equivalente a `if(x-y==0)`.

El menor número subnormal es referido por muchos autores como $\eta = \beta^{1-e_{min}-p}$. Aparece menos frecuentemente que μ y ε_M, pero resulta útil cuando se analizan algoritmos con posibles underflows.

Para todo $x \in \mathbb{F}$, debe cumplirse que si $a = (x - \text{ulp}(x))$ y $b = (x + \text{ulp}(x))$,

[7] Llamados originalmente desnormalizados.

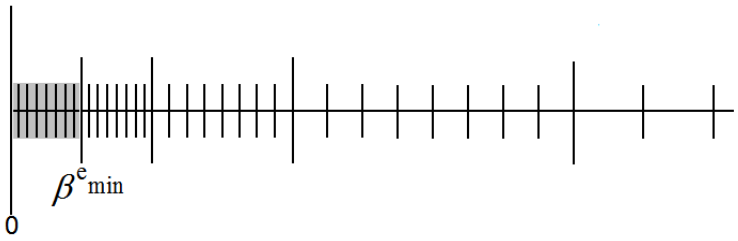


Figura 2.3: El conjunto $\mathbb{U}$ queda representado por la región sombreada que rodea al 0, aunque aquí sólo se muestra el eje positivo.

entonces se cumple que

$$\text{ó} \quad a < x \leq b \quad \text{ó} \quad a \leq x < b. \tag{2.7}$$

Esta regla aplica incluso para los valores especiales de 2.4. Aunque esto aparente tener fines exclusivamente teóricos, las señales de Operación Inválida de todo programa pueden tener sus orígenes en la falta de cuidado al no tener en cuenta propiedades tan simples como 2.7.

No es difícil encontrar ejemplos numéricos desagradables en $\mathbb{F}$, como ya se vio en la suma $x + y$ en la página 31. Si se calcula el promedio de dos número como $m = (x + y)/2$, pueden ocurrir varias cosas:

- $x + y$ resulta en overflow, aún cuando $m \in [x, y]$, por lo que no se puede realizar el cálculo.
- Si se intenta $x/2 + y/2$ y los x y y son el número subnormal más pequeño, los cocientes resultan en underflow, por lo que $m = 0$ y no $m = x = y$, que es lo exacto.
- En una base distinta a 2, el resultado puede estar fuera del intervalo $[x, y]$. Por ejemplo, en base 10 con de 4 decimales, sean $x = .1 \times 10^1$ y $y = .9999 \times 10^0$. Su promedio es

$$\begin{aligned} m &= \frac{1}{2}(x + y) \\ &= \frac{1}{2}(.1 \times 10^1 + .9999 \times 10^0) \\ &= \frac{1}{2}(.1 \times 10^1 + .09999 \times 10^1) \\ &= \frac{1}{2}(.1 + .09999) \times 10^1 \\ &= \frac{1}{2} \times .1999 \times 10^1 \\ &= .9995 \times 10^0 \end{aligned}$$

que queda fuera de [.9999,.1], si se redondea por truncamiento. Aún si se redondea al más cercano, el resultado queda

$$\begin{aligned} m &= \frac{1}{2}(.1 + .09999) \times 10^1 \\ &= \frac{1}{2} \times .19999 \times 10^1 \\ &= .\,999\,95 \times 10^0 \\ &= .1 \times 10^1 \end{aligned}$$

que es un extremo del intervalo y no el punto medio, pero claro, el conjunto de NPF determinado por $\beta = 10$ y $p = 4$ no se puede representar a .99995 .

2.3. Propiedades de $\mathbb{R}$ y $\mathbb{F}$

El conjunto de los números reales cumple con propiedades que se definen sobre las operaciones básicas. En resumen es un campo ordenado, denso y no numerable[8], (un grupo conmutativo con respecto a suma y multiplicación). Sean x, y y z números reales cualquiera. Se enumeran a continuación las propiedades básicas de $\mathbb{R}$ con fines de comparación con $\mathbb{F}$. Por ello se considera al conjunto de los reales extendidos $\bar{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$. Todo usuario de computadoras desearía que estas propiedades se cumplieran con los NPF.

Principalmente los programadores.

2.3.1. Propiedades de los reales extendidos

Propiedades de la suma

1. **Cerradura**. La suma $x + y$, también es real.

2. **Identidad**. Hay un único número, llamado cero (0) con la propiedad de que $x + 0 = 0 + x = x$.

3. **Inversos**. Para cada numero x, existe un único número denotado por $-x$, tal que $x + (-x) = (-x) + x = 0$.

4. **Asociatividad**. Se cumple $(x + y) + z = x + (y + z)$.

5. **Conmutatividad**. Se cumple $x + y = y + x$.

Propiedades de la multiplicación

6. **Cerradura**. El producto xy, también es real.

7. **Identidad**. Hay un único número, llamado uno (1) tal que para todo x se cumple $1x = x1 = x$.

[8] Consultar cualquier libro de álgebra o cálculo.

8. **Inversos**. Para cada x,existe un único número, denotado con x^{-1}, tal que $xx^{-1} = x^{-1}x = 1$.

9. **Asociatividad**. Se cumple $(xy)z = x(yz)$.

10. **Conmutatividad**. Se cumple $xy = yx$.

Además, la suma y la multiplicación cumplen con otro axioma:

11. **Distributividad**. Se cumple $x(y + z) = xy + xz$.

Los axiomas anteriores definen un campo.

Propiedades de orden

Un campo es totalmente ordenado si se definen las relaciones $<$, $=$ y $>$ con las propiedades:

12. **Tricotomía**. Se cumple una y sólo una de las condiciones: $x < y$, $x > y$ o $x = y$.

13. **Reflexividad**. Si $x < y$, entonces $y > x$ y si $x = y$, entonces $y = x$.

14. **Transitividad**. Si $x < y$ y $y < z$ entonces $x < z$. Si $x = y$ y $y = z$ entonces $x = z$.

Relaciones de orden y las operaciones de grupo.

15. **Invarianza a traslaciones**. $x + z < y + z \iff x < y$. Además, $x + z = y + z \iff x = y$.

16. **Invarianza a escalas**. $xz < yz \iff x < y$. Además, $xz = yz \iff x = y$.

Propiedad de Densidad.

17. **Densidad**. Si $x < y$ entonces existe z tal que $x < z < y$.

Los *números racionales* son todos los de la forma a/b, donde tanto a como b son enteros, $b \neq 0$. Otra propiedad de los números reales es que son no numerables: son infinitamente muchos más que los números racionales.

Con las sólidas propiedades anteriores se han desarrollado gran cantidad de teorías matemáticas. Remover alguna propiedad puede tener repercusiones inesperadas.

2.3.2. Propiedades de $\mathbb{F}$

Siendo $\mathbb{F}$ un subconjunto de los números reales, sería deseable que cumpliera con propiedades similares pero no es posible. A partir de la definición 2.4 se sabe que $\mathbb{F}$ es un conjunto finito. Con más exactitud que antes, $\mathbb{F}$ es un subconjunto de los números racionales, no inlcuye ningún numero irracional.

De los parámetros de $\mathbb{F}$ se puede deducir que NO se cumplen varias propiedades: 1, 4, 6, 9, 11, 15 y 17 sin incluir los valores especiales, donde NaN tampoco cumple 12, 13, 14 y 16. Las propiedades 2, 3 y 8 se cumplen parcialmente. Es un buen ejercicio encontrar contraejemplos.

Por su significado, NaN no está ordenado, lo que implica que si x es NaN, las relaciones $x < y$, $x = y$ y $x > y$ son todas falsas. La única correcta es $x <> x$ (x distinto de x). Esto da más claridad al anterior párrafo.

Cambia por completo el panorama de la matemática que se puede desarrollar con $\mathbb{F}$ pues en general, sólo 7 propiedades se cumplen de alguna manera útil. La desigualdad del triángulo o la de Cauchy-Schwartz, con tantas aplicaciones, son otros ejemplos de propiedades no válida para los NPF.

Por otra parte, disponer de ∞ hace que la división entre cero sea una operación válida en $\mathbb{F}$, aunque genera otras indefiniciones, como $\infty - \infty$, ∞/∞ o $\infty \times 0$. Este símbolo NO representa lo indefinido.

Para poder analizar y cuidar los errores en procesos numéricos, se necesita que para todas las operaciones elementales $(+, -, \times, \div)$ se cumpla

$$\mathrm{fl}(x \odot y) = (x \odot y)(1 + \delta), \quad |\delta| \leq \mu$$

como ya se dijo y $\odot$ es cualquiera de las operaciones elementales[9]. Este es el modelo de aritmética de punto flotante (APF) más ampliamente utilizado, aunque no es el único posible.

Sean x y y dos números reales en el rango normal de $\mathbb{F}$. Su suma es

$$\begin{aligned}
\mathrm{fl}(x) + \mathrm{fl}(y) &= x(1+\delta_x) + y(1+\delta_y) \\
&= x + y + x\delta_x + y\delta_y \\
&\leq x + y + x\delta + y\delta, \qquad \delta = \max(|\delta_x|, |\delta_y|) \leq \mu \\
&= x + y + (x+y)\delta \\
&= (x+y)(1+\delta) \\
&= \mathrm{fl}(x+y)
\end{aligned}$$

y de esta manera, el error relativo δ de la suma cumple con las características de los NPF. También podría utilizarse la descripción más exacta $x = 0.x_1x_2...x_p \times \beta^{e_x}$, pero como se vio, no es necesario.

Una propiedad útil que se puede aplicar en otras operaciones es que $\mathrm{fl}(-x) = -\mathrm{fl}(x)$, así como $\mu^2 \approx 0$, que se aplica en el caso del producto.

Otra forma práctica de denotar el error relativo que no causa confusiones es escribir $\delta_\odot$ y similarmente para otras operaciones. De esta manera, en vez de escribir $\mathrm{fl}(x+y) = (x+y)(1+\delta)$ se escribe de manera compacta como δ_{x+y}. Por ejemplo, sin tratarse ahora de una operación elemental, supongamos el mismo

[9] En la práctica también se agrega $\sqrt{}$

error en la aproximación de x y y:

$$\begin{aligned} \text{fl}(xy\sqrt{x}) &= xy\sqrt{x}(1+\delta)(1+\delta)(1+\delta_{\sqrt{x}}) \\ &= xy\sqrt{x}(1+\delta)^2(1+\delta_{\sqrt{x}}) \\ &\approx xy\sqrt{x}(1+2\delta+\delta_{\sqrt{x}}) \end{aligned}$$

pues los términos cuadráticos se desprecian por $\mu^2 \approx 0$. Entonces, $\delta_{xy\sqrt{x}} = 2\delta + \delta_{\sqrt{x}}$. Es evidente que esta compacta notación no causa confusiones. Donald Knuth la utiliza en [117] y también otros más recientes como Hull en [92].

El caso de la resta merece especial atención. Supongamos que en base 2, con $p = 4$ se tiene $x = .1000 \times 2^1$ y $y = .1111 \times 2^0$. Para calcular su resta, primero se igualan los exponentes a 2^1 y se restan las matisas:

$$\begin{array}{rr} & .1000 \\ - & \underline{.01111} \\ & .00001 \end{array}$$

El resultado se normaliza sin necesidad de redondeo (en este caso), $x - y = .1 \times 2^{-3}$. Para poder realizar la resta es necesario que se use la mantisa intermedia .01111 que requiere $p + 1$ dígitos. Este dígito extra temporal es conocido como *dígito de guardia* y es obligatorio para que el error continúe siendo menor que μ.

Si no se utiliza el dígito de guardia, al igualar los exponentes se truncaría el último dígito de y, quedando la resta como

$$\begin{array}{rr} & .1000 \\ - & \underline{.0111} \\ & .0001 \end{array}$$

y el error es el doble (apenas se cumple $|\delta| \leq 2\mu$). Si se redondea en vez de truncar, la resta es 0.

En la actualidad, son pocas las computadoras que carecen de dígito de guardia (como algunos de los primeros modelos de las supercomputadoras Cray[10]) e incurren en resultados con errores relativos inesperadamente altos. Por ejemplo, de nuevo con la base $\beta \geq 2$, sea $x = .1$ y $y = x - \beta^{-p} = .0111...1$ (p dígitos). La diferencia exacta $x - y$ es β^{-p}, pero si no hay dígito de guardia, el último bit de y se corta, por lo que la diferencia es β^{1-p}. El error relativo es

$$\begin{aligned} \frac{\beta^{1-p} - \beta^{-p}}{\beta^{-p}} &= \frac{\beta^{-p}(\beta - 1)}{\beta^{-p}} \\ &= \beta - 1 \end{aligned}$$

[10]La Cray 1 en particular fue una queja constante de los analístas numéricos, pero incluso las X-MP y C90 no se escaparon de críticas, no solo por dígitos de guardia, sino por su modo de redondeo. El interés comercial por la velocidad a costa de todo tuvo sus perjuicios.

lo que es un error relativo enorme: en base 2 el error es tan grande como el resultado y si $\beta = 10$, ¡sería 9 veces mayor! La necesidad del dígito de guardia es evidente, para evitar lo que se conoce como *cancelación catastrófica*. El nombre suena alarmante, pero es más por el hecho de que es la operación aritmética que mayor error relativo puede alcanzar si se realiza mal, cosa que es de esperarse que ningún fabricante de computadoras realice actualmente.

Tal vez se pueda esperar la cancelación catastrófica en los procesadores gráficos (GPU), donde la precisión queda en segundo término pues se requiere que todas las transformaciones geométricas (típicamente en 3D) ocurran a la más alta velocidad posible. La APF de las tarjetas gráficas es más débil pero mucho más veloz, usando valores bajos de p.

Hay dos resultados importantes relacionados con el dígito de guardia, para cualquier β que se utilice.

Primeramente, Ferguson probó que si $x = m_x \times \beta^{e_x}$ y $y = m_y \times \beta^{e_y}$ son dos NPF para los que el exponente de $x - y \leq \text{mín}(e_x, e_y)$, y la diferencia se calcula con $p + 1$ dígitos de precisión, entonces $x - y$ da un resultado exacto.

Como corolario, Sterbenz demostró que si $y/2 \leq x \leq 2y$ y hay al menos $p+1$ dígitos de precisión intermedia, entonces $x - y$ da un resultado exacto. En este caso, la condición es fácil de verificar en un programa.

El dígito de guardia no fue suficiente en la práctica para redondear con eficiencia, por lo que tuvo que utilizarse una extención de esa idea. Esta se presentará con la norma de punto flotante, en los modos de redondeo, en la sección 3.2.3.

2.4. Otras aritméticas

También se han propuesto APF que utilizan un sistema numérico distinto a 2.1. Se han buscado sistemas que signifiquen alguna mejora de alguno de los aspectos de la APF convencional, sin empeorar sus virtudes (al menos no sustancialmente). Mientras algunos solo son nuevos sistemas de cálculo, algunos implican también otro formato. Ejemplo de esto son las fracciones continuas

$$a + \frac{1}{b + \frac{1}{c + \cdots}}$$

que tienen respuesta numérica excelente. Siendo precisas y veloces son poco utilizadas. El intento de representar números racionales a/b como la pareja de enteros también fue estudiada pero no se presenta aquí (véase el estudio de Peter Henrici [85]). A continuación se presentan los sistemas más importantes (o mejor conocidos).

2.4.1. Sistema logarítmico

Este es el caso del *sistema numérico logarítmico* (SNL), que requiere dos bits adicionales como banderas para señalar excepciones. El rango numérico es similar, así como la precisión, pero es mucho más sencillo realizar algunas operaciones. En general el formato es

$$x = (-1)^s \times n.f$$

donde $n.f$ es la mantisa con su partes entera y fraccionaria, codificadas en complemento a 2. No se requiere conservar exponente pero sí dos bits para excepciones. Este formato no es único, también pudiera ser $x = \pm\beta^i$, con β ligeramente mayor que 1.)

Lo que se necesita es transformar al logaritmo (normalmente base 2) de los números iniciales y realizar todo los procesos numéricos para sólo hacer la conversión inversa al final de los cálculos. Por ejemplo, si $x = m_x \times \beta^{e_x}$ y $y = m_y \times \beta^{e_y}$ ya están en sistema logarítmico, el producto xy se obtiene con la suma de mantisas pues $\log(xy) = \log x + \log y$, por lo que hay ahorro de tiempo, espacio en la ALU[11] del procesador y energía eléctrica.

Lo mismo ocurre con la división pues $\log(x/y) = \log x - \log y$, por lo que el costo de $\times$ y $\div$ equivale al de la suma y resta respectivamente.

Los problemas son con la representación de enteros pequeños, la suma y la resta en el SNL: si $x > y$

$$\log|x + y| = x + \log\left|1 + 2^{y-x}\right|$$

y 2^{y-x} se calcula con interpolación, basándose en una tabla de valores precalculada. El costo es menor aún que el de la multiplicación convencional, así que el SNL sigue siendo más eficiente, pero sólo si no se requiere gran cantidad de decimales.

El hecho de que números como 2 o 3 no puedan representarse con exactitud sí es algo que no es conveniente.

Una buena característica del SNL es que el error relativo entre dos números vecinos se mantiene constante, por lo que no hay wobbling (ver 2.6 en la página 33). La distancia crece de manera contínua y no discretizada, como ocurre en IF al pasar de una potencia de β a otra. Y sí hay mucha diferencia entre la constante y el error relativo con wobbling, como se muestra en la figura 2.4.

El SNL es utilizado más ampliamente en Europa, aunque poco a poco gana adeptos también en América y Asia. Puede consultarse [175] para mejores referencias. Además, Higham [88] presenta otro modelo logarítmico adicional.

[11] Arithmetic Logic Unit (Unidad Aritmético Lógica), encargada de las operaciones numéricas.

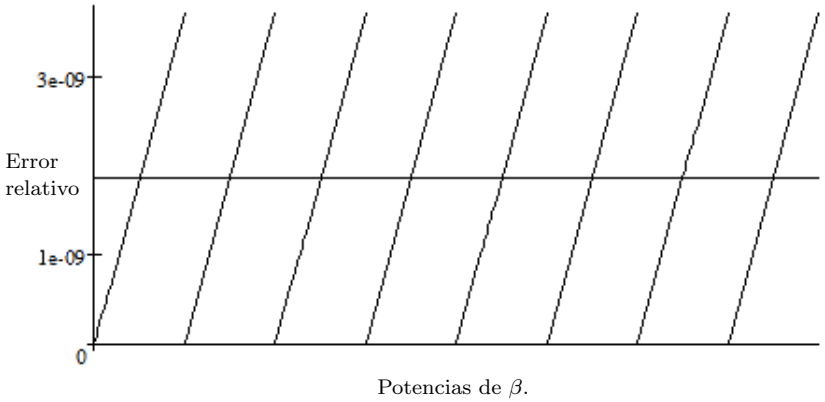


Figura 2.4: El error relativo del Wobbling o tambaleo. La línea horizontal representa el error relativo constante, comparado con el wobbling (líneas en diagonal.)

2.4.2. Sistema exponencial

Otro ejemplo de aritmética es la que propusieron Clenshaw y Olver en [30], con la intención de ampliar el rango de números representables, un poco a costa de la precisión. La idea es representar cada número x como

$$x = e^{e^{\cdots^{e^a}}}$$

donde el proceso de exponenciación se realiza l veces. Cuando $l = 0$ se tiene $x = a$. Clenshaw justifica que para fines prácticos no es necesario pasar de $l = 4$ para representar a todos los números reales. Con 3 bits que indiquen los niveles, el overflow estaría olvidado del cómputo convencional.

Este sistema es también llamado *índice de nivel simétrico*. A diferencia del SNL, el exponencial ha prosperado muy poco.

2.4.3. Aritmética de intervalos

Cada $\text{fl}(x)$ representa una aproximación del número real x, cuya exactitud depende de diversos factores. En un proceso de cálculo en general, no se puede estar del todo seguro de qué tan cerca están uno del otro y ni siquiera si $x < \text{fl}(x)$ o $x > \text{fl}(x)$. Este es un problema intrínseco de la APF.

Una forma de evitar esta incertidumbre es no trabajar con $\text{fl}(x)$ sino con el intervalo más pequeño que lo contiene, idea surgida en la década de los 60s. Así, si los dos NPF más cercanos a x son $\underline{x}$ y $\overline{x}$, con $\underline{x} < \overline{x}$, entonces $[\underline{x}, \overline{x}]$ es un intervalo donde es seguro que se encuentra x. Típicamente, se representa $[x] = [\underline{x}, \overline{x}]$.

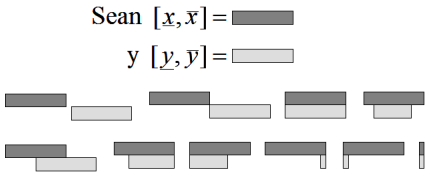


Figura 2.5: Los rectángulos representan intervalos con diversos límites.

Su *longitud* se define como $|[x]| = \overline{x} - \underline{x}$ y es puede ser cero sólo antes de la primera operación.

De esta forma, en vez de representar a x con una cantidad cuya exactitud se ignora, se le representa con un intervalo donde se tiene la seguridad que está contenido x. Con este nuevo concepto $[x]$, se desarrollan propiedades de un nuevo conjunto formado por todos los intervalos reales

Error máxino contra error desconocido.

$$\mathbb{I}\mathrm{R} = \{[a, b] \subseteq \mathbb{R}\}.$$

Por ejemplo, las operaciones aritméticas elementales se realizan de la siguiente manera.

$$\begin{aligned}
[x] + [y] &= \left[\underline{x} + \underline{y}, \overline{x} + \overline{y}\right] \\
[x] - [y] &= \left[\underline{x} - \overline{y}, \overline{x} - \underline{y}\right] \\
[x] \times [y] &= [\min\{\underline{x}\,\underline{y}, \underline{x}\,\overline{y}, \overline{x}\,\underline{y}, \overline{x}\,\overline{y}\}, \max\{\underline{x}\,\underline{y}, \underline{x}\,\overline{y}, \overline{x}\,\underline{y}, \overline{x}\,\overline{y}\}] \\
[x]^{-1} &= [1/\overline{x}, 1/\underline{x}] \\
[x] \div [y] &= [x] \times [y]^{-1}.
\end{aligned}$$

Los problemas surgen al momento de comparar $[x]$ con $[y]$ pues la disposición de los intervalos tiene muchas variantes. Un caso sencillo es cuando $[x] < [y]$ pero decidir cuando $\underline{y} < \overline{x}$ o $[x] \subset [y]$ no es tan claro. Además, parece que el punto medio de los intervalos debe tener su importancia al determinar la relación de orden. La figura 2.5 se muestra los posibles casos, faltando sólo los simétricos.

De cualquier manera siempre se dispone de un representante: el punto medio $\mathrm{fl}(x) \approx (\underline{x} + \overline{x})/2$, excepto cuando es demasiado pequeño y el punto medio coincide con alguno de los extremos o la tendencia es mayor que cierta tolerancia[12].

Las primeras investigaciones serias datan de la década de 1920, por ejemplo [27], con carácter teórico y sin preveer la utilidad que tendrían posteriormente. Por no ser un concepto tan nuevo, ya hay gran cantidad de teoría matemática en la que la computación moderna se apoya. Hay versiones para intervalos de gran cantidad de resultados sobre los reales, conjuntos y otras estructuras. Un

[12] Aquí hay más posibilidades de desarrollar cosas.

texto clásico sobre el tema es el de Alefeld y Herzberger [67]. Más actualizado y con el tratamiento correcto de los números especiales de $\mathbb{F}$ es [86].

Si no se utiliza bien, es fácil terminar los cálculos con intervalos muy grandes, lo que es perjudicial si se esperan intervalos de poca longitud (ver [107]). En general se le considera una aritmética pesimista, pero útil en combinación con otras técnicas.

Lo ideal es que, en los cálculos, la longitud del intervalo que contiene a x sea $|[x]| = ulp(x)$. Esto se cumple luego de la primera operación, pero conforme se realizan más operaciones, la longitud aumenta.

El mayor inconveniente es el aumento en el tiempo de cómputo (debido tanto al cálculo doble como al cambio constante de modo de redondeo del procesador), pero los defensores de la aritmética de intervalos lo justifican alegando

- la velocidad de las computadoras modernas y
- tener la seguridad de en qué intervalo está el resultado.

Lo cierto es que la aritmética de intervalos tiene más seguidores en Europa, pero su importancia es suficiente como para que instituciones y organismos la consideren en toda norma y especificación seria de lenguajes científicos.

Funciones como $f(x) = \cos(e^x)$ son más fáciles de graficar usando aritmética de intervalos.

Algunas propiedades estructurales de las funciones, como la inyectividad, difíciles de determinar con programas, han probado ser un tanto más sencillas con intervalos. Por ejemplo, el trabajo de Lagrange [120][13], donde no sólo se define $f([x])$ sino $[f]([x])$. Incluso extienden el concepto a inyectividad parcial para lograr una respuesta correcta siempre.

En la práctica, si se dispone de un ambiente de programación donde se controle el modo de redondeo, entonces toda operación aritmética puede realizarse primero redondeando hacia $+\infty$ y luego hacia $-\infty$ para obtener el intervalo que contiene el resultado.

Además de una amplia literatura, ya hay disponibles librerías matemáticas gratuitas para poder programar cómputo sobre intervalos. Por ejempo C-XSC de la universidad de Karlsruhe[14], flib++[15] [124] de la Universidad de Berische o Gaol, desarrollada en C++ por Frederic Goualard. Ambas poseen tipos de datos predefinidos para dar falicilidades al programar con intervalos.

En Fortran 90 está dispobible la INTLIB [114]. Más recientemente surgió la CoStLy[16] [141], con gran variedad de funciones elementales para números complejos. En este caso, los intervalos se convierten en un rectángulo sobre los números complejos.

[13] Las pocas referencias sobre el tema hablan de lo novedoso del tema.

[14] A la aritmética de intervalos también se le llama aritmética de Karlsruhe, donde el Prof. U. Kulish, del Instituto de Matemática Numérica ha sido un gran impulsor. Leéanse sus Letters to the IEEE Computer Arithmetic Standards Revision Group, disponibles en muchos sitios de internet.

[15] Originalmente desarrollada en Karlsruhe por Hofschuster y Krämer.

[16] También de la Universidad de Karlsruhe.

Ya hay disponibles calculadoras de escritorio que utilizan aritmética de intervalos, como ItvCalc, originalmente para sistema operativo Unix, pero ya con versión para Windows.

En ambos casos se dispone de una gran cantidad ejemplos, desde aproximación de raíces, sistemas de acuaciones lineales, optimización, ecuaciones diferenciales, integración y muchos más.

2.4.4. Aritmética de cifras significativas

La aritmética de cifras significativas (ACS) es similar en algunos aspectos a la de intervalos. La idea es de Nick Metropolis y consiste en ir conservando sólo los dígitos correctos de un cálculo a otro, con una serie de reglas que permiten aproximar el error, haciendo uso de distribuciones de probabilidad. En las computadoras surgió con el propósito de identificar los bits correctos al realizar operaciones numéricas, como apoyo a la aritmética convencional (de números normalizados).

En 1963, Max Goldstein presentó algunas de las primeras experiencias, sobre las máquinas IBM 7090 en [78]. En esas computadoras, el modo de punto flotante estaba separado del de cifras significativas.

El modelo de APF se base en que el error relativo tiene distribución uniforme, es decir, en $\mathrm{fl}(x) = x(1+\delta)$, con $|\delta| < \mu$, debe ocurrir que δ puede ser cualquier número en $(-\mu, \mu)$ con la misma probabilidad.

Según la ACS, la *propagación del error* se puede ir midiendo con reglas que dependen de la naturaleza de las operaciones y su inexactitud particular, conocida por anticipado.

Si dos números x y y tienen errores relativos δ_x y δ_y, entonces el error relativo de la suma $S = x + y$ pudiera calcularse por cuadratura:

$$\delta_S = \sqrt{\delta_x^2 + \delta_y^2}$$

si se supone distribución normal y una contribución pareja al error en la suma. Por ejemplo. $x = 2 \pm .1$ y $y = 3 \pm .2$. Considerando la aritmética convencional, se tiene $x + y = 5 \pm .3$. Pero si se supone una distribución normal del error, se tiene

Suponer cualquier distribución de probabilidad para el error es mucha osadía. Debe sustentarse en algo, incluso para la distribución uniforme

$$\begin{aligned} x + y &= 5 \pm \sqrt{.1^2 + .2^2} \\ &= 5 \pm .2236... \end{aligned}$$

lo que mejora la aproximación. En esta aritmética, comunmente se habla de *incertidumbre* en vez de error. La ACS es mucho más optimista que la aritmética de intervalos.

Lo mejor (más realista) es suponer que cada cantidad contribuye de manera diferenciada al error final, por lo que debe calcularse la contribución por

separado:

$$\delta_{S_x} = \left(\frac{\partial S}{\partial x}\right)\delta_x$$

en el caso continuo, que se discretiza para desarrollar programas de cómputo.

Si se trata del producto xy, el error relativo es

$$\delta = \sqrt{\left(\frac{\delta_x}{x}\right)^2 + \left(\frac{\delta_y}{y}\right)^2}$$

y hay fórmulas para cada operación y función elemental, que acotan mejor el error.

En 1984, Christopher Jones presentó el trabajo [96], en el que calcula el intervalo donde se encuentra el resultado inexacto de cada operación, pero los límites del intervalo se almacenan con precisión limitada por sus cifras significativas. Esto con la idea de reducir el costo de cómputo. Por ejemplo, si el resultado es 9.98328±.0088542, entonces es mejor escribir 9.98±.009. Se acostumbra la notación 9.98(9) para ahorrar espacio.

Otros trabajos sobre lo que llaman también *aritmética de rango*[17], son los de Aberth [1] y [2], aunque los trabajos iniciales datan de hace más tiempo, como el de Gibb de 1961 [73].

Otra alternativa para esta aritmética es usar las fórmulas de propagación de errores junto con simulación de Monte Carlo.

El famoso paquete Mathematica, de Wolfram Technology, utiliza esta dentro de una de sus modalidades de cálculo, que de hecho liga los conceptos de error relativo con el de cifras significativas, para ir midiendo el error y mejorar la exactitud cada operación. En algunas aplicaciones de la física también se utiliza. En ambos casos, debe combinarse con cálculo simbólico para garantizar exactitud y precisión.

En su contra, hay un resultado poco alentador de lo que se puede llamar folklor popular que dice que sin importar cómo se programe la ACS, ocurre una de dos cosas:

1. n operaciones **amplifican** la incertidumbre en $\beta^{\frac{n}{2}}$
2. n operaciones **disminuyen** la incertidumbre en $\beta^{\frac{n}{2}}$.

Por ello tampo resulta una buena opción, en particular por redondear con diferentes precisiones, en vez de una precisión acotada.

2.4.5. Aritmética de redundancia

En este sistema numérico cada dígito puede tener su propio signo. En base β se usan los dígitos $0, 1, 2, \ldots, \beta - 1$. Segun comenta Muller en [140], ya en 1840

[17] También le llaman Crank Three Times.

Cauchy sugirió utilizar los dígitos del -5 al 5 para facilitar las multiplicaciones. En algunos sistemas digitales se usa -1, 0 y 1 en base 2 para facilitar algunos cálculos. Por ejemplo, una ventaja es que en algunos sistemas es posible realizar sumas sin necesidad de acarreos.

La idea básica es

y es así como es referido en algunos textos (como [140]).

Capítulo 3

Norma 754 de IEEE

Se presenta a continuación una revisión rápida de la norma de punto flotante vigente en 2007, comenzando con las condiciones que propiciaron su desarrollo. No tiene sentido transcribir las 59 páginas del documento oficial, pero se extraen y analizan las características que más pueden interesa a los programadores.

3.1. Proceso histórico

Debido a la diferencia de valores en los parámetros de 2.4, así como el criterio de redondeo y operaciones especiales, academia e industria de la computación vieron la necesidad de definir una manera de trabajar efectiva y consistente. Al inicio de los 80s, había más de 20 formatos distintos para NPF en las computadoras existentes. Uno de los primeras referencias donde se comparan diversas APF es [185], aunque el trabajo sobre APF no normalizadas de Ashenhurst y Metropolis [12] es todavía anterior y abogan por el uso de formatos no normalizados, presentando las bondades en cuanto a la propagación de errores y la eficiencia temporal[1]. Posteriormente, en 1964, Ashenhurst presentó en [11] los criterios de ajuste para la evaluación de funciones con parámetros no normalizados. La decisión de la base β para la representación interna fue crítica y muy discutida.

Los códigos portables tenían que ser muy extensos para considerar las diferentes arquitecturas.

La necesidad imperiosa de normar la forma de hacer cálculos numéricos con cumputadoras fue plasmada por Kahan con gran claridad y muchos detalles en "Why do wee need a floating-point arithmetic standar?"[2] en 1981. Los primeros pasos ya los había dado durante su trabajo con la APF de IBM y las calculadoras HP. Por esos años, algunos alegaban que la propuesta era más bien un diseño de software más que una norma.

[1] La rapidez con que se realizan los cálculos.

[2] ¿Por qué necesitamos una norma para aritmética de punto flotante?

Puede resultar de gran interés el proceso de desarrollo de la norma, que comienza en Intel en 1976, en un intento por lograr que cada procesador que dijera Intel por fuera, no calculara disparates por dentro. La propuesta fue formalizada en [110] y sus implicaciones a los lenguajes de alto nivel fueron analizadas por Richard Fateman en [58], principalmente los aspectos referentes a Fortran, donde propone soluciones parciales a algunos problemas.

William Velvel Kahan ganó la medalla Turing por sus trabajos en el desarrollo de la norma de PF, referido como el padre de la APF.

La entrevista a William Kahan[3] revela mucho de la historia detrás de esta norma, como la indiferencia de los fabricantes de grandes computadoras hipnotizados por la velocidad o la interesante lucha de la compañía Dec (véanse [150], que presenta la otra propuesta y en particular [149] para la APF de VAX) en contra del underflow gradual incluido en la primera propuesta K-C-S (Kahan-Coonen-Stone) de 1979. Las otras propuestas de norma fueron la de Payne-Strecker y Fraley-Walther, ambas de 1980, así como la de Brown en [21].

Dos doumentos que ilustran y hacen ver los beneficios y problemas remanentes son el de Kahan [105], de observaciones a la norma de 1985 y el de Goldberg "lo que todo científico de la computación debe saber sobre APF [75], que presenta realmente lo que todo programador debe saber al respecto, basado en un curso que Kahan dio en Sun Microsystems en 1988. Posteriormente, en 2000, Sun agregó un anexo (de Doug Priest) que aclaraba algunas cosas más, disonible en su "Numerical Computation Guide. En 2006 se publicó algo que podría llamarse "lo que todo estudiante debe saber sobre APF" ([6]), de Chuck Allison.

Lectura recomendad para todo estudiante de esta área y afines.

Ante el interés de una norma para otras bases diferentes a 2, debido principalmente a la existencia de computadoras en base 8, 10 y16, surgió la norma 854, cuyo primer borrador inicial se presenta en 1985 en [36]. Se aprobó en 1987. En 1991, se publica la Language Compatible Arithmetic Standard (LCAS, ISO/IEC 10967:1991), propuesta por Mary Payne y Brian Wichmann, que fue duramente criticada por Kahan en [103] donde comenta que "está tan llena de errores que el mundo de la computación debe rehazarla", con argumentos de todo tipo; su lectura vale la pena. Al año siguiente, el grupo de trabajo de software numérico de IFIP[4] publicó otra crítica en [43] en la misma dirección que la opinión de Kahan.

Fue precisamente IFIP quien en 1978 publicó [61] donde se propone una estandarización de nombres y definiciones necesarias para la fácil portabilidad de códigos y algoritmos. El artículo se escribió a partir del cuarto borrador que había sido ampliamente difundido entre programadores y analistas numéricos. Muchas de las sugerencias fueron adoptadas por la norma de IEEE, hayan o no sido desarrolladas independientemente.

Los estándares de IEEE tiene 15 años de vigencia. ¡Se tardaron 7 años en la revisión!

Al momento de escribir esto (junio de 2007), IEEE no ha publicado la revisión de la norma de punto flotante, conocida como IEEE-754R, cuyo último borrador es de octubre de 2006, pero es de esperarse que los detalles técnicos no sufran

[3] `www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`

[4] International Federation for Information Processing, creada en 1960 por UNESCO. El grupo de análisis numérico de IFIP (WG 2.5) se reunió por primera vez en 1975, en Oxford.

modificaciones. La norma, indica la forma como debe implementarse un sistema de punto flotante, por software, hardware o alguna combinación de ambos. De hecho, se especifica que el soporte para punto flotante deben darlo el lenguaje, el compilador y el procesador.

Es importante enfatizar que no se garantizan respuestas correctas, como ya se vio que es imposible, pero sí respuestas consistentes de un sistema a otro. Un error común es pensar que la norma es para que la cumpla el hardware y que los programadores no deben preocuparse. Usar bien la norma en un programa puede significar los detalles con que se construye el mejor software.

Como el proceso de revisión de la norma ha sido completamente público, diseñadores de lenguajes y fabricantes de procesadores y compiladores han ido incluyendo en sus productos algunos de los lineamientos definidos, por lo que en este momento disponemos de sistemas de cómputo que cumplen parcialmete alguna de las características. Ninguno la cumple del todo, aunque hay excelentes esfuerzos, como el de la versión C99 del lenguaje C o Fortran 2003, aprobados por ANSI.

La norma de 1985, comenzó a revisarse en el año 2000, con la experiencia que dejaron más de 15 años de su uso. Los objetivos de la revisión fueron muy claros:

- Mezclar las normas 754 (punto flotante base 2) con la 854 (punto flotante base ≥ 4), emitida en 1986.
- Reducir las opciones de implementación.
- Incluir y unificar formatos de datos externos.
- Resolver las ambiguedades de las normas anteriores.
- Estandarizar la operación `fma` (fused multiply-add), $x + y \times z$ con un sólo redondeo final.
- Incluir la cuádruple precisión (128 bits).
- Estandarizar algunas funciones elementales ($\sqrt{}$ ya se había incluido en 1985).

`fma` ya se usaba desde antes, como se muestra en la página 65

Lo anterior, evitando invalidar la operación de ninguna computadora que respetara razonablemente la norma de 1985 y que siguiera en uso. Todos los detalles pueden consultarse directamente en el documento [39][5], cuyo borrador final está disponible en internet[6], pero se presentan a continuación los aspectos más destacados y de importancia para quien desarrolla programas donde es necesario cuidar la precisión y la confiabilidad numérica.

[5] Tambien conocido como IEC 60559:1989 para la versión de 1985.
[6] www.validlab.com/754R/, grouper.ieee.org/groups/754/

ISO define una serie de estándares adicionales para aritmética de computadora en el documento ISO/IEC-10967 (IEC 60559), de 2005 que en general es compatible con el de IEEE. Vale la pena su revisión por quien quiera adentrarse aún más a los detalles finos de la especificación de requisitos numéricos. A diferencia de IEEE754, no norma detalles como la representación numérica, bandera de inexacto, algunos modos de redondeo o NaNs, dejando más libertades (¿o riesgos?) que IEEE.

3.2. Contenido de la norma (y observaciones)

La norma tiene todos los elementos de una típica norma de IEEE, distinguiendo aquello que no pertenece a la norma sino que aparece para aclarar o informar.

3.2.1. Propósitos

En general, pone orden en el desempeño del cómputo de punto flotante (binario y decimal), para obtener resultados que sean independientes de si se procesan en hardware, software o una combinación. El usuario debe tener control sobre todos los parámetros para:

1. Facilitar la portabilidad de programas.
2. Permitir que usuarios no expertos desarrollen programas seguros y sofisticados.
3. Permitir que usuarios expertos desarrollen programas robustos, eficientes y portables.
4. Dar facilidades para el diagnóstico de problemas en tiempo de ejecución, manejo de excepciones sencillo y aritmética de intervalos adecuada.
5. Propiciar la programación de funciones elementales, APF de precisión múltiple y fácil cómputo simbólico.

Se agregan anexos que determinan posibles caminos a futuro, así como recomendaciones generales y sugerencias para fabricantes y desarrolladores.

La norma *especifica*

- Formatos para datos en punto flotante binario y decimal, para cómputo y almacenamiento.
- Conversiones entre enteros y formatos de punto flotante.
- Conversiones entre los diversos formatos.

- Excepciones y su manejo, incluyendo datos que no son números.
- Lo que es obligatorio, lo que se espera, lo que se permite y lo que se recomienda.

Aunque se especifican las características mínimas de las conversiones, la norma no dice cómo deben hacerse, por lo que la decisión queda a cada fabricante de compiladores.

La norma *No especifica*

- Formatos internos o externos para números enteros o secuencias de caracteres.
- Cómo convertir entre bases.
- Interpretación del signo o mantisa de NaN.
- Cuántos bits extras se deben usar en los cálculos intermedios.
- Qué parte del ambiente de programación debe proporcionar cada característica.

3.2.2. Formatos definidos

Los números de punto flotante se dividen en dos grupos, para cómputo o para almacenamiento. En ambos casos se dispone de representación en base 2 y 10. Para el formato interno, hay precisión simple, doble y cuádruple.

	Base 2	**Base 10**
Cómputo	32, 64 y 128	64 y 128
Almacenamiento	16	32

Todos los formatos se especifican con 4 parámetros básicos: base, precisión, exponente máximo y exponente mínimo. Los valores correspondientes a los formatos de cómputo ee presentan en la tabla 3.1. Para los valores de almacenamiento, referirse a [39].

Con estos parámetros, se representan números de la forma

$$x = (-1)^s \times m \times \beta^e \tag{3.1}$$

donde los campos corresponden a los explicados en (2.4), página 29. m es un número representado con una cadena de dígitos de la forma $0.d_1d_2d_3...d_p$, con $0 \leq d_i < \beta$, por lo que $0 \leq m < 1$.

Cualquier formato debe poder representar las cantidades $-\infty$, ∞, una señal NaN, un NaN silencioso.

Base	2			10	
Tamaño	32	64	128	64	128
p	24	53	113	16	34
$e_{\text{máx}}$	127	1023	16383	384	6144
$e_{\text{mín}}$	-126	-1022	-16382	-383	-6143

Tabla 3.1: Parámetros binarios.

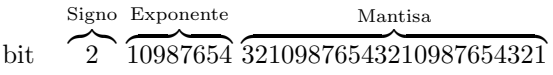


Figura 3.1: Formato binario en precisión simple. Nótese cómo los bits se cuentan de derecha a izquierda.

Formatos base 2

En el caso de los números de *precisión simple*, el formato es de 32 bits, reservando el primero para el signo (s), 8 para exponente (e) y 23 para mantisa (m). En cada campo, los bits más significativos están a la izquierda. Para la computadora, el formato es el indicado en la figura 3.1, donde los números indican la significancia de los bits. El valor x en (3.1) se calcula según las definiciones de la tabla 3.2.

En el caso de los números normales, aparece el bit implícito y los subnormales permiten el underflow gradual.

El número 1.0 queda representado por 0 01111111 00000000000000000000000. En este caso, $m = 0$ y $e = 127 - 127 = 0$, por lo que $(-1)^0 \times 2^0 \times 1.0 = 1.0$.

El número 2.0, es 0 10000000 00000000000000000000000. $m = 0$ y $e = 128 - 127 = 1$, por lo que $(-1)^0 \times 2^1 \times 1.0 = 2.0$.

Otro ejemplo: ¿Qué número representa 0 10000000 10010010000111111011010? $m = .570\,796\,370\,506\,286\,621\,093\,75$ y $e = 1$, por lo que $(-1)^0 \times 2^1 \times 1.570\,796\,370\,506\,286\,621\,093\,75 = 3.141\,592\,741\,012\,573\,242\,187\,5$. Una aproximacióna al valor exacto π. Obsérvese que esta no es la mejor aproximación posible (falla en el séptimo decimal), pero es el formato de precisión simple.

e	m	Valor de x	
255	$\neq 0$	NaN	
255	$= 0$	$(-1)^s \times \infty$	
$0 < e < 255$	cualquiera	$(-1)^s \times 2^{e-127} \times (1.m)$	(normales)
0	$\neq 0$	$(-1)^s \times 2^{e-126} \times (0.m)$	(subnormales)
0	$= 0$	$(-1)^s \times 0$	

Tabla 3.2: Parámetros del formato binario en precisión simple.

e	m	Valor de x
2047	$\neq 0$	NaN
2047	$= 0$	$(-1)^s \times \infty$
$0 < e < 2047$	cualquiera	$(-1)^s \times 2^{e-1023} \times (1.m)$
0	$\neq 0$	$(-1)^s \times 2^{e-1024} \times (0.m)$
0	$= 0$	$(-1)^s \times 0$

Tabla 3.3: Parámetros del formato binario en precisión doble.

e	m	Valor de x
16383	$\neq 0$	NaN
16383	$= 0$	$(-1)^s \times \infty$
$0 < e < 16383$	cualquiera	$(-1)^s \times 2^{e-8191} \times (1.m)$
0	$\neq 0$	$(-1)^s \times 2^{e-8190} \times (0.m)$
0	$= 0$	$(-1)^s \times 0$

Tabla 3.4: Parámetros del formato binario en precisión cuádruple.

Otras cantidades fáciles de representar son

0 0000001 00000000000000000000000	Mínimo número normal
0 0000000 11111111111111111111111	Máximo subnormal
0 0000000 00000000000000000000001	Mínimo subnormal.

En general, todos los formatos en base dos tiene la misma forma de representación, excepto que los valores de los parámetros (tabla 3.1) son distintos. Para precisión **doble** y **cuádruple** son las tablas 3.3 y 3.4 respectivamente.

Se pueden calcular los valores mínimos y máximos para los formatos, tanto normales como subnormales y se presentan de manera aproximada en la tabla 3.5. No todos los compiladores o procesadores disponen del tipo de dato para precisión cuádruple. De hecho es más común que se disponga del tipo de dato de precisión extendida (80 bits). Pero sí existen varios disponibles en forma gratuita en internet con los que se puede trabajar (Ch, Dev-C++ y otros).

Precisión	**Simple**	**Doble**	**Cuádruple**
Mayor normal	3.402×10^{38}	1.797×10^{308}	1.189×10^{4932}
Menor normal	1.175×10^{-38}	2.225×10^{-308}	3.362×10^{-4932}
Menor subnormal	1.401×10^{-45}	4.94×10^{-324}	6.475×10^{-4966}
ε_M	1.192×10^{-7}	2.22×10^{-16}	1.926×10^{-34}
μ	5.96×10^{-8}	1.11×10^{-16}	9.629×10^{-35}

Tabla 3.5: Valores límites aproximados de $\mathbb{F}$.

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Tabla 3.6: Conversión directa de base 10 a binario con bcd.

Formatos base 10

Pese a que la base 2 tiene ventajas sobre las otras, en muchas aplicaciones donde el ser humano está en contacto constante con los números (como en los sistemas financieros) no resulta tan conveniente. Ya en 1986 la norma 854 describía las formas de operación para bases distintas a la 2, incluyendo el 10. Ante la ventaja de la base 10 para cantidades donde la interacción humana es constante[7], durante la revisión de la norma se decidió incluir tanto la base 2 como la base 10 como las únicas aceptables.

Como los números con que el usuario trabaja tienen representación exacta en base 10, es de esperar se que al hacer operaciones sin salirse de esta base se evite el problema de convertir con inexactitud a base dos, hacer los cálculos y reconvertir con otra inexactitud a base 10. Uno de los mayores promotores y desarrolladores de esta base es Mike Cowlishaw, de IBM, quien ha colaborado intensamente en el desarrollo del formato y la aritmética decimal, en particular la Especificación Aritmética Decimal de IBM (ver [45], aunque la versión mas reciente es de 2007). Para mayor ilustración del trabajo decimal, revisar las librerías decNumber para ANSI C y BigDecimal class de Java.

Como es de esperarse por 2.6 en la página 33, el wobbling pesa más que en base 2, pero este inconveniente es compensado por la exactitud de la representación de las cantidades de interés, como las monetarias. No es recomendable en cálculos meramente científicos, por lo que no se presenta con todo el detalle de la representación en base 2, sino sólo las generalidades.

Utilizada por múchos compiladores.

Desde hace muchas décadas, se ha utilizado la representación binaria para números decimales y se desarrollaron gran cantidad de variantes, que cumplían con alguna necesidad o restricción. La más simple es donde cada dígito decimal es representado por su conversión natural en binario:

De esta manera, el 16743 puede representarse como

$$\text{Representación: } \overbrace{0001}^{1}\overbrace{0110}^{6}\overbrace{0111}^{7}\overbrace{0100}^{4}\overbrace{0101}^{3}.$$

Esta codificación se conoce como *decimal codificado en binario* (bcd, por sus siglas en inglés). Como no es la única forma de conversión, esta es referida como bcd8421. Algunas ventajas son:

O casi directo.

1. Convertir la cadena numérica de entrada al formato interno es directo.
2. El redondeo es trivial.
3. Escalar por potencias de 10 es muy sencillo.
4. Alinear cantidades con el punto decimal es fácil.

[7] Las calculadoras de mano son un buen ejemplo.

Algunos compiladores disponen de este tipo de datos para variables tanto enteras como flotantes. Si se utiliza este método con notación científica normalizada, entonces hay que guardar el signo, la mantisa y el exponente de la base 10. Como el exponente es un entero, se usa la notación complemento a 2 convencional, que es el método más ampliamente utilizado para representar cantidades enteras en las computadoras modernas, por sus cómodas propiedades aritméticas.

Notación complemento a 2 y bcd

Con n bits, existen 2^n configuraciones. Los positivos y el cero se identifican porque el bit más significativo es 0. Los negativos se calculan restando a 2^n el valor absoluto del número.

Por ejemplo, si se utilizan 8 bits, el 54 es 00110110_2. El -54 se calcula inviertiendo los bits y se suma 1:

$$\begin{array}{rl} 54 = 00110110_2 & \\ 11001001_2 & \text{Se invierte} \\ -54 = 11001010_2 & \text{Se suma 1.} \end{array}$$

Para la conversión contraria, se toma el $-54 = 11001010_2$, se le resta 1 y se invierten los bits.

El problema de la notación en bcd es que las representaciones del 9 al 15 no se utilizan, es decir, las que faltan en la tabla 3.6.

Dígito	9	10	11	12	13	14	15
Binario	1001	1010	1011	1100	1101	1110	1111

Si se utiliza bcd para codificar los 1000 números del 0 al 999, se usarían en total 12 bits para cada número. La observación de que con sólo 10 bits hay $2^{10} = 1024$ representaciones deja ver que es posible ahorrarse 2 bits cada 3 dígitos decimales[8].

Del hecho anterior surge la idea de codificar el formato decimal de una manera compacta. La primera versión data de 1971, por Tien Chi-Chen e Irving T. Ho, [178], publicada con varios refinamientos. La versión utilizada en IEEE754 es llamada *Densely Packed Decimal* (DPD) [44], que tiene la ventaja de utilizar de manera óptima no solo 10 bits para 3 dígitos decimales, sino 7 bits para dos dígitos y 4 bits para 1 dígito, este último codificado como el bcd convencional.

Así, la norma define el *declet* como un grupo de 10 bits que representan 3 dígitos decimales. La mantisa del formato flotante en base 10 se forma por varios declets. La representación es óptima pues si se trata de 5 dígitos decimales, se usan dos bytes: 10 bits para 3 y luego 7 más para los 2 restantes. El bit restante se usa en un campo que se combina con el del exponente.

Y se le llama campo de combinación.

La codificación trabaja de la siguiente manera. Si los tres dígitos decimales en código bcd son los 12 bits `B=(abcd)(efgh)(ijkm)`, entonces, cada bit de la representacin dpd se construye a partir de la tabla 3.7.

De esta manera se forma `D=(pqr)(stu)(v)(wxy)`. También aquí el último bit de cada representación se mantiene constante. Quedan aún 24 de las posibles 1024 combinaciones para futuras extensiones. Para decodificar, se parte de `D` y se obtiene `B` utilizando la tabla 3.8.

La norma no especifica cómo se deben usar.

[8] Y sobran 24 representaciones.

aei	pqr	stu	v	wxy
000	bcd	fgh	0	jkm
001	bcd	fgh	1	00m
010	bcd	jkh	1	01m
100	jkd	fgh	1	10m
110	jkd	00h	1	11m
101	fgd	01h	1	11m
011	bcd	10h	1	11m
111	00d	11h	1	11m

Tabla 3.7: Conversión de decimal a dpd.

vwxst	abcd	efgh	ijkm
0....	0pqr	0stu	0wxy
100..	0pqr	0stu	100y
101..	0pqr	100u	0sty
110..	100r	0stu	0pqy
11100	100r	100u	0pqy
11101	100r	0pqu	100y
11110	0pqr	100u	100y
11111	100r	100u	100y

Tabla 3.8: Conversión de dpd a decimal.

Por ejemplo, para convertir el número `B`=437. En bcd se representa con

$$\texttt{B} = \overset{4}{\texttt{0100}}|\overset{3}{\texttt{0011}}|\overset{7}{\texttt{0111}}$$

Considerando la tabla de codificación, se ve que `aei`=000, por lo que corresponde al primer renglón. Entonces, `D` se forma como `100 011 0 111`. Para decodificarlo, como `v`=0 en la tabla de decodificación, corresponde al primer renglón de nuevo, sin importar el valor de `wxst`. Y de nuevo se obtiene `B`.

Restando aún más ortogonalidad a la norma.

La representación fue uno de tantos motivos de debate intenso y el resultado fue que la norma incluye tanto representación en bcd como en dpd, como se explica en el apartado 5.5.c incisos 1 y 2 de la norma.

En general la representación de un número de punto flotante tiene la forma

$$\text{bit} \quad \overbrace{1 \text{ bit}}^{\text{Signo}} \overbrace{w+5 \text{ bits}}^{\text{Exponente}} \overbrace{T = J \times 10 \text{ bits}, \; J \text{ declets}}^{\text{Mantisa}}$$

El campo del exponente, G, es de longitud variable, llamado *campo de combinación*. Mide al menos 5 bits, para los valores especiales de 2.4:

G	Significado
11111	NaN
11110	$\pm\infty$
00000	
10000	± 0
01000	

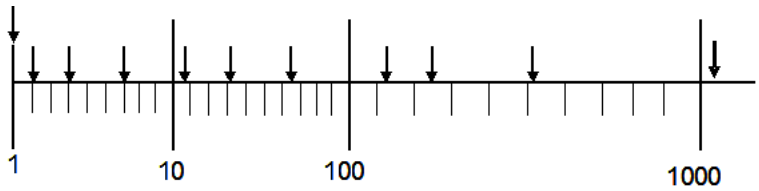


Figura 3.2: Potencias de 2 y 10. Las flechas muestran la posición aproximada de las potencias de 2.

Si G tiene otra combinación de bits, entonces se trata de un número finito con declets válidos como los de las tablas 3.7 y 3.8. Como es natural, la mantisa se forma con la concatenación de los bits de los declets y posiblemente alguno del campo de combinación. Aún en números finitos, los primeros bits de $G = g_1 g_2 ... g_{w+5}$ determinan la forma de codificación.

El lector interesado por el resto de los detalles de la codificación base 10 debe remitirse a [39] o a la patente 6437715 de Cowlishaw, en USA, donde se dan recomendaciones de implementación.

Es recomendable utilizar códigos probados en vez de capturar las tablas 3.7 y 3.8.

No es difícil determinar los valores mínimos de la precisión binaria y decimal para garantizar que la conversión decimal$\longleftrightarrow$binario no tenga problemas. Supóngase (como hasta ahora) que se tiene p dígitos binarios de precisión y P dígitos decimales. Sea el número real $x \in [2^b, 2^{b+1}]$ y $x \in [10^D, 10^{D+1}]$. Úsese de referencia la figura 3.2.

El error de redondeo máximo del formato binario es 2^{b+1-p} y el del formato decimal es 10^{D+1-P}, es decir, la distancia de x con los NPF vecinos más cercanos. Las conversiones de binario a decimal y decimal a binario tendrán un error de redondeo máximo de $E_D = 5 \times 10^{D-P}$ y $E_b = 2^{b-p}$ respectivamente. La conversión de $\mathrm{fl}(x)$ de binario a decimal no tendrá pérdidas (mayores a E_D) si la suma de los errores es menor que la distancia entre los NPF del formato binario, es decir,

$$E_D + E_b < 2^{b+1-p}.$$

Esto se cumple cuando $P > D + 1 - (b + 1 - p) \log 2$.

Como se supone que $10^D \leq 2^{b+1}$, entonces $D \leq (b+1) \log 2$, por lo que se obtiene

$$P > 1 + p \log 2.$$

Si $p = 53$ como en precisión doble, entonces con $P = 17$ es suficiente. El proceso en el otro sentido, es decir, la conversión de $\mathrm{fl}(x)$ de decimal a binario, requiere que $E_D + E_b < 10^{D+1-P}$ y se obtiene que para garantizar 53 dígitos binarios correctos se requieren 15 dígitos decimales como máximo. Dígitos decimales extra ya no estarán correctamente redondeados ($E_b \leq 2^{b-p}$) con 53 bits.

Estos cálculos y otros similares determinan los parámetros de los diversos formatos para cunplir con la demanda exigida.

3.2.3. Modos de redondeo

El modo de redondeo es un parámetro implícito en la norma. El ambiente de programación debe permitir modificar el modo en tiempo de ejecución, de compilación o

		Redondeo empírico	TiesToEven
	2.005	2.01	2
	2.015	2.02	2.02
	2.025	2.03	2.02
	2.035	2.04	2.03
Suma:	8.08	8.1	8.07
Error	absoluto	0.02	0.01
	relativo	0.00247525	0.00123762

Tabla 3.9: Diferencia entre el redondeo empírico con el TiesToEven. Se muestra el caso de la suma, con los errores absoluto y relativo.

de depuración, preferentemente los tres, así como el manejo de excepciones correspondiente[9]. Este parámetro afecta a todas las operaciones inexactas.

Según la norma, todas las operaciones deben realizarse como si calculara un resultado correcto hasta el infinito y sin cotas, para después ajustarlo a su formato final mediante el modo de redondeo, que puede ser de dos tipos. Supóngase que se tiene un resultado infinitamente exacto x y que los NPF más cercanos son $\underline{x}$ y $\overline{x}$, de manera que $\underline{x} \leqslant x \leqslant \overline{x}$.

Como en aritmética de intervalos.

- Redondeo al más cercano:
  - TiesToEven. x se convierte en el NPF más cercano. Si están a la misma distancia, se usa aquel cuyo último bit sea 0.
  - TiesToAway. x se convierte en el NPF más cercano. Si están a la misma distancia, se usa aquel con mayor magnitud.
- Redondeo direccionado:
  - TowardPositive. x se convierte en $\overline{x}$.
  - TowardNegative. x se convierte en $\underline{x}$.
  - TowardZero. x se convierte en el NPF más cercano no mayor en magnitud.

El modo de redondeo default es siempre el redondeo al número más cercano. Si los dos más cercanos están igualmente cerca, se preferirá aquel que tenga su bit menos significativo igual a cero. La justificación es más clara en base 10. Si se va a redondear a dos decimales los números 2.005, 2.015, 2.025, 2.035, entonces redondear empíricamente sumando .005 sesga las cantidades hacia arriba. La mejor decisión es restar cuando el dígito decimal anterior es par (como en 2.005 y 2.025) y sumar cuando sea impar (como en 2.015 y 2.035). En esto consiste el redondeo TiesToEven, reduciendo el sesgo, como lo muestra la tabla 3.9.

La idea data de 1894, de Jules Tannery.

La utilidad mayor de los modos de redondeo direccionados, a parte de la obvia aplicación en aritmética de intervalos, tiene fundamento en que normalmente los errores de cálculo de las computadoras se deben a unos cuantos errores críticos de redondeo más que a legiones de errores numéricos de todo tipo distribuidos uniformemente en todo

[9] Esto es particularmente importante en cuando se trata de aritmética de intervalos.

el código. (Debe suponerse esta regla con sus debidas excepciones.) El caso evidente son los cálculos alrededor de singularidades, como $\frac{1}{1-x}$ cerca de $x = 1$.

Por ello, aislar un bloque de código sospechoso y ejecutarlo con modos de redondeo distinto puede llevar a un diagnóstico más eficiente del problema. Al mismo tiempo, se debe estar atento a la reorganización del código por parte de la etapa de optimización del compilador, para evitar cambios de semántica numérica.

Diseñar pruebas es normalmente más difícil que diseñar bien las rutinas que se quieren probar.

Las primeras experiencias con lo que en su momento era la propuesta de norma, hizo que los fabricantes desarrollaran esquemas esquemas eficientes para realizar el redondeo. El primer bit ya fue presentado en la página 39 al ver las propiedades de los NPF $\mathbb{F}$: el bit de guardia. En la representación, este se coloca al lado del bit menos significativo de la mantisa, como es de esperarse.

Estudios estadísticos pioneros son [119][10], donde Kuki y Cody reaalizaron un análisis estadístico del desempeño de los bits de guardia con bases 2 y 16 y modos de redondeo; y [113], donde Kanedo y Liu calculan el error relativo de la suma utilizando dígitos de guardia. En ambos se revelando su gran importancia.

El bit de guardia asegura que la resta no incurra en errores relativos altos, pero no es suficiente para asegurar un redondeo que garantice un error máximo de .5 ulp. Para esto, se agrega un bit más, que afina el redondeo y es conocido como *bit de redondeo*. Este se coloca al lado del bit de guardia, en la parte menos significativa.

En la práctica, cada fabricante decide qué hacer. Con el formato de doble precisión se tiene una mantisa de 52 bits y uno implícito (el ufp). En vez de usar 52+2 bits, Intel dice usar 80+2 bits en sus registros flotantes, aunque 64 son los utilizados por la mantisa, así que trabaja con 64+2 para redondeo. Por su parte, algunos procesadores de IBM, como el del servido z990, usan 56+4 bits. Sparc sí usa los 52+2 que la norma indica.

La diferencia de bits de redondeo daría problemas de portabilidad, de no ser por un mecanismo adicional que termina por asegurar que todos redondeen como si redondearan a partir del resultado infinitamente correcto.

Para indicar que durante la normalización se han perdido dígitos al salirse de la mantisa en las operaciones intermedias, se agrega un tercer bit llamado *sticky bit*[11]. Una vez que el sticky se prende, ya no puede apagarse pues es un indicador de que ha ocurrido una pérdida. Claro, esto es mientras se están haciendo cálculos en los registros, pues una vez regresado el valor a la memoria, sus 52 bits redondeados es lo único que prevalece.

Queda “pegado”.

Con este tercer bit ya no se necesita tener una precisión intermedia de una gran cantidad de bits. Uno solo sirve para representar toda mantisa descartada durante la normalización. El bit queda “pegado”.

Por ejemplo, supoóngase que se tiene un número en precisión simple cuyo formato indicando el bit implícito y los tres bits adicionales es

$$0\ 0000001\ (1)10000000000000000000010\ 000$$

El campo del exponente es 1, por lo que al dividir entre dos exponente se hace 0, usando la representación de los subnormales. Al ir dividiendo entre 2 sucesivamente

[10] Antes había sólo modelos probabilísticos que modelaban la propagación de errores.

[11] Alejandro Casares le llama bit pegajoso, en la traducción [147] del libro de Michael Overton [146]

se obtiene la secuencia

Valor inicial	0 0000001 (1)11000000000000000000010 000
/2	0 0000000 (0)11100000000000000000001 000
/2	0 0000000 (0)01110000000000000000000 100
/2	0 0000000 (0)00111000000000000000000 010
/2	0 0000000 (0)00011100000000000000000 00**1**
/2	0 0000000 (0)00001110000000000000000 00**1**
×2	0 0000000 (0)00011100000000000000000 00**1**
⋮	⋮

y el sticky (en negrita) no debe apagarse pues es la señal de que se perdieron bits.

En el caso del redondeo al más cercano, se toman las siguientes decisiones:

Si el dígito de redondeo es ...	**entonces se resuelve ...**
$< \beta/2$	truncando
$> \beta/2$	sumando 1 ulp (redondeo hacia arriba)
$= \beta/2$	redondeando al más cercano

En algunas fuentes no se diferencia entre estos 3 bits y simplemente se les llama bits de guardia, aunque esto pueda llegar a confundir con el primero de estos. Estos tres bits, junto con otros se colocan en los *registros de estado y control*, del procesador. También se incluyen bits para indicar el modo de redondeo empleado y las banderas de excepción.

La bolsa de valores de Vancouver

Desde enero de 1982, la bolsa de valores de Vancouver, Canadá, realizaba alrededor de 3000 transacciones diarias, operando con cuatro decimales y truncando a tres en cada ocasión. Esto resultaban en la pérdida de unos 20 puntos diarios en el índice de precios. En noviembre 25 de 1983, el índice de precios estaba en 524.811 puntos pero el mercado lucía muy bien, por lo que decidieron revisar. Después de tres semanas de trabajo los consultores pudieron recalcular y corregir los 22 meses de errores de redondeo, comenzando el siguiente lunes en 1098.892 puntos, 574.081 sobre el valor anterior.

Revisaron en busca de un error casi dos años después.

3.2.4. Operaciones

La norma define más de 100 operaciones, que pueden ser clasificadas por tipo y excepción que generan o por la relación de los operandos y el resultado. Como ya se comentó, la idea fundamental es que cada cálculo debe realizarse suponiendo un resultado intermedio con precisión infinita y sin cotas, para luego redondear al formato destino.

Para los fanáticos del C++, Java y otros lenguajes, la sobrecarga de operadores imposibilita similares que el tipo de dato esperado influencie la selección del operador, dependiendo sólo de los operandos inmediatos, que es contrario al requisito de la norma. Variables atómicas debieran considerarse como intervalos si aparecen en expresiones mezcladas con intervalos, pero no ocurre así. Por ello se prefiere C, tal como lo diseñaron Kernigham y Ritchie.

En general los orientados a objetos.

Los interesados en los detalles de todas las operaciones deberán referirse a [39]. Se presentan a continuación sólo algunas operaciones relevantes y útiles para el resto de este texto.

nextUp, nextDown y nextAfter

Estas operaciones permiten ir al siguiente NPF en la dirección indicada. `nextUp`(x) es el menor NPF mayor que x. `NextAfter` represa el siguiente NPF de x en la dirección de y y puede implementarse fácilmente a partir de las otras dos funciones.

`nextDown`(x)=−`nextUp`$(-x)$. Si x es el número de menor magnitud en su precisión y negativo, entonces `nextUp`(x)= -0. nextUp(± 0) es el número positivo de menor magnitud.

La propiedad reflexiva sólo se cumple para $\pm\infty$ pues

`nextUp`$(\infty)=\infty$
`nextDown`$(-\infty)=-\infty$.

Lo mismo se cumple si el argumento es NaN.

Conversión de formato externo a interno y viceversa

La norma 754 define conversiones entre todos las combinaciones de formatos, incluyendo entre decimales y binarios (excepto con formatos externos) aunque sin decir cómo deben realizarse. Tampoco especifica la conversión de cadenas de caracteres a formatos internos, como cuando el usuario captura números flotantes y la computadora realiza la conversión.

Niguna empresa está obligada a revelar su tecnología

Hay dos motivos por los que un número real no puede ser representado. El principal se ilustra con el número decimal 0.1. Aún teniendo una representación decimal finita, su representación binaria es infinita.

$$\begin{aligned} 10^{-1} &= (0.00011001100110011.....)_2 \\ &= (0.\widehat{00011})_2 \end{aligned}$$

Esto significa que se encuentra exactamente entre dos números de punto flotante y puede ser representado por cualquiera de estos con la misma precisión[12].

El otro motivo por el que la representación no es exacta es el siguiente. Supóngase que se trabaja con formato de simple precisión. Al sumar $2^0 + 2^{-23}$, el 2^0 ocupa el bit implícito mientras que el 2^{-23} ocupa el bit menos sinificativo de la mantisa de 23 bits. En notación binaria tenemos

$$\begin{aligned} b &= 2^0 + 2^{-23} \\ &= (1.00000000000000000000001)_2 \end{aligned}$$

que utiliza toda la mantisa del formato de simple precisión. Al realizar la operación

$$\begin{aligned} c &= 2^0 + 2^{-24} \\ &= 1.00000000000000000000|1 \end{aligned}$$

[12]Está en el intervalo [0.099999999999999984, 0.10000000000000002].

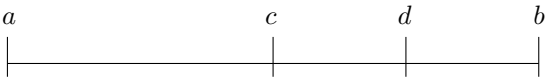


Figura 3.3: Redondeo al más cercano (TiesToEven).

deja el último bit fuera de la mantisa y el resultado debe redondearse a $a = 1$. Esto es debido a que como $2^0 + 2^{-24}$ queda exactamente entre a y b, el redondeo es equivalente al truncamiento (roundTiesToEven) y a es el NPF cuyo último bit es cero. Pero cuando se suma

$$\begin{aligned} d &= 2^0 + 2^{-24} + 2^{-25} \\ &= 1.00000000000000000000000|11 \end{aligned}$$

el resultado d se redondea a b pues $c < d < b$. Esto se ilustra en la figura 3.3. donde

$$\begin{aligned} a &= 1.00000000000000000000000 \\ c &= 1.00000000000000000000000|1 \\ d &= 1.00000000000000000000000|11 \\ b &= 1.00000000000000000000001 \end{aligned} \tag{3.2}$$

Considerando arquitectura Intel, sumarle algo tan pequeño como 2^{-63} a c resulta en el redondeo a b pues $c + 2^{-63}$ es mayor que c. Una potencia más pequeña aún $2^q < 2^{-63}$, es decir, $q < -63$, ya no logra ese redondeo pues la mantisa de los registros de punto flotante de Intel es de 64 bits, por lo que $\text{fl}(c + 2^q) = \text{fl}(c)$. Y de la misma forma, si la suma queda entre a y c, el redondeo dará a.

El sticky bit solo resolverá este problema si las operaciones están suficientemente cercanas en el código como para mantener todos los valores en registros flotantes, disponiendo de bits adicionales. Si la variable se copia de la memoria a un registro, los tres bits adicionales son cero.

Aunque estas cantidades tengan representación exacta en el formato de IEEE, al imprimirlas en pantalla deben convertirse a base 10 y su representación no resulta el mismo número que en base 2 en todos los casos. Por ejemplo, $1 + 2^{-7} = 1.0078125$ ocupa toda la mantisa de la simple precisión, pero $1 + 2^{-8} = 1.0039062|5$ requiere una posición decimal adicional[13].

Mientras el valor de una variable de punto flotante exista en algún registro del procesador, disfrutará de todos los bits disponibles. Pero cuando el registro se necesita para otra cosa el valor de la variable se copia a memoria con la precisión que indica el tipo de dato.

Una forma sencilla de verificar la diferencia con el formato intermedio es usando el código 3.1.

[13]Recuérdese que para la simple precisión de IEEE, la precisión de máquina $\mu = 2^{-24} \approx 5.96 \times 10^{-8}$.

Listing 3.1: Comprobando si hay diferencia con los valores intermedios

```
1 volatile double x = 3./7.;
2 if ( x != 3./7. )
3    puts(''Formato intermedio mayor'');
```

En algunos casos, funciones como `printf` leen directamente del registro flotante del procesador, por lo que aún cuando el tipo sea de simple precisión, la conversión a base 10 utilizará los 64 bits de mantisa del registro y no los 24 del formato. De esa manera, $1 + 2^{-23}$ aparecerá como 1.0000011920928955|078125, quedando sólo 7.8125×10^{-19} fuera de la conversión.

El problema no es sencillo. Considérese ahora el formato de doble precisión. La suma $s = 1+2^{-1}+2^{-24}+2^{-52}$ tiene representación exacta en $\mathbb{F}$, pero su representación en base 10 es

$$1.50000005960464499743522992503130808472633361816406 25$$

por lo que no es posible realizar una conversión a base 10 correcta a partir de la representación binaria exacta, así que se redondea. s es un número que no tiene conversión de binario a decimal exacta.

Similarmente, la suma $S = 1 + 2^{-1} + 2^{-24} + 2^{-53}$ no cabe en el formato de doble precisión de IEEE, por lo que la representación es inexacta y por supuesto la conversión a decimal:

$$1.50000005960464488641292746251565404236316680908203125$$

tampoco es posible. No ha sido difícil encontrar estos ejemplos adversos, como puede apreciarse.

Mucho se ha estudiado al respecto sobre cómo hacer la conversión correctamente tanto en un sentido como en el otro desde los inicios de la computación. Por ejemplo en 1962, Lynch en [129] y Lake en [121], o Goldberg en 1967 [77], que observó de que con m dígitos binarios y n dígitos decimales la conversión en un sentido y otro es exacta (con redondeo) si $2^{m-1} > 10^n$. En [132], 1968, David Matula analiza las condiciones del formato interno para la conversión exacta (cuando es posible) o al menos la conversión redondeada con 1 ulp correctamente redondeado.

Estos y otros trabajos fueron la base para que Clinger [31], Steele y White [82] diseñaran algoritmos que logran la conversión exacta, salvo redondeos, utilizando la precisión recomendada por la norma IEEE754 de 1985. También pueden revisarse [26] y [70].

fma

Esta operación *fused multiply-add*, es novedad en la norma. Tras décadas de cómputo numérico, los expertos se dieron cuenta de lo frecuente que es la operación

$$a + b \times c$$

en casi todo tipo de cálculos: al evaluar polinomios, operaciones con vectores y matrices, estadística, etcétera. Utilizada inicialmente en DSP (procesadores digitales de

señales por sus siglas en inglés), en 1999 fue agregada como rutina estándar en el lenguaje C99 por ANSI y considerada en la revisión de la norma flotante de 1985[14].

Este fenómenos era conocido desde los inicios de la APF.

En la APF convencional, $a+b\times c$ requiere dos redondeos, por lo que el programador obtiene $\mathrm{fl}(a+\mathrm{fl}(b\times c))$ en vez de $\mathrm{fl}(a+b\times c)$. El riesgo consiste en que al hacer *doble redondeo* se puede llegar al resultado incorrecto. Esto ocurre cuando se implementan mal algunas operaciones básicas, sin respetar la norma. Es el caso cuando un cálculo se realiza primero con 80+2 bits[15] y se redondea en un registro flotante de 80 bits, para después ajustar al formato destino de 32 bits redondeando por segunda ocasión.

En general se tiene que

$$\begin{aligned}
\mathrm{fl}(a+\mathrm{fl}(b\times c)) &= \mathrm{fl}(a+(b\times c)(1+\delta_1)) \\
&= \mathrm{fl}(a+b\times c+\delta_1(b\times c))) \\
&= (a+b\times c+\delta_1(b\times c))(1+\delta_a) \\
&= a+b\times c+\delta_1(b\times c)+\delta_a(a+b\times c+\delta_1(b\times c)) \\
&= a+b\times c+\delta_a a+\delta_1(b\times c)+\delta_a(b\times c)+\delta_a\delta_1(b\times c)
\end{aligned}$$

y si $\delta=\max\{|\delta_1|,|\delta_a|\}$, ocurre que

$$\begin{aligned}
\mathrm{fl}(a+\mathrm{fl}(b\times c)) &\leq a+b\times c+\delta a+2\delta(b\times c) \\
&< a+b\times c+2\delta(a+b\times c) \\
&= (a+b\times c)(1+2\delta)
\end{aligned}$$

suponiendo que $\delta_a\delta_1\approx 0$. Esto es, en el peor de los casos, el doble del error, debido a las dos operaciones independientes de suma y multiplicación. La ventaja de `fma` es que se obtiene

$$\mathrm{fl}(a+b\times c)=(a+b\times c)(1+\delta)$$

con $|\delta|<\mu$ porque sólo se realiza un redondeo.

Para comprender el peligro del doble redondeo, supóngase el caso hipotético de $\beta=2$ y $p=3$ bits de precisión: si el resultado exacto de una operación es $x=.100101$, el redondeo al más cercano es .101 por los motivos descritos en la sección 3.2.3, página 60 con un error relativo de

$$\begin{aligned}
\frac{|.100101-.101|}{.100101} &= \frac{|.\,578\,125-.5625|}{.\,578\,125} \\
&= .027027...
\end{aligned}$$

que es menos del 3 %. Si se realizan sólo dos redondeos, primero a 4 bits en un formato intermedio y luego a los tres bits finales, se obtiene .100. La razón es que, cuando se redondea primero a 4 bits, el valor x inicial está en el centro del intervalo [.1001**0**, .1001**1**], por lo que se prefiere el número con el último bit en 0. Al redondear .1001 a 3 bits, por la misma regla se obtiene .100 con un error relativo de

$$\begin{aligned}
\frac{|.100101-.100|}{.100101} &= \frac{|.\,578\,125-.5|}{.\,578\,125} \\
&= .135135...
\end{aligned}$$

[14] `fma` es la pieza central en el conjunto de instrucciones de punto flotante de algunos procesadores, como el Itanium, RS6000 y PowerPC.

[15] Los bits extra, que pueden ser más dependiendo del procesador, o los bits de guardia en el caso de la resta.

que es más del 13 %. Obsérvese que en cada redondeo, el número inicial estaba exactamente en medio de las dos representaciones más cercanas. Por ello, utilizar variables en una sola precisión es mejor y más fácil de analizar en la mayoría de los casos, principalmente para los programadores que NO pueden controlar el redondeo de variables intermedias declarados con precisión mayor.

Por este motivo, calcular $a + b \times c$ con un solo redondeo es más eficiente, sin ser el único motivo:

- Se gana velocidad al ser una operación atómica y tener alto grado de paralelización, explotando bien el pipeline de los procesadores[16], a diferencia de sumar y multiplicar por separado (con redondeos también separados). Para detalles consúltese [118].
- Procesos como el producto de matrices acumulan la mitad de errores de redondeo (ver [143] donde se calcula con exactitud hasta el penúltimo dígito).
- Es sencillo extenderlo para calcular $a - b \times c$ y $b \times c - a$.
- La suma y la multiplicación se programan como casos particulares: $a + 1 \times c$ y $0 + b \times c$.
- Permite calcular cocientes y raíces cuadradas más eficientemente. Por ejemplo, hay instrucciones que aproximan con 8 bits de precisión ambas operaciones. A partir de allí, se refina con unas cuantas aplicaciones de Newton-Raphson con `fma`.

  Para la división de a/b, se parte de una aproximación (por tabla) de $1/b$ y se considera $f(x) = b - 1/x$, iterando con

  $$\begin{aligned} e_i &= 1 - bx_i \qquad \text{el error en la aproximación} \\ x_{i+1} &= x_i + e_i x_i \qquad \text{la siguiente aproximación.} \end{aligned}$$

  Ahora ya se puede multiplicar $a \times (1/b)$. Todas las operaciones redondeadas al más cercano. El residuo puede calcularse fácilmente como $a - b \times (a/b)$, siempre y cuando se respete el orden de las operaciones.

  Para la raíz, de nuevo se aplica Newton-Raphson a la función $f(x) = 1/x^2 - a$, con una buena aproximación de $1/\sqrt{a}$ y se itera

  $$\begin{aligned} e_i &= \frac{1}{2} - \frac{a x_i^2}{2} \\ x_{i+1} &= x_i + e_i x_i \end{aligned}$$

  y este caso, al igual que en la división, se llega en pocas iteraciones al NPF correctamente redondeado.

  Estos ejemplos se detallan y demuestran en [41] y [42] publicados altruistamente por Intel. El trabajo de Diran Sarafyan ya veía posibilidades similares [167].

También es necesario considerar que operaciones como $a \times b + c \times d$ no resultan tan obvias pues en general $\mathrm{fl}(\mathrm{fl}(a \times b) + c \times d)$ es distinto de $\mathrm{fl}(a \times b + \mathrm{fl}(c \times d))$ en $\mathbb{F}$ y el compilador no puede tomar la mejor decisión fácilmente, requiere ayuda. No es fácil dar esa ayuda, como explica Kahan en [106]. Esto ocurre cuando se implementa con `fma` el producto de un número complejo por su conjugado. Matemáticamente la parte

Otro ejemplo bien conocido es la hipotenusa de un triángulo, que se estudia con mayor detalle en el siguiente capítulo.

[16] El exponente de a se ajusta en paralelo con el producto $b \times c$, así que no hay esperas, sólo un poco más de espacio (registros suficientemente grandes) para garantizar la normalización de exponentes.

imaginaria es cero, pero no en $\mathbb{F}$[17].

Otro caso similar es al calcular el discriminante $b^2 - 4ac$ en la ecuación cuadrática. Con `fma` se pierde la relación válida en $\mathbb{F}$ de $\text{fl}(b^2) > \text{fl}(4ac)$ cuando $b^2 > 4ac$, ocasionando problemas en vez de resolverlos.

C99 sí permite activar o desactivar `fma` por bloques de instrucciones, por lo que el programador tiene el control para decidir cómo hacer los cálculos.

3.2.5. Infinito, NaNs y bit de signo

En este apartado, íntimamente relacionado con el de excepciones, la norma define las circunstancias que deberán dar como resultado $\pm\infty$ y los dos tipos de NaN, así como el significado y usos del bit de signo. Un aspecto importante es la *propagación* de los valores especiales. Esto significa que si se evalúa

$$\cos(e^{\frac{1}{xy}})$$

y xy causan overflow, su resultado es ∞, por lo que el cociente $\frac{1}{xy}$ debe resultar en 0, así que el resultado final es cos(1). Por supuesto, el programador debe poder enterarse de que ocurrió el overflow, pero el procesador no debe tomarse la atribución de detener la ejecución del programa. Más que otra cosa, este apartado de la norma trata de casos especiales, como es de esperarse al operar con los valores especiales de $\mathbb{F}$.

Infinitos

En el caso de $\pm\infty$, se considera un número exacto que debe lanzar una excepción cuando no sea un operando inválido. Es el resultado exacto con operandos válidos cuando ocurre una división entre 0, no es un error numérico. En caso de que los operandos sean válidos y ocurra un overflow, el resultado debe ser ∞ y señalarse overflow e inexacto.

La operación $a/0$ debe resultar en ∞, con el signo de a. Además, $x/0 = \infty$ y $x/(-0) = -\infty$.

Si los(el) operandos son inválidos y el resultado de la operación se almacenará en una variable, la variable debe resultar en NaN. Por ejemplo $\infty \times 0$, `fma`$(0,\infty,a)$ o `fma`$(\infty,0,a)$, $\infty - \infty$.

¿Cuánto es $0 \times \infty$+qNaN?

Otra característica importante es la *presubstitución*. Consiste en sustituir un valor o un procedimiento para evitar o corregir los problemas de las excepciones (ver el siguiente apartado). De esta manera, se espera que no sea necesario tomar medidas agresivas como interrumpir el programa en ejecución. Cada excepción evitada por anticipado, tiene un valor posible para la presubstitución determinado por la norma:

La excepción . . .	**se presubstituye con . . .**
Operación inválida	NaN
División entre cero	$\pm\infty$
Overflow	$\pm\infty$
Underflow	0
Inexacto	redondeo u overflow

[17] En el procesador PowerPC, desabilitan `fma` para compilar programas como MatLab.

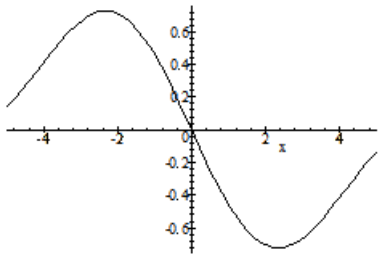


Figura 3.4: La función $(\cos(x) - 1)/x$.

A estos el programador puede agregar los suyos propios, dependiendo de la aplicación. La presubstitución permite al programador ignorar las excepciones o posponer su detección para mejores lugares del código, o simplemente pausar la ejecución durante la depuración. Son el único método completamente portable para manejar excepciones, pero si no se implementan correctamente pueden perjudicar seriamente el desempeño del programa.

Esto debiera enseñarse en todo curso de programación.

Por ejemplo. Si durante un proceso iterativo se va a calcular el cociente de dos números como parte de una condición como

```
if ( fabs(a/b) < eps )
```

entonces se está esperando que la relación vaya decreciendo. Si al inicio de las iteraciones alguna (o ambas) variables tomara como valor a ∞, por ejemplo leida como el resultado de la evaluación de otra función, como `a=IniIzq(x)`, entonces el programador puede prevenir situaciones excepcionales como $1/\infty$ o ∞/∞ substituyéndolas automáticamente con 0 o 1 respectivamente, con los elementos que provea el lenguaje de programación empleado. A la fecha, ningún lenguaje de programación, salvo C99 y Fortran2003, ofrece esta posibilidad que indica la norma, pero siempre es posible simularla con instrucciones como

```
if ( isInfinito(b) )
    Cociente = 0;
```

También puede impedirse que el cociente a/b se evalúe como ∞ alterando ligeramente los valores de a, b o ambos, dentro de los límites válidos de la aplicación, posiblemente con las funciones `nextUp` para mayor cuidado. Esta técnica la utilizan en la librería BLAS para el cálculo de eigenvalores de matrices de gran tamaño ($> 20000 \times 20000$).

La presubstitución permite evaluar límites que son imposibles con las operaciones convencionales en $\mathbb{F}$, como $(\cos(x)-1)/x$ pues el denominador puede ocasionar división entre cero, aunque la función tiene una gráfica bastante simple en 0. Ver figura 3.4.

Cuando la operación y valores lo amerite, se puede presubstituir con NaN. En todos los casos, las banderas correspondientes deben levantarse (ver el apartado de excepciones). Para más detalles e ideas, ver las notas de Kahan [109].

NaNs

En el caso de NaN, se definen dos tipos. El NaN silencioso (o *qNaN*) controlado por el usuario, adecuado para diagnóstico retrospectivo. El otro NaN (o *sNaN*) es una alarma que avisa de operaciones inválidas como como $\sqrt{-1}$, a^{NaN} y otras similares. Se diferencian en la codificación porque el primer bit (el menos significativo) de la mantisa es 1 para el qNaN y es cero en el otro caso. Si se trata de formato decimal, entonces qNaN posee el bit 6 del exponente (el campo de combinación G) apagado.

NaN se crea en una operación inválida cuando cualquier otro valor (incluyendo 0 e ∞) puede resultar más confuso, como $0 \times \infty$ y otras similares.

La comparación con NaN siempre es falsa, por lo que instrucciones como

```
if ( x != x )
    return Operacion_Invalida;
```

sirven para saber si `x` es NaN. El programador deberá tener cuidado de comprobar que el compilador que utiliza no cambia esta línea al optimizar sin precauciones, pues parecería que siempre es falsa en aritmética convencional. Por fortuna, muchos lenguajes definen funciones como `isnan` o `isinf` para identificar estos y otros valores, comunmente en el archivo de cabecera `math.h` de C.

Bit de signo

Las reglas generales de los signos que sigue la norma son

positivo+positivo = positivo
negativo+negativo = negativo.

Se define que $x - y = x + (-y)$ y la operación $x + x = x - (-x)$ debe preservar el signo de x aún cuando sea cero, es decir, sólo $(-0) + (-0)$ y $(-0) - (+0)$ dan -0.

Cuando en un program adebe decidirse saltar una discontinuidad, como en $\frac{1}{x}$ al acercarse a 0, pasar de un lado a otro de la discontinuidad es crítico. Si es crítico el signo del punto de cruce por la discontinuidad, se requiere algo más de información para conocer en qué extremo se está. Esta información extra es el signo del 0. Esto ofrece la posibilidad de diferenciar entre $f(-0)$ y $f(0)$ cuando sea válido en $\mathbb{F}$ (lo que depende de las propiedades de continuidad de f). El caso obvio es $1/a$.

También se define (por continuidad y propagación) que $\sqrt{-0} = -0$. El estándar no interpreta el bit de signo cuando se trata de NaNs, en ningún caso.

3.2.6. Excepciones

Una excepción ocurre cuando una operación termina en un resultado que no es adecuado para ninguna aplicación razonable. Esta es la definición que aparece en la norma y de ello se desprende que no hay una manera suficientemente formal que defina el concepto. Basándose en los primeros borradores de propuesta de la norma, Eggers y otros publicaron en 1978 un estudio proponiendo la forma de manejar excepciones [55].

Se especifíca la manera de manejar las siguientes situaciones:

- Operación inválida.
- División entre cero
- Overflow
- Underflow
- Inexacto

La especificación se da en términos de las características que deben cumplir las operaciones y operandos para señalar una excepción. Al ocurrir una excepción, debe enviarse la señal que corresponda y las *banderas de estado* cambiarse adecuadamente. Estas forman parte del registro de control y estado de punto flotante. Por las limitaciones de la APF, toda operación que redondee el resultado debe enviar una señl y levantar la bandera de inexacto.

No existe ningún lenguaje de programación que sea ortogonal, sin efectos colaterales (y al mismo tiempo útil).

Levantar banderas es un efecto colateral de una sentencia de código, lo que no es conveniente, desde el punto de vista de los expertos en la teoría de lenguajes de programación. Lo cierto es que la computación está plagada de efectos colaterales, tan solo medítese sobre el manejo de archivos o sincronización de procesos paralelos.

Los lenguajes debe definir medios para leer y modificar los valores de las banderas. Esto depende de la forma como el fabricante del procesador codifica su registro de control. Por ejemplo, en C se utilizan los archivos de cabecera `fenv.h` y `float.h` para poder acceder a las banderas de estado y verificar de qué excepción se trata o administrarlas en general.

En una publicación de 1993, John Hauser [84] expone con gran claridad las maneras como los programadores pagan la falta de cuidado de los fabricantes de procesadores y compiladores en cuanto a las características de un manejo de excepciones efectivo. Con su análisis, respalda como mejor solución los lineamientos de la norma 754, así como de LIA de IEC (véase la sección 3.1).

Hauser sintetizó claramente las 4 posibles soluciones ante underflow/overflow:

1. Fuerza bruta: reevaluar con un formato de mayor precisión o precisión múltiple.
2. Escalando la operación, por ejemplo dividiendo entre una constante y al final multiplicar para reescalar de nuevo. En ocasiones se pueden hacer varios intentos de escala. Los mejores factores son las potencias de β, ya que solo alteran el esponente, sin tocar la mantisa, excepto en los subnormales.
3. Presubstituir con ∞ o 0. Si en una operación como $10+1/(x+y)$, si $x+y=\infty$, entonces se puede sustituir sin riesgos a $x+y$ por el mayor número posible en su formato y terminar la operación, ya que aún así el redondeo resultará en 10 y no se perderá tiempo en la excepción de overflow, sólo levantando la bandera de inexacto.
4. Si es underflow, usar underflow gradual. En ocasiones, presubstituir con el menor número subnormal puede ser correcto.

Ya en 1988, Hull y otros [91] habían pesentado lineamientos generales para el manejo de excepciones, que reducía a un mínimo las excepciones predefinidas (incluyendo algunas no numéricas), sin limitar las que el programador pudiera agregar. En 1997, Hull publicó versiones complejas de funciones trigonométricas [92] donde se muestra un excelente manejo de las excepciones numéricas como apoyo al cómputo numérico.

Otro buen ejemplo es el manejos de los casos especiales de números complejos que presentó Kahan en [102].

Todo programador debe ver las excepciones como una oportunidad para hacer más cálculos, no para detenerlos. Como dice Kahan:

"Las excepciones son errores sólo cuando se manejan mal."

El salto es una medida desesperada la mayoría de las veces.

Pese a que la norma existe desde 1985 y que C99 (no C++) y Fortran2003 la cumplen, ninguno otro lenguaje o compilador permite a los programadores un manejo de excepciones consistente, comenzando con el significado que le dan al propio concepto de excepción. Por ejemplo, en C, C++ y Java se le llama así a la trampa (trap) o transferencia de control ante una situación no considerada por el usuario, incluyendo no numéricas. La excepción es la respuesta a la situación y el salto la única manera de resolverlo (principalmente en el caso de Java). Atrapar la excepción no es la mejor solución.

Otro ejemplo es el lenguaje Ada, donde la respuesta natural a un error numérico es abortar los cálculos, incluso ante un simple overflow. Lo peor es que los clasifica con la bandera NUMERIC_ERROR (o CONSTRAINT_ERROR en la versión de 1995) y el manejador no puede distinguir qué ocurrió de manera inmediata. En Basic, la instrucción `On Error goto ...` tiene la misma débil e incorrecta base.

Eso cambia el significado que tiene en la norma de ser un evento inusual que requiere atención especial. La respuesta debe ser la presubstitución y levantar una bandera, no dar un salto sin preguntar nada. Es claro que el salto es solo una de tantas posibles acciones ante una excepción, principalmente cuando se está depurando.

Detección de excepciones

La manera común de detectar una excepción es con una señal SIGFPE, que normalmente es una constante simbólica declarada en el archivo de cabecera `signal.h` de cada sistema operativo. SIGFPE es una de las señales definidas por POSIX (Portable Operating System Interface); la X viene de UNIX que es la plataforma original donde surgieron. POSIX es normado por IEEE.

No toda señal SIGFPE se refiere a un problema con punto flotante, pero no hay manera de cambiar el nombre de la señal sin comprometer la compatibilidad de la propagación de valores especiales. Las excepciones floantes que generan SIGFPE son las descritas en la norma, así como el overflow de un entero y un subíndice fuera del rango de un arreglo, aunque no en todos los lenguajes se señalan.

Si en un programa se ignora una señal SIGFPE el resultado es tipicamente impredecible. El programador debe crear la subrutina o función que se encargará de manejar la excepción. Lo más sencillo es utilizar un lenguaje que permita establecer con facilidad un manejador de excepciones. En C, lo que se hace es utilizar la función signal:

```
signal(SIGFPE, Manejador);
```

y `Manejador` es la rutina que será llamada cuando se de la señal SIGFPE. Cada compilador y sistema operativo define los detalles. Además, ANSI define variables externas

Plataforma	Ejemplo de llamadas
80x86	_status87, _clear87, _control87
Macintosh	getflag, setflag, TestException
Unix (System V)	fpgetsticky, fpsetsticky
SUN (SunOS)	ieee_flags

Tabla 3.10: Cada plataforma utiliza sus propias llamadas. Por la diversidad de estilos de semántica no es fácil que puedan coincidir.

como `errno` de lenguaje C, que indican el tipo de error, cuyo valor se actualiza cada operación.

En Fortran no se maneja al mismo concepto, pero se tiene acceso a las banderas del procesador con la función IEEE_GET_FLAGS. Por ejemplo

Listing 3.2: Usando excepciones con Fortran

```
1 Discriminante = sqrt(b**2 - 4*a*c)
2 call IEEE_GET_FLAGS(OUT_OF_RANGE,FLAGS)
3 if ( any(FLAGS) ) then
4     ...
```

Este método es transparente para el programador. Las instrucciones de las líneas 2 y 3 son sencillas y elegantes. Si el lector está interesado, consulte el reporte técnico ISO/IEC TR 15580 sobre manejo de excepciones en punto flotante para Fortran.

Es ideal tener códigos que se leen fácil.

De hecho es de esperarse que todo lenguaje que pretenda hacer cálculos numéricos con eficiencia disponga de llamadas de alto nivel `get_flags` y `set_flags` para leer y escribir las banderas de excepción.

Los compiladores de Microsoft (C/C++) han usado la bandera de operación inválida exclusivamente para detectar FP stack overflow, por lo que no está disponible para su uso según la norma.

El argumento fue que casi todo el tiempo la bandera estaría apagada.

La tabla 3.10 indica ejemplos de las llamadas nativas de cada arquitectura, no las de algún lenguaje en particular. Por ejemplo, en lenguaje C, en la versión C99 utiliza las llamadas `fegetexceptflag` y `feclearexcept`, independientemente de la plataforma.

En el mejor de los casos se tendría construcciones como

```
Intentar:
   Calculo riesgoso
Si ocurre al menos una excepcion
   Intentar calcular de otro modo o presubstituir
```

y esto comienza a verse en algunos pocos lenguajes desde hace pocos años.

Por último, si la aplicación no es demandante numéricamente, por ejemplo indicadores estadísticos generales, conversión de imágenes, procesos gráficos y otros, se dispone de formas de omitir señales, como `signal(SIGFPE, SIG_IGN)` que ignoran toda señal[18]. Incluso opciones de compilación como `cc -mieee` para anular la verificación.

[18] Realmente direccionan toda señal de punto flotante a un manejador vacío.

Dos problemas famosos

El error costo 500 millones de dólares, mas otros 7000 del desarrollo.

Ya hay ejemplos famosos del manejo incorrecto de excepciones numéricos, como el cohete Ariane 5. En el lanzamiento, un sensor reportó una aceleración tan fuerte que ocurrió un overflow al convertir de flotante a entero. En vez del manejo esperado (levantar las banderas de overflow e inexacto y regresar el resultado) se activó el sistema de diagóstico (que quedó activo por error), que lanzó información para depuración posterior a una región de memoria en uso. El resultado pudo simplemente ser ignorado por el sistema de control de los motores y continuado con la trayectoria normal, en vez de considerar que era necesario un brusco cambio de dirección. ¿Y si un problema similar ocurriera con el sistema que activa los explosivos que evitan que caiga en una región habitada?

El costo pudo ser en vidas humanas.

El otro caso es el del navío de combate Yorktown, cuyo sistema se quedó esperando el reinicio de la computadora ante una división entre cero calculada por un campo en blanco (por error) en una base de datos. De seguirse la norma, se hubiera calculado ∞ y continuado con etapas correctivas, en vez de navegar en círculos de manera incontrolada hasta que la tripulación logró reiniciar la computadora. ¿Y si hubiera ocurrido en un puerto?

3.3. Anexos de la norma

Consisten en información general, razonamientos y recomendaciones para diseñadores y fabricantes de hardware y software, pero realmente es en lo que NO se puedieron poner de acuerdo pero consideraron importante.

A **Bibliografía**. Documentos en los que se apoyó el trabajo, así como referencias que explican y justifican con mas detalles algunos aspectos.

B **Evaluación de expresiones**. Expone lineamientos sobre cómo debieran evaluarse, optimizarse y asignarse las expresiones en general. Los compiladores que cambian expresiones como `e=((a+b)-a)-b` por `e=0` no respetan la norma pues no permiten rescatar errores de redondeo. El uso de banderas de excepción tampoco tiene una secuencia obvia para los compiladores.

C **Ajuste de tamaño** (widento). Sugerencias para cuidar el tamaño de los formatos intermedios y finales para permitir el redondeo controlado por el programador. Es importante para evitar problemas como el doble redondeo. 31 de las operaciones incluidas en la norma requieren de este cuidado.

Es una lástima que x^n no se incluya en esa lista aún, pero todavía falta determinar sus peores casos.

D **Funciones trascendentes elementales**. Se enlistan las funciones elementales mínimas que un programador debiera tener disponible. Son públicos los métodos que garantizan funciones elementales correctamente redondeadas (para todos los modos de redondeo), aunque a veces en dominios limitados, por lo que debiera ser posible disponer de librerías matemáticas exactas hasta .5 ulp. Se trata de garantizar, en la medida de lo posible, preservar propiedades como la monotonicidad y simetría.

E **Modos alternativos de excepciones**. Se recomiendan las formas como los lenguajes debieran permitir al programador definir sus propias excepciones, así como su respuesta, tanto para un programa completo, módulos individuales e incluso subrutinas o funciones.

F **Operaciones básicas con arreglos**. Debido a que operaciones como la norma de un vector pueden resultar en un overflow por a los cálculos intermedios ($\|\mathbf{x}\| = \sqrt{x_1^2 + \cdots x_n^2}$ y para alguna i, x_i^2 mayor que el máximo NPF en $\mathbb{F}$), se enlistan las funciones que debieran poder calcularse evitando este fenómeno. Son solo 3: $\|\mathbf{x}\|$, $\mathbf{x} \cdot \mathbf{y}$ y $\prod(x_i - y_i)$.

G **Lineamientos de depuración**. Describe las facilidades que debe tener un programador para depurar programas numéricos y encontrar errores, así como motivos de sensibilidad numérica y excepciones. Esto resulta en sugerencias para desarrolladores de depuradores y la posibilidad de disponer (algún día) de mayores controles en tiempo de depuración.

Estos controles inlcuyen actividades en tiempo de depuración como:

- Cambiar el modo de redondeo, para detectar programas muy sensibles por mal condicionamiento de problemas[19].
- Alterar el manejo de excepciones, aún cuando no se disponga del código fuente.
- Pausar el programa cuando ocurra una excepciones específicas.
- Prender o apagar las banderas de excepcion en cualquier momento.
- Guardar el historial de excepciones por subprograma para su análisis posterior, con sus respectivas banderas e historia. Esto al menos en cada NaN producido.
- Detectar las 24 cadenas de bits en dpd (ver 3.2.2) que no representan a ningúna combinación de números en los formatos de base 10.
- Detectar regiones de memoria (variables) sin valores iniciales establecidos.
- La posibilidad de que cada variable que entra a la pila del programa levante una bandera de NaN para que quede la referencia de su origen, aún teniendo un valor correcto.

Por ejemplo al momento de declararla.

Todo programador de rutinas científicas desearía disponer de un compilador así y todo indica que lo seguirá deseando por algún tiempo.

Otros anexos que no se incluyeron fueron sobre características de lenguajes, incompatibilidad con la norma previa, procesamiento de subnormales con hardware y otros formatos numéricos posibles (incluyendo tipos de 256 bits que tal vez aparezcan en la próxima revisión).

3.4. Lo que faltó en la norma

Pese a todo el trabajo detrás de la norma y a los excelentes científicos que participaron, han quedado vacíos, algunos ocasionadas posiblemente por intereses comerciales, pero otros divergencias académicas auténticas. De los borradores del trabajo se sabe que en muchas cosas el debate fue intenso y a veces no llegaron a ningún acuerdo.

[19] Sí es posible confirmar alta sensibilidad al redondeo pero su exposición es accidental. Un esquema parecido es el de repetir los cálculos, permitiendo que programas donde los errores de redondeo se compensan beneficiosa pero aleatoriamente, queden expuestos, detectando sensibilidad. El cómputo repetido con modo de redondeo aleatorio no es fácil de programar bien y es lento: nada se compara con un buen análisis matemático del redondeo.

Sin pretender encontrar y exponer todos lo vacíos de la norma, sí hay algunos importantes para quienes hacen programas numéricos. Las siguientes son consideraciones generales, incluidas las del autor. Dentro de las cosas que no se detallan o faltaron:

1. Decidirse por un solo formato decimal. Ante la posibilidad de formatos bcd y dpd para formato decimal, solo queda esperar que la portabilidad de software (programas y datos) no tenga problemas serios. Aún no teniéndolos, esto implica mayor cantidad de código en alguna parte, ya sea el compilador, el procesador y, en el peor de los casos, el programador final.
2. Para el formato decimal, debiera haber una forma de indicarle al programador que una operación o una variable tiene una de las 24 combinaciones de bits no considerados en el dpd. Si las excepciones no se habilitan, ¿cuál será el resultado si se propaga una de estas combinaciones? No es obvio que su decodificación debiera resultar NaN. La recomendación es verificar la bandera de estado después de cada operación sospechosa, lo que podría complicar los códigos. El resultado es que la presubstitución con fines de propagación deja de ser eficiente.

Podría ser útil disponer en tiempo de depuración de una variable alterna en base 2.

3. El anexo sobre funciones elementales redondeadas correctamente hasta el último bit debiera ser obligatorio, más aún siendo públicas las técnicas para hacerlo. La excepción son las funciones x^y (incluyendo el caso particular x^n), para las que aún no se tienen implementaciones exactas, por desconocerse los peores casos.
4. El anexo sobre depuración debiera ser obligatorio. Los fabricantes de procesadores o depuradores no incluirán algo que es opcional. Siempre habrá algo pendiente que tenga mayor prioridad sobre lo que simplemente se recomienda. Actualmente, los depuradores permiten ver los valores de los registros del procesador, pero no actividades deseables como detener la ejecución hasta que se levanta la bandera de inexacto o exclusivamente la de underflow.

Todavía menos cuando lo obligatorio lo incluyen parcialmente.

5. Cuando no se dispone de un depurador como el deseado, algunas anomalías de redondeo pasan de unas versiones a otras de procesador y compiladores, posiblemente cambiando el síntoma anómalo. Sin herramientas eficaces para su detección, muchas persistirán, con resultados inesperados. Sin depuradores eficientes, han sido décadas de experiencia con punto flotante base 2, lo que ha conseguido la valiosa información de cómo hacer las cosas. Con el creciente uso de los tipos decimales, ¿cuánto tiempo se necesitará?
6. La norma no dice que si se define una variable decimal en dpd, la inspección de su valor durante la depuración debe mostrar tanto la codificación interna como el valor decodificado, aunque sería de esperar que todo compilador que incluye tipos decimales ofrezca esta característica que no implica muchos problemas.
7. Como el manejo alternativo de excepciones (las definidas por el usuario) no es obligatorio, sino sólo aparece como información en un anexo, el programador no puede confiar que su programa será portable, pues el manejo de excepciones es diferente en distintas arquitecturas. Singnifica que el programador no puede confiar en expresiones como $e^{-\frac{1}{x}}$, donde si $x = 0$ el cociente da infinito y la exponencial es 0.
8. No se indica qué ocurrirá al dispararse dos excepciones a la vez, como inexacto y underflow. Si por el ambiente de programación, una de ellas obliga un salto o una trampa, no es claro si el compilador debe indicarlo o si el depurador permitirá seleccionar el camino por seguir.

9. No es fácil regular la sintaxis de los predicados que detecten o controlen las excepciones, pues eso depende de la semántica de cada lenguaje, así como sus modos y estilo. El problema verdadero es que aún siendo el mismo lenguaje, al pasar a otras arquitecturas las llamadas pueden ser distintas y cada fabricante defiende sus propios esquemas. En estos casos, la portabilidad es inversamente proporcional a la legibilidad de los códigos, que es primordial para efectos de mantenimiento.
10. En ambientes multihilo (multithreading), que se espera que dominen el mundo de los procesadores de aquí en adelante, el proceso hijo hereda del padre parámetros de precisión, modo de redondeo y manejadores de excepción, pero no se define si también hereda las banderas de estado o si se reinician en todas en cero para este nuevo hilo (que es lo que habría que esperar: el nuevo hilo no ha realizado ninguna operación)[20]. Esto fue discutido desde 1992 por Goldberg en [76].
11. La observación anterior debiera extenderse al uso de la función `fork()`, es decir, es claro que algunos atributos deben heredarse a los procesos hijos, pero no necesariamente todos. El programador es quien debe tomar la decisión.
12. Posiblemente queda fuera del contexto inicial y propósito de la norma, pero la programación de procesos sincronizados debe analizarse con mayor cuidado.
13. Valdrían la pena análisis como el que Evgenija Popova publicó en 1995 [154], que si bien concluye con las buenas perspectivas de las normas 754 y 854, también pone de manifiesto sus inconsistencias formales (algebraicamente).

Mucho de lo anterior fue discutido. Cuando un tema fue enviado como opcional o informativo (es la misma) a un anexo, los participantes en la reforma fallaron en ponerse de acuerdo, condenando a toda la comunidad de programadores y analistas numéricos a realizar análisis de punto flotante cada vez más complicados (lo sencillo poco a poco se cubre) sin disponer de herramientas que orienten y diagnostiquen con eficiencia, dando información amplia y útil sobre las características de las anomalías.

3.4.1. Aspectos educativos

Otro aspecto problemático que no depende de la norma, cae en la educación. Por su parte, los profesionales de la computación reciben (o debieran hacerlo) en algún momento instrucción sobre programación numérica, al menos generalidades. Esto con el objetivo de una orientación mínima que les permita saber cuándo vale la pena preocuparse al desarrollar sistemas. En México, la ANIEI[21] define perfiles profesionales en los que se han determinado una proporción de conocimientos diferente, pero sin especificar detalles. ¿Cuánto le cuesta a una empresa el retraso al buscar errores numéricos en los módulos financieros de un sistema con fecha límite de entrega que no podrá cumplirse? ¿Qué tan frecuentemente ocurre esto? Sin información, la decisión de qué tanto enseñar sólo puede ser por apreciaciones personales o de grupo.

Por otra parte, los profesionales de otras ramas (químicos, economistas, físicos-matemáticos, geólogos, ...) hacen uso de paquetería para obtener resultados. En la mayoría de los casos no tienen problemas numéricos graves pues utilizan menos precisión que la que aporta la computadora. En el caso de desarrollar sus propias rutinas,

La SHCP no perdona ni un centavo y es cuando los errores numéricos pegan donde más duele.

[20] En el caso de los uthreads, lo mejor sería compartir el estado de PF del verdadero thread del que penden.

[21] Asociación Nacional de Instituciones de Educación en Informática, A.C.

un problema sería la inestabilidad, pero es mayor cuando se realiza investigación y se usan modelos matemáticos un poco más complejos.

Es el caso de los actuarios, estadísticos y otros investigadores que incluso usan simulaciones por computadora. Es fácil detectar anomalías cuando un resultado diverge mucho, pero cuando todo parece estar bien y los resultados tiene la mitad de las cifras correctas, tal vez ya no es tan sencillo darse cuenta. ¿Hasta dónde se le debe advertir a estos profesionales? Esa es una discusión para otro lugar.

* * *

Pese a todo lo anterior, esta sigue siendo una de las normas con grandes beneficios y de uso muy extendido. Quizas la siguiente revisión, dentro de 15 años, con la experiencia creciente de los programadores, características de nuevos lenguajes y procesadores novedosos, permita que el siguiente comité se ponga de acuerdo más fácilmente. Desde otro punto de vista, ya no es posible pensar en no disponer de una norma, aún cuando sea parcialmente respetada por el mundo de la computación.

El éxito en la solución de muchos problemas numéricos de la computación moderna es un hecho transitorio, como lo muestra el libro de Bo Einarsson [56], donde se trata el tema del incremento de los problemas críticos de la exactitud esperada en las nuevas y futuras computadoras. Habrá de seguirse revisando esta norma conforme evolucionen lenguajes, procesadores y la computación en general, pues con los modelos de cómputo actuales no se dispondrá de precisión infinita jamás.

Capítulo 4

Programando la norma

Apoyarse en la norma es útil, pero durante las décadas en que no la había se hicieron gran cantidad de buenos cálculos y se desarrolló experiencia valiosa. Mucha de esa experiencia sigue siendo útil y se complementa con con los lineamientos que casi siguen los fabricantes de hardware y software. En honor a eso, se presentan primero algunas ideas generales de cómo minimizar los problemas de exactitud de los cálculos numéricos en una computadora.

En los ejemplos de programación, se usará pseudocódigo, lenguajes C y Fortran. Como se comentó con anterioridad lenguajes populares como C++ o Java no ofrecen garantías suficientes (ver [111]).

4.1. Antes de la norma

Desde los inicios del cómputo numérico, las limitaciones de la computadora fueron compensadas con el ingenio de muchos científicos. Se desarrollaron lo que en su momento fueron trucos y que pasaron a ser técnicas estándar en poco tiempo, para compensar las deficiencias de exactitud, principalmente al tratar de escribir código que funcionara lo más parecido posible entre las diversas arquitecturas.

En todo tipo de problemas, sencillos o complejos, surgieron este tipo de ideas. Desde sumar hasta calcular eigenvectores de matrices muy grandes, ecuaciones diferenciales o simulaciones. Se dice que las primeras experiencias en la programación con números de punto flotante fueron de Wilkinson y su equipo. Cuando trabajaba en el National Physical Laboratory de Inglaterra, trabajó con Alan Turing en el diseño de la computadora ACE[1] al tiempo de estudiar análisis numérico[2]. Turing le solicitó en 1946 que programara un conjunto de rutinas para trabajar en APF con la versión 5 de la ACE, que inicialmente operaba en punto fijo. Estas fueron las primeras rutinas de

James Hardy Wilkinson (1919 – 1986), analista numérico inglés, recibió la medalla Turing por el desarrollo de algoritmos numéricos, así como su análisis programación.

[1] Y en 1948 la Pilot-ACE, una vez que Turing abandonó NPL, dejando a la cabeza a Wilkinson. La Pilot-ACE era rápida y pequeña para su época, terminada en 1950. Trabajaba a 1MHz y podía resolver un sistema de 10 ecuaciones lineales en 1 segundo. No disponía de multiplicación, por lo que se usaba una rutina de sumas sucesivas.

[2] Al lado de expertos en el área como Goodwin, Fox, Olver y Clenshaw.

punto flotante, posteriormente pulidas por otros miembros del equipo. Según Wilkinson, Turing fue de los primeros que concientizaron en la importancia de la velocidad de los cálculos, obligando a pulir la APF, con lo que ganaron experiencia antes de tener construida la primera computadora. Allí progresó el análisis de error para la APF, en 1955-56, que posteriormente se publicaría en [187] y [188].

Esa experiencia que ganaron cada uno de los equipos que construyeron las primeras computadoras, fue revelado cuando ya habían pasado las secuelas de la segunda guerra mundial y no se consideraban secretos militares. A finales de los 50 se comenzaron a conocer una bateria amplia de técnicas de cálculo numérico.

Uno de los primeros ejemplos fue la diferencia de cantidades casi iguales

$$a - b = \frac{a}{2} - b + \frac{a}{2} \tag{4.1}$$

para evitar la cancelación de cifras significativas. Pequeños detalles como estos se fueron juntando, como la evaluación de `((0.5D0-x)+0.5D0)` en vez de `(1.0D0-x)` recomendado en las computadoras S/360 de IBM. El ingenio dio pie a transformaciones de expresiones y reescritura de fórmulas para minimizar los errores de redondeo. Actualmente, las precauciones de la norma permiten no necesitar de técnicas como la fórmula (4.1) que en su momento fueron de gran utilidad.

Se podía escribir $p \times 1$ o bien $p/1$.

Un ejemplo que causa impresión es que en algunas computadoras era preferible escribir `if ( p*1 == 0)` en vez de `if ( p==0 )` no sólo porque no se usaban números subnormales, sino por la deficiente representación de los números más pequeños[3].

Entre los ejemplos notables está el formato de *precisión múltiple* (consúltense [79] y [93]) que consiste en emplear dos variables para representar un número en más alta precisión (el doble y el triple respectivamente, en esa época, 84 y 132 bits). Por ejemplo,

$$x = .11122233344455566677788899$$

puede representarse como la suma de $x_h + x_l$:

$$x_h = .11122233344455$$
$$x_l = .666777888999 \times 10^{-3}$$

Esta técnica es empleada en muchas librerías de funciones elementales para garantizar un error de .5 ulp con el doble de la precisón máxima, pero la norma no considera esta representación.

Sin embargo, sí garantiza que tanto x_l como x_h van a estar correctos hasta .5 ulp y con eso se garantiza que la suma también. Técnicas como esta existen desde hace mucho tiempo, por ejemplo la que publicó Dekker en 1971 en [48], para cierto tipo de APF de esa época, restringidas a una precisión par ($p = 2k$). Las técnicas, presentadas originalmente en Algol, aún son útiles, por ejemplo, la suma de $z = x + y$ se calcula en dos partes como $z_h + z_l$, que puede programarse en C con una simple macro ($z_h + z_l$ se representan con `z+zz`) en el código 4.1.

[3] Dekker y Bus comentan en [28] en 1975 que sería preferible comparar contra una cantidad muy pequeña, que es la idea actual de ε_M y justifican el uso de operaciones dependientes de la arquitectura lamentándose de que no exista algúnb acuerdo internacional al respecto.

Listing 4.1: Macro para duplicar la precisión al sumar.

```
#define SUMA(x,y,z,zz) \
          z=(x)+(y);     \
          zz=(ABS(x)>ABS(y)) ? (((x)-(z))+(y)) : \
                               (((y)-(z))+(x));
```

La suma **z+zz** es exacta y corresponde a $x + y$. A esto se le conoce como *transformación libre de error*.

Dekker incluso da una extensión para cuando se dispone ya de cantidades con precisión múltiple $x_h + x_l$ y $y_h + y_l$:

Listing 4.2: Macro para duplicar la precisión al sumar cantidades con precisión ya duplicada.

```
#define SUMA2(x,xx,y,yy,z,zz,r,s) \
{ double r, s;          \
  r=(x)+(y);            \
  s=(ABS(x)>ABS(y)) ? (((((x)-r)+(y))+(yy))+(xx))    : \
                      (((((y)-r)+(x))+(xx))+(yy));     \
  z=r+s; zz=(r-z)+s; \
}
```

En este caso, la suma ya no puede ser exacta, pero se garantiza un error máximo de

$$4.94 \times 10^{-32}(|x_h + x_l| + |y_h + y_l|).$$

Estas técnicas sólo cubren suma, resta y multiplicación (la división siempre es más difícil) y son empleadas por IBM en su IBM Accurate Mathematical Library de 2001. Estos trabajos fueron iniciados más de 10 años antes, como presentan Shmuel Gal y Boris Bachelis en [69], garantizando 1 ulp correctamente redondeado con una seguridad del 99.7 % en el caso de las funciones elementales.

Hay formas de garantizar .5 ulp con el doble de la precisión máxima en la suma sin suponer que el error de redondeo es menor que .5 ulp ni que la precisión es par, según presentó Linnainmaa, en 1981 [126], generalizando el resultado de Dekker [48] por sugerencia de Kahan. En este caso, la transformación libre de error para la suma es

Listing 4.3: Macro mejorada para duplicar la precisión al sumar.

```
#define SUMAMEJ(x,xx,y,yy,z,zz,r,s)    \
{ extended r,q,s;      \
  r=(x)+(y);           \
  q=(x)-r;             \
  s=(((q+(y))+(x)-(q+r)))+(xx))+(yy))   \
  z=r+s; zz=(r-z)+s; \
}
```

El método es similar al de Dekker, pero con garantías teóricas adicionales, en particular porque se apoya en la primera norma de IEEE, que en ese momento estaba en discusión. Hay métodos adicionales que ya incluyen la división, además de resta y multiplicación. Por ejemplo, Linnainmaa formaliza la separación inicial de $x = x_h + x_l$

como

$$\begin{aligned} t &= \text{fl}(x(1+\beta^{p-k}) \\ x_h &= \text{fl}(t-(t-x)) \\ x_l &= \text{fl}(x-x_h) \end{aligned} \tag{4.2}$$

donde p es la precisión, como de costumbre, k es un entero con $1 \leq k < p$, x_{hi} tiene k dígitos y x_{lo} los restantes $p-k$. Lo anterior suponiendo que el redondeo es correcto.

Existen trabajos para el análisis de errores automatizado, notablemente los de Webb Miller, de 1975 [136] y 1978 [137], en los que diseña rutinas en Fortran para perturbar sistemas de ecuaciones en busca de inestabilidades numéricas.

Librerías matemáticas y otros recursos se hicieron del dominio público, ampliándo las posibilidades de cómputo numérico confiable. En 1975, William James Cody presentó la FunPack en [32], una colección de funciones especiales programadas en Fortran, cuidadosamente diseñadas para ser robustas y eficientes en distintas plataformas.

Con la experiencia de FunPack, Cody publicó un excelente libro en 1980 [38] sobre la programación de funciones elementales, así como su artículo de lineamientos de programación [33], base para la librería ELEFUNT de funciones elementales. Estas y otras fuentes forman un aglutinado de ideas no solo de programación, sino de métodos para probar lo correcto de las rutinas. Cody posterirmente publicó la versión para variables complejas CELEFUNT [35]. Ambas librerías están programdas en Fortran, disponibles en NetLib.

4.1.1. Cambio de fórmulas

El descubrimiento de la cancelación catastrófica motivó el replanteamiento de fórmulas comunes como

$$\frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

para evitar el problema de que la magnitud de b^2 sea parecida a la de $4ac$, o que $b^2 \gg 4ac$, ocasionando la diferencia de cantidades casi iguales en el numerador. En la segunda parte se analiza este caso detalladamente.

Otro ejemplo famoso es el de la fórmula de Herón para calcular el área de un tríángulo a partir de sus lados

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

donde $s = (a+b+c)/2$ es el semiperímetro. Si uno de los lados es sumamente pequeño, ocurrirá una cancelación que puede ocasionar un error relativo grande. En 1986 Kahan propuso reorganizar las aristas para que $a \geq b \geq c$ y emplear la fórmula

$$A = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}$$

para garantizar un error más pequeño. El lector ya debe ser sensible a la importancia de respetar los paréntesis.

Muchas otras operaciones más complicadas que estas fueron desarrolladas, así como sugerencias de cómo evaluar o reescribir expresiones matemáticas. La tabla 4.1

Para calcular . . .	**escribir . . .**
$1-\cos(x)$	$2\,\mathrm{sen}(x/2)^2$
$\frac{1}{\cos(1-x)}$	$\frac{2}{\mathrm{sen}\,\sqrt{x/2}}$
e^x-1	$\tanh(x/2)(e^x+1)$
$\ln(x+1)$	$2\arctan(\frac{x}{x+2})$
$\tanh(x)$	$\frac{e^{2x}-1}{e^{2x}+2}$
$\tanh^{-1}(x)$	$\ln(\frac{2x}{1-x}+1)/2$
$\sqrt{1-x}-1$	$\frac{x}{\sqrt{1-x}+1}$
$1-\sqrt{1-x}$	$\frac{x}{\sqrt{1-x}+1}$
$\sqrt{x^2+1}-1$	$\frac{x^2}{\sqrt{x^2+1}+1}$
$\sqrt{x+\delta}-\sqrt{x}$	$\frac{\delta}{\sqrt{x+\delta}+\sqrt{x}}$

Tabla 4.1: Estas deben usarse con cuidado pues es sólo para la cercanía a ciertos valores críticos.

muestra algunos ejemplos famosos[4]. El problema de redondeo o cancelación no ocurre en todo el dominio, sino en cierto punto crítico, cerca de una asíntota o por cancelación catastrófica. Es de esperarse que en este momento el lector pueda determinarlos.

En este caso, la tabla sólo muestra algunos ejemplos de manipulaciones algebraicas que se ha considerado son numéricamente más estables. De algunas de estas hay más expresiones posibles, códigos eficientes o rutinas disponibles en la librería del compilador por lo que poco a poco se usan menos. Por ejemplo, hay casos donde el código estable puede ser tan simple como eficiente, como la versión de Kahan para e^x-1:

Listing 4.4: Manera correcta de calcular e^x-1.

```
1 double expm1(double x)
2 {
3     double u = exp(x);
4     if (u == 1.)
5         return 1.;
6     return (u-1.)/ln(u);
7 }
```

Este procedimiento simple tiene gran cuidado y garantiza un resultado correcto. La división reduce los errores generados por e^x-1 y $\ln(e^x-1)$.

4.1.2. Hipotenusa

La hipotenusa de un triángulo se calcula con $c=\sqrt{a^2+b^2}$. De hecho es un caso particular de la *norma* de un vector. Un intento inocente es la función

[4]Busque el lector dónde fallan las expresiones de la primera columna.

```
double Hipotenusa(double a, double b)
{
    return sqrt(a*a+b*b);
}
```

Los valores intermedios son positivos y pueden ocasionar underflow u overflow, ya sea a^2 o b^2 y la suma pude ocasionar overflow. La solución típica para evitar overflow es *escalar*:

$$c = s\sqrt{\left(\frac{a}{s}\right)^2 + \left(\frac{b}{s}\right)^2} \tag{4.3}$$

y s se selecciona para minimizar errores de redondeo. Normalmente $s = \text{máx}(|a|, |b|)$ y la fórmula queda

$$c = s\sqrt{1 + \left(\frac{\text{mín}(|a|, |b|)}{s}\right)^2}.$$

La otra opción es imponer a s el mínimo valor para el que $\left(\frac{a}{s}\right)^2 + \left(\frac{b}{s}\right)^2$ no causa overflow y que sea potencia de 2. Si L es el mayor número en el formato usado, entonces el valor mínimo de s se aproxima buscando la siguiente potencia de 2 mayor que

$$\begin{aligned} s &> \frac{1}{L}\sqrt{a^2 + b^2} \\ &= \sqrt{\left(\frac{a}{L}\right)^2 + \left(\frac{b}{L}\right)^2} \end{aligned}$$

y se recalcula 4.3. Este esquema sigue siendo un tanto inocente pues evita el overflow pero promueve el underflow, además de ser demasiados cálculos para encontrar c. Como puede verse, no es tan sencillo calcular una hipotenusa. Por fortuna, casi todos los compiladores disponen de una función de librería `hypot` que se espera que esté bien optimizada.

En sí lo que se busca es un valor s para escalar poder calcular $(sa)^2 + (sb)^2$ sin riesgos. Si ambos a y b son muy grandes s será un valor pequeño y viceversa. Pero no debe ser tan pequeño o grande que ocasione problemas de precisión y mucho menos overflow. Además, si s es una potencia de β no se ocasionarán problemas de redondeo.

Los NPF máximo y mínimo son

$$\begin{array}{ll} \text{Mayor:} & R = \beta^{e_{\text{máx}}}(1 - \beta^{-p}) \\ \text{Menor:} & r = \beta^{e_{\text{mín}}}. \end{array} \tag{4.4}$$

El menor número es realmente el subnormal β^{e+1-p} pero puede ocasionar problemas de exactitud, por lo que se prefiere aquí el mínimo normal.

En 1978, James Blue, de los Laboratorios Bell, publicó [16], donde presenta una forma que cuida estos aspectos para evitar problemas de overflow o underflow. Realmente su algoritmo evalúa la norma de un vector. Lo que hace es calcular las constantes

Listing 4.5: Método de Blue para calcular la hipotenusa.

```
a = |a|; b = |b|;
c = cm = cM = 0.0;
if ( a>T )
    cM = (a/S)*(a/S);
else if ( a<t )
    cm = (a/s)*(a/s);
else
    c = a*a;
if ( b>T )
    cM += (b/S)*(b/S);
else if ( b<t )
    cm += (b/s)*(b/s);
else
    c += b*b;
if ( cM>0.0 ) {
    if ( sqrt(cM) > R/S )
        return R*R; /* Overflow inevitable */
    if ( c>0.0 ) {
        xm = min(sqrt(c),S*sqrt(cM));
        xM = max(sqrt(c),S*sqrt(cM));
    } else
        c = S*sqrt(cM);
} else if ( cm > 0 )
    if ( c>0.0 ) {
        xm = min(sqrt(c),s*sqrt(cm));
        xM = max(sqrt(c),s*sqrt(cm));
    } else
        c = s*sqrt(cm);
if ( xm < sqrt(eps)*xM )
    c = xM;
else
    c = xM*sqrt( 1 + (xm/xM)*(xm/xM) );
return c;
```

t, T, s y S a partir de las definiciones (4.4) de la siguiente manera.

$$\begin{aligned} t &= \beta^{\lceil (e_{\text{mín}}-1)/2 \rceil} \approx 1.491 \times 10^{-154} \\ T &= \beta^{\lfloor (e_{\text{máx}}-p+1)/2 \rfloor} \approx 1.998 \times 10^{146} \\ s &= \beta^{\lfloor (e_{\text{mín}}-1)/2 \rfloor} \approx 7.458 \times 10^{-155} \\ S &= \beta^{\lceil (e_{\text{máx}}-p+1)/2 \rceil} \approx 1.998 \times 10^{146} \end{aligned}$$

Los valores aproximados corresponden al formato de doble precisión. Estos darían problemas de overflow o underflow al elevarse al cuadrado. Si a o b se salen del rango $[t, T]$, se requiere escalar, cada uno independientemente. Si alguno queda en ese rango, se puede calcular su cuadrado sin problemas.

El procedimiento escala cada valor independientemente. En el segmento de código 4.5 se usan 3 acumuladores, `cm` para escalar underflows, `c` cuando no se requiere escalar y `cM` para evitar overflows.

El pseudocódigo 4.5 se acerca a la versión que Blue presenta y son necesarias algunas consideraciones adicionales para obtener buen código de C. Por ejemplo, la

Listing 4.6: Método Moler-Morrison para calcular la hipotenusa.

```
1  p=max(|a|,|b|);
2  q=min(|a|,|b|);
3  if ( p==0 )
4      return 0;
5  pf = p;
6  q = q/p;
7  r = q;
8  p = 1;
9  while ( 1 ) {
10     r *= r;
11     s = r+4;
12     if ( s==4 )
13         return p*pf;
14     r /= s;
15     p += 2*s*p;
16     q *= r;
17     r = q/p;
18 }
```

división `/S` debiera cambiarse por el producto con el recíproco y realizarse una sóla vez al elevar al cuadrado. El lector ya debiera poder escribir el código adecuado.

En su artículo, Blue demuestra que el algoritmo anterior evita underflows y que el overflow ocurrirá sólo cuando $\sqrt{a^2+b^2} > R$.

La $\sqrt{}$ es más costosa que la división.

Un interesante procedimiento se debe a Cleve Moler y Donald Morrison, presentado en [138]. El procedimiento es iterativo, sin necesitar calcular la raíz cuadrada. En escencia es el método de aproximación de raíces de Halley, que se estudia con más detalle en la segunda parte.

En el código 4.6, que presenta el método de Moler–Morrison, va aumentando p y disminuyendo q. El método de Halley tiene convergencia cúbica, lo que significa que cada iteración triplica la cantidad de cifras correctas. Es más lento que el de Blue (por ser iterativo y no directo) pero muy preciso, pues con pocas iteraciones (de 3 a 4) se llega a p bits correctos en el resultado.

4.1.3. División compleja

Otro ejemplo muy ilustrativo es cuando se trabaja con números complejos, la división tiene sus problemas. Sean $z = a + ib$ y $w = c + id$. El cociente $z/w = x + iy$ se calcula como

$$\begin{aligned} \frac{z}{w} &= \frac{a+ib}{c+id} \\ &= \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2} \end{aligned}$$

Con $|w|$ debe entenderse el módulo del número complejo.

y puede ocurrir overflow si $|w| > \sqrt{\beta}\beta^{\frac{e_{\text{máx}}}{2}}$. Para evitar este problema, Robert L. Smith en 1962 publicó un método [172] para escalar la división y evitar el overflow,

Listing 4.7: Método Stewart para dividir números complejos.

```
1  F = false;
2  if ( |d|>=|c| ) {
3     Intercambiar(c,d);
4     Intercambiar(a,b);
5     F = true;
6  }
7  s = 1/c;
8  t = 1/(c+d*(d*s));
9  if ( |d|>=|s| )
10    d = s;
11 if ( |b|>=|s| )
12    x = t*(a + s*(b*d));
13 else if ( |b|>=|d| )
14    x = t*(a + b*(s*d));
15 else
16    x = t*(a + d*(s*b));
17 if ( |a|>=|s| )
18    y = t*(b - s*(a*d));
19 else if ( |a|>=|d| )
20    y = t*(b - a*(s*d));
21 else
22    y = t*(b - d*(s*a));
23 if ( F == true )
24    b = -b;
```

manteniendo un error de redondeo de 1 ulp:

$$\frac{a+ib}{c+id} = \begin{cases} \dfrac{a+b(d/c)}{c+d(d/c)} + i\dfrac{b-a(d/c)}{c+d(d/c)} & \text{si } |d| < |c| \\ \dfrac{a(c/d)+b}{c(c/d)+d} + i\dfrac{b(c/d)-a}{c(c/d)+d} & \text{si } |d| \geq |c|\,. \end{cases}$$

aunque tanto overflow como underflow pueden ocurrir cerca de los límites de los NPF, además de requerir una división extra para el factor d/c o c/d. Esta técnica se ha utilizado desde entonces, incluso para aritmética de precisión arbitraria, como en [171] en 1998.

Los problemas remanentes fueron analizados por Stewart en 1985, en [173], logrando evitar la mayoria, al hacer operaciones cuidadosas. Stewart planteó el algoritmo que se presenta en el código 4.7. El resultado queda en las variables `x` (la parte real) y `y` (la parte imaginaria).

Lo que hace es calcular con mucho cuidado expresiones como $b(d/c)$ asociando de la manera más segura posible. Se logran eliminar mayor cantidad de problemas, pero aún quedan algunos. Más adelante se presentan otras versiónes de división compleja que hacen uso de la norma.

The Art of Scientific Computing.

En el famoso libro "Numerical Recipes in C", [157], se presentan sugerencias generales para la multiplicación, módulo, división y raíz cuadrada de múmeros complejos, aunque mucho menos detalladas y referenciadas que aquí. Al abarcar tanto, [157][5] ha

[5] La versión inicial fue en Fortran y resultó de gran a poyo para programadores inexpertos.

sido criticado por diversos expertos en algunas áreas. De cualquier manera, es un buen compendio de técnicas tanto de matemáticas como de programación.

* * *

Otros ejemplos interesantes pueden consultarse en [161], donde Reid recomenó una forma de implementar rutinas básicas para manejo portable de números de punto flotante, basándose en las definiciones de normativas de IFIP en [61]. Funciones no tan básicas pero igualmente interesantes aparecieron en [22], donde Brown y Feldman proponen parámetros básicos para los NPF, en la forma de funciones básicas, ejemplificando su uso con logaritmo, exponencial y escalamiento de vectores.

Se requiere un volumen a parte para poder presentar una colección que intente ser completa de todas las técnicas que durante décadas los programadores han ideado para reducir problemas numéricos, pero es mejor comprender los fundamentos para poder entenderlas y generar las propias.

4.2. Después de la norma

Comenzando con el 8087 de Intel.

Antes de aprobarse, la norma 754 ya comenzaba a ser utilizada tanto por programadores (los más) como por fabricantes de procesadores y diseñadores de lenguajes.

Si el procesador, lenguaje y compilador cumplen con la norma, entonces cualquier programador principiante o aficionado pudiera desarrollar programas moderadamente complejos y obtener resultados cuya exactitud estará limitada sólo por la APF. Esto sin necesidad de conocer la norma y sin ser un especialista en análisis numérico.

4.2.1. Evaluando si se cumple

Si se desea hacer uso de las bondades de la norma, lo mejor sería poder evaluar si procesador y compilador cumplen con los requisitos mínimos. Los algoritmos pueden ser verificados más facilmente, pero probar que el dispositivo fisico sigue cumpliendo con los requisitos es otra cosa. En los primeros años de su desarrollo, las baterías de prueba y simulaciones fueron la forma de asegurar que lo planteado en papel se cumplía en el procesador y esa metodología sigue siendo fundamental para todos los fabricantes.

Existen propuestas de pruebas formales en las que se busca una demostración matemática de que un programa es correcto. Al realizarse y aprobarse por un ambiente en particular, no se dejan dudas sobre la confianza que se puede tener, pero son más laboriosas. Poco después de la publicación de la norma, Barret publicó [14] donde aplica el lenguaje Z en la verificación de la norma.

La especificación formal es una técnica aún difícil de aplicar en sistemas grandes, pero útil para rutinas y programas pequqños.

Una referencia muy completa es el trabajo de Kern y GreenStreet [115], sobre verificación de diseño de hardware en general. Describen una basta cantidad de técnicas, desde la abstracción y el refinamiento hasta la prueba automática de teoremas, pasando por lógicas temporales, lenguajes regulares y otras tantas cosas, sin olvidarse

Actualmente no es complicado encontrar serias críticas a los códigos que lo acompaãn.

de mostrar sus limitaciones. Las 197 referencias del artículo reflejan el estudio a fondo sobre el tema.

Trabajos sobre pruebas formales de operaciones particulares pueden servir de referencia para iniciar un estudio más específico, como [94] sobre `fma`, [112] sobre la multiplicación, [133] sobre la división o [170] donde se estudia la verificación de programas numéricos en paralelo.

Algunos fabricantes publican los trabajos de verificación formal y pruebas prácticas de los algoritmos empleados en sus procesadores y de los procesadores mismos, comparando las cotas teóricas con las prácticas, como [145] de Intel o [166] de AMD.

Hace muchas décadas ya hay programas para evaluar las características de la APF de las computadoras. Son más prácticas que los trabajos de verificación formal y generalmente los programas pueden servir para varias computadoras, comparando parámetros detectados o resultados de operaciones.

Una de las primeras referencias es el trabajo de Redish y Ward de 1971 [160], en el que plantean que es de esperarse que la siguiente generación de lenguajes tengan algunas facilidades comunes normadas (lo que no ocurrió). Mientras tanto, era necesario ir pensando a futuro en los requisitos que debían exigirse a los lenguajes. Propone algunas llamadas de alto nivel para separar las variables de punto flotante en sus componentes básicos, como en la definición 2.4, de la página 29.

Michael A. Malcolm de la Universidad de Stanford fue de los primeros que desarrollaron programas para la APF como objetivo. Las primeras rutinas las desarrolló en Fortran (véase [131]) y posteriormente se aplicaron en otros programas. Básicamente eran subrutinas que calculaban los parámetros β, p y si se redondea o trunca, que era lo relevante en esa época.

La base β puede ser calculada con el código 4.8.

Listing 4.8: Calculando la base.

```
1  a=1.0;
2  while ( ((a+1)-a)-1 == 0 )
3       a = 2*a;
4  Base = 1.0;
5  while ( ((a+Base)-a)-Base != 0 )
6       Base = Base+1;
```

Teniendo la base, la precisión p se calcula con el código 4.9.

Listing 4.9: Calculando la precisión.

```
1  a=1.0;
2  do {
3       Prec = Prec + 1;
4       a = a*Base;
5       b = a + 1;
6  } while ( b-a == 1 );
```

Un error de programación flotante cometido por Malcolm en su programa fue corregido por Gentleman y Marovich en [71]. Ese error es el típico ejemplo de problemas que a la fecha existen, aunque sin causar daños tan graves. Si los cálculos intermedios se realizaban con una precisón distinta al del formato destino, los parámetros eran

incorrectos. ¿Qué precisión se debe reportar, la interna o la del formato? En una computadora Honeywell 6000, el programa de Malcolm reportó $\beta = 0$, $p = \infty$ y redondeo detectado por truncamiento: todo mal. Reescribiendo en C (el original está en Fortran), las condiciones de la forma

```
if ( a+b != a ) ...
```

debían cambiarse por

```
if ( (a+b)−b > 0 ) ...
```

para corregir el problema y persistía aún en algunas computadoras.

En [126], en 1981, aparece un nuevo procedimiento atribuido a Kahan para calcular la base y la precisión. Estas nuevas formas sugeridas por Kahan para calcular β y p son:

$$\begin{aligned} U &= fl\left(3 \times \left(\frac{4}{3} - 1\right) - 1\right) \\ R &= fl\left(\left(\frac{U}{2} + 1\right) - 1\right) \end{aligned}$$

si $R \neq 0$ se hace $U = R$

$$\begin{aligned} u &= fl\left(3 \times \left(\frac{2}{3} - .5\right) - .5\right) \\ r &= fl\left(\left(\frac{u}{2} + .5\right) - .5\right) \end{aligned}$$

si $r \neq 0$ se hace $u = r$. Ahora ya se pueden calcular

$$\begin{aligned} \beta &= fl\left(\frac{U}{u}\right) \\ p &= \left\lfloor fl\left(\frac{-\log(u)}{\log(\beta)} + .5\right)\right\rfloor. \end{aligned}$$

Mientras se discutía la norma, en 1983 Kahan programó *Paranoia* en Basic, un programa que permite determinar algunos de los parámetros de la APF, como la base, precisión, modo de redondeo, existencia del bit de guardia y operaciones correctamente redondeadas. Ha sido traducido a otros lenguajes y está disponible en NetLib[6]. En 2004 se publicó una versión de paranoia para sistemas empotrados (embedded systems) por Les Hatton en [83]. El mismo año, Hillesland y Lastra presentaron una versión de Paranoia para evaluar los procesadores de las tarjetas gráficas (GPU) en [97].

Entre otras cosas, Paranoia encuentra base y precisión (con dos maneras para comprobar), modo de redondeo, límites de exponentes, valores límites, épsilon de máquina, si se realiza undeflow gradual, evaluación de subexpresiones con precisión extra, el sticky bit y otros más. Adicionalmente, busca anomalías aritméticas en las operaciones elementales.

El autor recomienda ampliamente descargar el código, compilarlo y ejecutarlo. La mejor documentación es la salida misma del programa.

[6] http://www.netlib.org/

Basado en sus trabajos de 1980, Cody escribió el programa *Machar* como parte de la librería para probar funciones elementales ELEFUNT y en 1988 lo reescribió acorde a la norma de IEEE [34] en [34] para reportar parámetros similares a los de paranoia. A diferencia de paranoia, no busca anomalías en operaciones elementales.

En 2002, Nelson Beebe reescribió el programa Machar en Java para usarlo la nueva librería java.lang.Math que extiende las funciones matemáticas para una mejor programación numérica. Con este mismo objetivo surgió el lenguaje Borneo (a propuesta de Kahan y Joseph Darcy) como una extensión de la semántica de punto flotante de Java, para resolver todas las deficiencias numéricas y apegarse a la norma.

A la fecha no existe ninguna implementación de Borneo que el autor conozca.

En 1996, Kahan propuso otra forma de evaluar exclusivamente la precisión en [105]. Lo hace resolviendo una ecuación cudrática con una serie de coeficientes que cada vez requieren una mantisa mayor, lo que propone como medida estándar de precisión.

Los códigos de Paranoia y Machar se consiguen fácilmente en Internet. A parte de estos, otros tantos se han diseñado.

- UCBTEST, que es un conjunto de programas para probar algunos casos difíciles de la norma. Además de Paranoia, incluye UCBmultest, UCBdivtest y UCBsqrttest, que generan casos de prueba problemáticos, en el sentido de producir resultados muy cerca del punto medio de dos NPF vecinos. UCBTest prueba funciones elementales. Para más detalles consultar [180, 181].
- TestFloat es un conjunto de pruebas de John Hauser que utiliza la librería de gran calidad SoftFloat de IEEE que emula con software las operaciones aritméticas para compararlas con las del hardware. Una discrepancia indicará anomalías de algún tipo.
- NAG Floating-Point Validation package, del Numerical Algorithms Group, uno de los grupos de desarrollo de software numérico mas consistentes. Algunos notables investigadores, como Cody, consideran NAG superior en muchos aspectos a IMSL.
- SRTEST es un programa pequeño de Kahan para probar la correctitud de los algoritmos de división de los procesadores. La idea original es detectar errores de división mucho antes de lo que se espera que el uso cotideano logre.
- MPCHECK es un programa escrito por Revol, Pélissier y Zimmermann y disponible en la página web `http://www.loria.fr/~zimmerma/free/.`
- IeeeCC754 verifica que la precisión y los rangos sean los correspondientes a las normas 754 y 854. Está basado en partes en el programa inicial de Jerome Coonen FPTEST. En la página `http://www.win.ua.ac.be/~cant/ieeecc754.html` hay más detalles.

Además de las anteriores, Nelson Beebe ha incluido en su página[7] una colección amplia de software para probar características numéricas, además de notas y ligas a software interesante para todo programador.

Aún no se puede confiar por completo en programar con la norma. De hecho, la situación no ha cambiado mucho desde 1997 cuando el estudio de Dennis Verschaeren y otros [182], reporta qué tanto cumplen la norma diversos ambientes de programación, en el caso de C/C++ y Fortran. Por ello es necesario poder evaluar cada ambiente de desarrollo, en particulares aquellos con los que el programador no está familiarizado.)

En partitular cuando se trata de detalles finos de la APF.

[7] `http://www.math.utah.edu/~beebe/software/ieee/.`

4.2.2. Programación básica

Conciendo la norma, lo más evidente es hacer uso tanto del formato como de sus propiedades. Por ejemplo, si $\mathrm{fl}(x) = m \times \beta^e$, funciones como $\log(x)$ debieran poder simplificarse como

$$\log(m \times \beta^e) = \log(m) + e \log(\beta).$$

Esto es lo obvio, pero hay otras operaciones donde las cosas no son tan agradables, como x^y. En ese caso, qué hacer luego de $(m_1 \times \beta_1^e)^{m \times \beta^e} = m_1^{m \times \beta^e} \times \beta_1^{e \times m \times \beta^e}$ ya no es tan obvio. Es mejor hacer las cosas paso a paso y con cuidado.

Gran cantidad de librerías matemáticas se han desarrollado o en Fortran o en C. Este último tiene la ventaja de disponer de recursos para manipular el bajo nivel con facilidad, tanto direcciones de memoria como los bits de manera directa con operadores nativos, por lo que se usará más comunmente en los ejemplos subsecuentes.

Planteamientos para extender la precisión y garantizar errores relativos menores en las operaciones al estilo de Dekker y Linnainmaa [48,126] siguen siendo útiles, pero ya hay versiones más modernas como la de Bayle [13], programada en Fortran y utilizada por la excelente librería BLAS (ver [125][8]).

Sin la intención de adentrarse en los detalles de aplicaciones muy amplias, reservadas para la segunda parte de este documento, se ejemplifican algunos ejemplos sencillos.

4.2.3. Identificando valores especiales

Primeramente, las comprobaciones de que los números sean finitos y no especiales debiera poder hacerse desde el lenguaje con instrucciones booleanas como isNaN(x), isSubnormal(x), isFinite(x) o isInfinito(x). Incluso debiera existir la función boleana is754() que regrese cierto si el ambiente opera conforme a la norma. Cuando no se dispone de esto, se puede comprobar de otras maneras, como `if ( x!=x )`, que es cierto sólo cuando `x` es NaN.

Ojo con los compiladores que cambian esto a falso.

En C, es posible hacer la conversión de apuntador de doble a entero como en el código 4.10.

Listing 4.10: Conversión de apuntadores de doble a entero en lenguage C.

```
1 int *p;
2 float x;
3 p = (int *)&x;
```

Despues del cast en la línea 3, la variable `p` tiene la dirección en memoria de la variable `x` y la ve como entero, por lo que es posible hacer operaciones como si se tratara de un entero normal. Para saber si `x<0`, basta ver si el bit de signo está prendido:

Listing 4.11: Verificando si el bit de signo está prendido.

```
4 if ( *p & 0x10000000 )
```

[8] En particular el código de la función `BLAS_dsum_x`.

Usar ...	En vez de ...
`if (Fl2Int(f) <= 0 )`	`if ( f<=0.0 )`
`if (Fl2Int(f) > 0 )`	`if ( f>0 )`
`if ( Fl2Int(f) >= 0x404000000 )`	`if ( f>=3.0 )`
`if ( Fl2Int(a-b) < 0x80000000 )`	`if ( a<b )`
`Fl2Int(f) ^= 0x80000000}`	`f = -f`
`Fl2Int(f) &= 0x7fffffff}`	`f = fabs(f)`
`if ( Fl2Int(f) == 0x80000000 )`	`if ( f == -0.0 )`
`if ( Fl2Int(f)|0x7fffffff == 0x7fffffff )`	`if ( isInfnity(f) )`

Tabla 4.2: Algunas instrucciones de punto flotante equivalentes en C. Estas conversiones son más rápidas que las del procesador pero no consideran a los números especiales. Si `f` es ∞ el resultado ser incorrecto.

El único inconveniente es la posibilidad de que el compilador utilizado asigne sólo 2 bytes al entero, en vez de 4 que es lo que se necesita para empalmarse con los 4 bytes del float (precisión simple). La tabla 4.2 muestra otras posibilidades. Supóngase que se ha definido la macro

```
#define Fl2Int(f) (* ((int *)(&f)))
```

y que `f` es una variable `float` (precisión simple).

El programador debe cuidar el uso de los ejemplos de la tabla y otros esquemas similares dependiendo del tipo de arquitectura. Las computadoras BigEndian colocan las posiciones de memoria en orden decreciente de la significancia de bytes, en sentido contrario de las LittleEndian. Normalmente las arquitecturas x86 son LittleEndian y otras como RS6000 son BigEndian. Esto significa que un código portable debiera verificar primero en qué arquitectura se compila para seleccionar el orden adecuado de las referencias a memoria.

Por ejemplo, si una variable entera de dos bytes toma el valor en hexadecimal 0xff00, ocho unos seguidos de ocho ceros, en una máquina LittleEndian las celdas de memoria estarán dispuestas primero el 255 y luego el 0, mientras que en una BigEndian será alrevés. Un flotante `f` con cuatro bytes tendrá representación $b_1b_2b_3b_4$ en una LittleEndian y $b_4b_3b_2b_1$ en la otra.

Por otra parte, las operaciones de la tabla 4.2 son más rápidas que las del procesador, pero no consideran los valores especiales. Si `f` es NaN, el resultado puede ser incierto (más bien inválido). El procesador considera todos los casos y regresa el valor correspondiente, que es lo correcto. Las versiones de la tabla se deben usar si se está seguro de que los valores no son especiales.

4.2.4. Decodificando NPF

Otra técnica que se emplea con frecuencia es utilizar las uniones de C, pues como los datos comparten memoria, se pueden encimar flotantes con enteros o caracteres para tener acceso a los bits del flotante. Por ejemplo, usando un arreglo de tantos bytes como el tamaño del tipo de PF:

Listing 4.12: Unión para decodificar variables de punto flotante.

```
union {
   unsigned char Byte[4];
   float f;
} U;
```

Si a `f` se le da un valor, se pueden leer los bits del campo `Byte` para decodificar y ver el formato. Como el flotante ocupa 1 bit de sig, 8 de exponente y 23 de mantisa, el exponente tiene 7 bits en Byte[3] y uno (el más alto) en Byte[2]. Por ejemplo el bit de signo se revisa con

```
   if ( U.Byte[3] & 0x80 )
```

El valor del exponente puede obtenerse con

```
   e = ( ( U.Byte[3] & 0x7f ) << 8 ) |
         ( U.Byte[2] & 0x80 )
```

El valor como entero de la mantisa (sin el bit implícito) puede extraerse con

```
   e = ( ( U.Byte[2] & 0x7f ) << 16 ) |
           (U.Byte[1] <<8 ) | U.Byte[0]
```

Si los enteros miden 4 bytes (la mayoría lo son), entonces es sencillo con

Listing 4.13: Otra unión para decodificar variables de punto flotante.

```
union {
   unsigned int B;
   float f;
}  U;
```

De nuevo, el exponente se obtiene con `U.B & 0x7f800000}`. Este método es más compacto, siendo deseable que el lector pueda usar ambas posibilidades. En caso de que `f` sea de doble precisión, puede utilizarse el primer método (código 4.12), pero por fortuna algunos compiladores incluyen enteros de 8 bytes (tipo `long long`).

Usar campos de bits también puede resultar atractivo:

Listing 4.14: Campos de bits para decodificar variables de punto flotante.

```
union {
   struct {
      unsigned Signo:1,
               Exp:8,
               Mantisa:23;
   } B;
   float f;
} U;
```

La forma de acceder a los campos es más sencilla, pero con el inconveniente de la poca portabilidad de los campos de bits, comparada con las dos técnicas anteriores. Para extraer exponente y mantisa se escribe el código adcional.

Llamada	Efecto
`n=ceil(x)`	redondea hacia arriba (al menor entero mayor que `x`)
`n=floor(x)`	redondea hacia abajo (el mayor entero menor que `x`)
`r=fmod(x,y)`	el residuo de `x/y`
`n=modf(x, &f)`	separa la parte entera[a] de fracción normalizada de `x=n+f`
`m=frexp(x,&e)`	extrae la mantisa y el exponente de `x`
`x=ldexp(m,e)`	calcula $\texttt{m}\times 2^{e}$
`e=logb(x)`	extrae el exponente de `x`
`x=scalb(m,e)`	calcula $\texttt{m}\times\beta^{e}$ [b]

[a] La parte entera se declara como flotante
[b] Si $\beta = 2$ es equivalente a `ldexp`.

Tabla 4.3: Llamadas en C de la norma de punto flotante

```
9  U.f=4.17;
10 e = U.B.Exp;
11 m = U.B.Mantisa;
```

En lenguajes que no tengan acceso a los bits de una variable mediante operadores nativos, siempre es posible hacer las mismas operaciones lógicas a partir de operaciones aritméticas. Obsérvese que se decodifican los bits del formato indicado por la norma, sin tener acceso a todos lis bits del registro flotante.

Para apoyar el uso de la norma, lenguajes como C disponen de rutinas de alto nivel que facilitan las cosas. Por ejemplo en C, algunas llamadas útiles se presentan en la tabla 4.3, la mayoría incluidas en el apartado de operaciones de la norma.

Partiendo de las llamadas de la tabla 4.3, es más sencillo hacer uso de la norma, ya que los códigos son más portables y legibles. Funciones como `scalb` y otras, que ya habían sido sugeridas en [22], resultan de gran utilidad.

Librerías comerciales y gratuitas incluyen funciones apegadas a la norma para el caso de que el compilador empleado no las provea. Por ejemplo, la librería científica gsl del proyecto GNU[9] incluye, entre muchas otras cosas, rutinas para la programación eficiente de punto flotante, en particular para identificar valores especiales, determinar y establecer precisión y modo de redondeo. La libraría lee la variable de ambiente GSL_IEEE_MODE, por lo que es fácil modificar los modos sin cambiar nada en el código fuente.

GNU Scientific Library.

[9] El autor recomienda ampliamente al lector descargar la librería de internet y comenzar a utilizarla. Esta escrita por completo en ANSI C, por lo que no debiera haber prblema de usarla con ningún buen compilador. Claro, si se utiliza con el compilador gcc del proyecto GNU (Dev-C, por ejemplo) hay mayores garantías.

4.2.5. Levantando banderas

Levantar banderas tiene el inconveniente de que en cada sistema la función puede tener su propio nombre, como se ilustra en la tabla 3.10, página 73. La solución normal es utilizar directivas de preprocesamiento para que el compilador seleccione los códigos correctos al generar el programa ejecutable.

Por ejemplo, si se está calculando $1/n!$, con $n > 170$ se ocasiona overflow en precisión doble, por lo que el resultado correcto debe ser 0 por las propiedades de propagación de los valores especiales. Supóngase por simplicidad que la variable `UnderflowFlag` ya tiene el valor correcto.

Listing 4.15: Levantando banderas y determinando la arquitectura en tiempo de compilación.

```
1 if ( n>170 ) {
2    #ifdef 80x86
3     _status87(UnderflowFlag);
4    #else /* Unix por default */
5     fpsetsticky(UnderflowFlag);
6    #endif
7    return 0.0;
8 }
```

El código 4.15 deberá crecer tanto como arquitecturas o plataformas se desee considerar, causando problemas de legibilidad. En muchas circunstancias esto es inevitable, pero hay ocasiones en las que el problema tiene una solución más elegante.

Sencilla y elegante, considerando que a la fecha aún hay más de 10 arquitecturas distintas.

Una técnica sencilla es hacer una operación que asegure que la bandera de excepción correcta será levantada, independientemente de la plataforma. En el ejemplo anterior, en vez de regresar cero (que no levanta banderas) se puede regresar el cuadrado del valor más pequeño posible, lo que ocasionará un underflow con toda seguridad, evitando levantar la bandera desde el código.

Listing 4.16: Levantando banderas sin conocer la arquitectura.

```
1 if ( n>170 )
2    return (1/HUGE_VAL)*(1/HUGE_VAL);
```

Con el código 4.16 se asegura que el procesador se encargará de levantar las banderas de inexacto y underflow, lanzando la señal correspondiente. `HUGE_VAL` está definido en `math.h`.

El ejemplo es minimalista si se considera que con $n = 171$ a 178 se obtienen valores subnormales correctos[10], además de que a la variable externa `errno`, que provee el sistema, se le debe asignar el valor que corresponda, pero el ejemplo es claro de esta manera.

4.2.6. Más sobre la Hipotenusa

Retomando el caso del cálculo de la hipotenusa $c = \sqrt{a^2 + b^2}$, antes de la norma se aplicaba la técnica aritmética de escalar para evitar overflow. Con la norma, primero

[10] Lo mejor sería precalcularlos y guardarlos en un arreglo, pues sólo son 7 valores.

es conveniente revisar lo valores especiales, pues podría ocurrir que los valores de a o b son NaN o ∞ y regresar el valor correspondiente.

Como de costumbre, la experiencia anterior a la norma es útil. Los problemas de overflow y underflow fueron resueltos con los esquemas ya presentados. Sólo falta remediar la garantía de tener el último bit correctamente redondeado.

La raíz cuadrada se incluye entre las operaciones elementales de la norma, por lo que su error relativo es menor que μ, lo que garantiza un error menor que .5 ulp. Falta garantizar lo mismo para $a^2 + b^2$. Si $a \geq b$, puede medirse

$$h = ((a^2 + b^2) - a^2) - b^2$$

para detectar el error h. Entonces el cálculo será realmente

$$c = \sqrt{a^2 + b^2 + h}.$$

Si el resultado final c se eleva al cuadrado, la diferencia

$$g = (a^2 + b^2) - c^2$$

incluye el error de redondeo de la raíz cuadrada y es cota superior de h. El fordward error es mayor que el backward error por pocas μ. Esto significa que el cálculo es estable.

Para garantizar la máxima precisión, primero es necesario calcular $a^2 + b^2$ con la máxima precisión posible. Una primera idea es $a^2 + b^2$ tiene 3 redondeos que pueden reducirse a dos. Se pude usar la operación fma, basta decidir por $\mathrm{fl}(\mathrm{fl}(a^2) + b \times b)$ o $\mathrm{fl}(a \times a + \mathrm{fl}(b^2))$, es decir, `fma(a*a,b,b)` o `fma(b*b,a,a)`. La elección depende de cuál es el valor más grande. Si $a < b$ la primera es la mejor opción. Ahora sólo hay tres redondeos, el cuadrado, fma y la raíz.

Recuérdese que esta no es una decisión que el compilador deba tomar.

Una manera de garantizar la hipotenusa con el último bit correctamente redondeado se logra si se aplica la técnica de Dekker 4.2 tanto a a como a b. Supóngase que $a \geq b$. Considerando variables de precisión doble, se tiene

$$\begin{aligned} a &= a_h + a_l \\ b &= b_h + b_l \end{aligned}$$

y $a_h = a$ con los 32 bits más bajos en 0, similarmente con b_h. Como

$$\begin{aligned} (a_h + a_l)^2 &= a_h^2 + 2a_h a_l + a_l^2 \\ (b_h + b_l)^2 &= b_h^2 + 2b_h b_l + b_l^2 \end{aligned}$$

entonces, hay que analizar las posibilidades. Al elevarse al cuadrado, las cantidades requieren una mantisa mayor, por lo que la suma dejará afuera los bits más bajos, que están en

$$\begin{aligned} a^2 + b^2 &= a_h^2 + (b^2 + 2a_h a_l + a_l^2) \\ &= a_h^2 + (b^2 + x_l(x + x_h)) \end{aligned}$$

es decir, es más conveniente sumar los términos más pequeños primero y dejar a lo último a a_h^2. Esta técnica es la que emplean en la librería de SUN.

Listing 4.17: La hipotenusa en Fortran 2003.

```
1  real function hipot(x, y)
2  use, Intrinsic :: ieee_arithmetic
3  real x,y
4  real scaled_x. scaled_y, scaled_result
5  logical, dimension(2) :: flags
6  type (ieee_flag_type), parameter, dimension(2) :: &
7        out_of_range = (/ ieee_overflow, ieee_undeflow /)
8  intrinsic sqrt, abs, exponent, max, digits, scale
9    hipot = sqrt( x**2 + y**2 )
10   call ieee_get_flag(out_of_range,flags)
11   if ( any(flags) ) then
12     call ieee_set_flag(out_of_range,.false.)
13     if ( x==0.0 .OR. x==0.0 ) then
14       hipot = abs(x) + abs(y)
15     else if ( 2*abs(exponent(x)-exponent(y)) >
16                                    digits(x)+1 ) then
17       hipot = max( abs(x), abs(y) )
18     else
19       scaled_x = scale( x, -exponent(x) )
20       scaled_y = scale( y, -exponent(x) )
21       scaled_result = sqrt( scaled_x**2 + scaled_y**2 )
22       hipot = scale( scaled_result, exponent(x) )
23     end if
24   end if
25 end function hipot
```

Si $a > 2b$ se utiliza la fórmula $a_h^2 + (b^2 + (a_l \times (a + a_h)))$. Debe ser claro que $a_l = a - a_h$.

En caso de que $a \leq 2b$ se usa $t_h b_h + ((a-b)(a-b) + (t_h b_2 + t_l b_h))$, donde $t_h = 2x$ con los 32 bits más bajos en cero y $t_l = 2x - t_h$.

En su momento, Dekker no tuvo manera de hacerlo con los lenguajes de alto nivel de su época.

En vez de realizar las operaciones que indica Dekker, se pueden manipular directamente los bits de la representación numérica para no tener errores de redondeo y alcanzar una alta eficiencia. Con este procedimiento se obtiene un resultado con sólo 2 redondeos, uno por $a^2 + b^2$ y otro por la raíz.

Otro esquema es el empleado por algunas librerías que usan precisión arbitraria, como la MPFR en Fortran (véase [63]).

En los documentos sobre Fortran 2003, la misma introducción incluye el código de una función que calcula la hipotenusa, como ejemplo del uso de las facilidades, reproducido con cambios menores en el código 4.17.

Primero se intenta un cálculo directo y rápido (línea 9), verificando que ninguna bandera se levantó (líneas 10 y 11). De haber ocurrido, se repite el cálculo de una manera más lenta pero segura (líneas 13–22). Este es un ejemplo poco común en el sentido de que, en los cálculos numéricos, la mayoría de los casos, las situaciones excepcionales son más comunes de lo que se quisiera. Por ello lo normal sería poner el caso directo (como la línea 9 del código 4.17) al final, luego del manejo de los casos especiales.

Caso	Resultado
$1023 < n$	∞ (overflow)
$-1022 <= n <= 1023$	Número normal
$-1074 <= n < -1022$	Número subnormal
$n < -1074$	Cero (underflow)

Tabla 4.4: Esto cubre todas las posibilidades, asegurando una respuesta confiable de la función. Como el exponente es un entero, no hay peligro de recibirl algún valor especial.

4.2.7. La función 2^n

Por supuesto que lo más práctico es utilizar la función `ldexp(1,n)` pero es ilustrativo ver cómo puede implementarse. Supóngase el caso de la precisión doble. Lo más evidente en este caso sería un ciclo que multiplicara $2 \times 2 \times 2 \times \cdots \times 2$, $n-1$ veces pero sería una solución ineficiente, ya que

$$2^{34} = 2^2((((2^2)^2)^2)^2)^2$$

requiere sólo 6 productos, a diferencia de 33 multiplicaciones. La codificación del formato flotante de las potencias de 2 facilita aún más las cosas, excepto para $\beta = 10$, por supuesto.

Usando la idea del código 4.12 en la página 94, para superponer en memoria NPF con bytes, para la doble precisión es como sigue.

Listing 4.18: Calculando 2^n.

```
1  union {
2     unsigned char b[8];
3     double d;
4  } u;
```

Con esto los 8 bytes del arreglo `b` comparten los 8 mismos bytes de la variable `d`. Ahora se deben considerar todos los casos posibles, que se presentan en la tabla 4.4.

Lo más conveniente desde el principio es dejar en ceros todos los bits del valor que se va a regresar para proceder a prender los que sean necesarios.

```
5  u.d = 0;  /* Para poner a cero todos los bits */
```

El primero y último casos (overflow y underflow) requieren una señal y codificar el resultado, que no es difícil. Para el overflow se requiere construir la representación del ∞, prendiendo todos los bits del exponente y apagando el resto.

```
6  if ( n > 1023 ) {
7     u.b[7] = 0x7f;  /* Respetando el bit del signo */
8     u.b[6] = 0xf0;
9     raise(SIGFPE);
10    errno = ERANGE;
11    return u.d;
12 }
```

O lo más sencillo si se puede confiar de que la plataforma respeta la norma de PF

```
if ( n > 1023 ) {
   u.d = HUGE_VAL*HUGE_VAL;
   return u.d;
}
```

Para el underflow hay que regresar 0.0, por lo que es suficiente con

```
13 if ( n < -1074 ) {
14    raise(SIGFPE);
15    return u.d;
16 }
```

Considerando que para los números normales el formato de las potencias de 2 sólo tienen un bit prendido en la mantisa (el implícito), cambiando sólo el exponente, habría que modificar los bits para obtener de manera directa nuestro resultado.

Como al exponente del formato de doble precisión se le resta 1023 al codificarse, primero se debe obtener el valor final del exponente en la representación. Si se va a calcular 2^n, en la codificación se tendrá $n+1023$ como exponente, por lo que el número que se guardará en los bits que corresponden al exponente es $e = n + 1023$.

En doble precisión, los bits que ocupa el exponente son los 7 menos significativos de u.b[7] y los 4 más significativos de u.b[6], por lo que el caso normal queda

```
17 Exp = n + 1023;
18 u.b[7] = Exp>>4;
19 u.b[6] = ( Exp&0x0f ) << 4;
20 return u.d;
```

El caso subnormal debe tratarse distinto. El exponente será cero y en la mantisa habrá un bit prendido, por lo que se deben calcular el byte y el bit que le corresponden, a partir del valor del exponente, calculado como `Exp + bits de la mantisa`, 52 en este caso. Sean `Bit` y `Byte` dos variables enteras.

```
21 Pos = n+1023+52;
22 Byte = Pos/8;
23 Bit = Pos%8;
24 if ( Bit )
25   Bit = 1<<(Bit-1);
26 else {
27   Bit = 1<<7;
28   Byte -= 1;
29 }
30 u.b[ Byte ] |= Bit;
31 return u.d;
```

El valor de `Bit` debe ajustarse para tome un valor de 1 a 8 y no de 0 a 7. En el caso de `Bit==0` se debe ajustar también hacia el byte anterior. La penúltima línea prende el bit adecuado.

Si se busca mejor desempeño, el código puede reescribirse, pero de esta manera permite ver un manejo más evidente de uso del formato. Las operaciones más costosas son la división y el módulo, que en ensamblador se reducirían a una sola división pues el módulo se obtiene del acarreo. Por ello algunos compiladores codifican `ldexp` en ensamblador.

4.2.8. Más de la división compleja

Recordando: se calcula $x + iy = (a + ib)/(c + id)$.

Luego del escalamiento de Smith (pag. 86) para evitar overflow y los cuidados de Stewart para evitar algunos underflow y overflows remanentes, finalmente, en 2000 casi todos los problemas fueron eliminados en la librería BLAS, véase Li en [125], escalando con mayor cuidado que el procedimiento de Smith, obteniendo un error relativo de apenas unas cuantas ulp, a costa de más cómputo. Escala tanto el numerador como el denominador.

Este método considera las cotas de overflow y underflow de la norma. Sean R y r como en (4.4), página 84. Primero calcula $m = \text{máx}(|a|, |b|)$, $n = \text{máx}(|c|, |d|)$.

En caso de que $m > R/16$, se escala con limpieza: $a = a/16$ y $b = b/16$ sin tocar la mantisa, sólo el exponente. En caso de $m < k \times rN/\epsilon$, ϵ el épsilon de máquina y k un entero pequeño potencia de 2 puede ser de nuevo 16), se escala $a = a \times k/\epsilon^2$ y $b = b \times k/\epsilon^2$.

Se repite el proceso de escala con el otro número complejo $c + id$. Ahora, los 4 coeficientes están en el intervalo $(k \times r/\epsilon, R/16)$. Se aplica el método de Smith y se resultado se escala con las operaciones contrarias.

Uno de los mejores procedimientos, que elimina todos los problemas además de considerar los números especiales de la definición (2.4), página 29 es el de Kahan en [102] de 1987, donde presubstituye con software a falta de soporte del lenguaje. Se basa en la fórmula

Anterior incluso al de [125].

$$\begin{aligned} x + iy &= \frac{a + ib}{c + id} \\ &= (a + ib)\frac{c - id}{c^2 + d^2}. \end{aligned} \tag{4.5}$$

El pseudocódigo de la idea de Kahan se presenta en el código 4.19. Claro que habría que confiar en que el producto de complejos en al antepenúltima línea ocurre como es de esperarse. La llamada `copysign(x,y)` está especificada en la norma y copia a `x` el signo de `y`. Las banderas se restauran luego de la llamada a la multiplicación para continuar con el mismo contexto numérico previo, evitando cambios producidos por la rutina `Multiplicar`. La rutina no produce overflows ni underflows innecesarios y preserva la identidad

$$\left|\frac{a + ib}{c + id}\right| = \frac{|a + ib|}{|c + id|}$$

incluso en los casos ∞ y 0. El caso de los subnormales queda cubierto automáticamente. Este procedimiento y otros tantos son los que emplean las calculadoras HP, donde Kahan trabajó como consultor.

El problema de la anterior rutina es que es lenta, si se piensa utilizar en gran demanda, como grandes matrices con entradas complejas. En su favor habla el hecho de que la división se realiza poco, pero sí hay ocasiones donde su uso es inevitable.

El costo de tantas llamadas a la función `scalb`.

Ante esto, Priest diseñó un método publicado en 2004 en [159] que se basa también en la fórmula (4.5) usando la idea básica de

$$\begin{aligned} r &= 1/(c^2 + d^2) \\ x &= (ac + bd)/r \\ y &= (bc - ad)/r \end{aligned}$$

Listing 4.19: Método de Kahan para división compleja

```
1  Verificar que a,b,c,d no sean valores especiales
2  Salvar las banderas
3  L = logb(MaxDouble)/2-1
4  h = logb(max{|c|,|d|})

6  K = L - entero(h+.5)
7  c = scalb(d,K)
8  d = scalb(d,K)

10 J = L - entero(h+.5)
11 a = scalb(a,J)
12 b = scalb(b,J)
13 d = x*x + y*y
14 if (d==Infinito or d==0 ) {
15     if ( |x|==d ) x = copysign(1,x)
16     if ( y != 0 ) y = copysign(1,y)
17 }
18 m+in = Multiplicar(a+ib,c+id)
19 Restaurar las banderas
20 return ( scalb(m/d, K-J) + i scalb(n/d,K-J) )
```

Es claro que los primeros problemas pueden surgir al calcular `r`, por lo que se necesitará escalar adecuadamente con algún factor s y calcular $r = (sc)^2 + (sd)^2$. Lo único que se requiere es que máx$(|sc|, |sd|)$ esté suficientemente cerca de 1 para calcular r y su recíproco con seguridad. El detalle fino del método de Priest radica en la selección de s y el orden de evaluación que se selecciona.

Priest cuidó todas las posibilidades, garantizando que sólo ocurre overflow o underflow cuando es realmente inevitable aritméticamente.

Una versión no completa se presenta en el código 4.20, donde sólo falta manejar los casos especiales de valores ∞ y NaN, pero es de esperarse que no sea problema para el lector extenderla. Revísese [159] para el código completo. Se supone que los enteros son de 4 bytes y que existe el tipo `long long`.

Este código se ejecuta con precisión y velocidad, con cotas casi ideales de exactitud.

Aunque más lenta, vale la pena por sencilla.

Otra idea interesante y sencilla, aplicable a muchas otras situaciones, es la de Jim Thomas y Fred Tydeman, que usaron el control de banderas de excepción de C99. Primero guardan y limpian las banderas de excepción. Calculan el divisor $c^2 + d^2$ de manera directa y revisan de inmediato las banderas. Estas indican qué ocurrió y de ser necesario se vuelve a hacer el cálculo reescalando, siempre con potencias de 2.

Comenzando con los compiladores de Microsoft.

Esta técnica no puede emplearse en lenguajes donde no se puedan leer las banderas de excepción indicadas por la norma. En ese caso es inevitable buscar un buen factor de escala previo al cálculo.

Ejemplos más extensos se presentan en la segunda parte de este texto.

Listing 4.20: Método de Priest para división compleja

```
1  void DivCompleja(double *v, const double *z,
2                   const double *w)
3  {
4    union {
5      long long int i;
6      double d;
7    } aa, bb, cc, dd, ss;
8    double a, b, c, d, t;
9    int ha, hb, hc, hd, hz, hw, hs;
10   /* Componentes de z y w */
11   a = z[0];
12   b = z[1];
13   c = w[0];
14   d = w[1];
15   /* Extraer los 32 bits mas altos para
16                        estimar |z| and |w| */
17   aa.d = a;
18   bb.d = b;
19   ha = (aa.i >> 32) & 0x7fffffff;
20   hb = (bb.i >> 32) & 0x7fffffff;
21   hz = (ha > hb)? ha : hb;
22   cc.d = c;
23   dd.d = d;
24   hc = (cc.i >> 32) & 0x7fffffff;
25   hd = (dd.i >> 32) & 0x7fffffff;
26   hw = (hc > hd)? hc : hd;

28   /* |z| < 2^-909 and 2^-215 <= |w| < 2^114 */
29   if (hz < 0x07200000 && hw >= 0x32800000 &&
30       hw < 0x47100000)
31     hs = (((0x47100000 - hw) >> 1) & 0xfff00000) +
32                                          0x3ff00000;
33   else
34     hs = (((hw >> 2) - hw) + 0x6fd7ffff) & 0xfff00000;
35   ss.i = (long long int) hs << 32;
36   /* Se escalan c y d, y se calcula el cociente */
37   c *= ss.d;
38   d *= ss.d;
39   t = 1.0 / (c * c + d * d);
40   c *= ss.d;
41   d *= ss.d;
42   v[0] = (a * c + b * d) * t;
43   v[1] = (b * c - a * d) * t;
44 }
```

Parte II

Aplicaciones de la Programación Numérica

Introducción a las aplicaciones

Presentar todas las aplicaciones importantes de programación numérica es imposible a menos que se escriba una enciclopedia sobre el tema, donde se reserve uno o varios volúmenes para cada aplicación o familia de aplicaciones. Lo que sí es posible es mostrar con cierto nivel de detalle algunas que permitan ejemplificar la utilidad de la base teórica y práctica de la aritmética de punto flotante y su norma oficial con su correspondiente manejo de excepciones.

Los tres temas se han seleccionado por la sencillez de su planteamiento, eficiente ilustración de la teoría, utilidad práctica y diversidad. Aunque se espera que esta segunda parte sea autocontenida, es necesario que el lector esté familiarizado con la mayoría de los conceptos matemáticos y de computación que se utilizarán, como programación y estructuras de datos, además de álgebra y cálculo, ya utilizados en la primera parte.

Aunque no se presentan todos los detalles posibles en los tres tópicos, si se analizan los suficientes para orientar a estudiantes y programadores profesionales sobre maneras diversas de resolver problemas básicos, que es de gran utilidad para resolver problemas prácticos.

Los coódigos que se presentan son de carácter ilustrativo y solo en algunos casos se incluyen versiones completas y eficientes de los algoritmos. Detalles como asegurar la portabilidad hacen crecer indeseablemente los códigos, por lo que sólo se indican versiones con ANSI C, salvo unas cuantas excepciones, al igual que en la primera parte.

Primeramente se caracterizan los procesos de suma de NPF en términos de la propagación de errores de redondeo. Sumar es un proceso sencillo que se realiza con eficiencia en los procesadores modernos, pero cuando la cantidad de sumandos crece, la suma de errores de redondeo puede ser mayor de lo que puede creer el programador promedio. Por ello se han desarrollado diversas maneras de reducir este fenómeno.

En segundo lugar se presenta un estudio sobre diversas técnicas para aproximar las raíces de ecuaciones no lineales. Este es un problema que desde hace muchos siglos ha causado interés y alrededor del cual hay gran catidad de teoría y métodos. Se conocen cientos de algoritmos y métodos iterativos distintos y cada año aparecen novedades en la literatura especializada, aunque en los cursos de Métodos o Análisis Numerico, siguen estudiándose unos pocos (y los mismos, que se cuentan con la mano).

Por último se aborda el tema de la programación de funciones elementales, donde se presentan las técnicas generales que se aplican y el estudio de el caso particular de

la función exponencial. Esta es una de las funciones que la norma de IEEE [39] define dentro de las que deben estar disponibles en todo ambiente de programación de punto flotante correctamente redondeadas. Las técnicas empleadas en e^x pueden aplicarse a otras funciones y rutinas en general, por lo que se espera que el lector encuentre útiles algunas de las ideas.

En cada tema se indican referencias suficientes para los interesados en estudiar más a fondo cada uno. La bibliografía es amplia en los tres problemas, principalmete en la solución de ecuaciones no lineales, donde la exposición es notablemente mayor pues el problema es mucho más antiguo.

Quedan pendientes temas tan importantes como solución de sistemas de ecuaciones lineales, problemas de eigenvectores, optimización, integración, ecuaciones diferenciales o elemento finito, pero había que tomar una decisión[11]. El autor espera que la selección presente sea adecuada para dar un panorama general, sensibilizar e informar al programador promedio sobre las posibilidades y limitaciones de la Programación Numérica.

[11] Textos clásicos como [25, 29, 72, 116] pueden servir como buena introducción.

Capítulo 5

Sumatorias

La suma una de las operaciones elementales definidas para la aritmética de punto flotante. Su error relativo está acotado por la unidad de redondeo μ:

$$fl(a+b) = (a+b)(1+\delta), \quad |\delta| < \mu. \tag{5.1}$$

Como debe recordarse, ya Dekker mostró que la suma de dos números puede calcularse de manera exacta si se utilizan dos variables como en la ecuación (4.2). Si a y b son elementos de $\mathbb{F}$ y la suma (5.1) es $s = fl(a+b)$, entonces el error

$$e = (a+b) - s$$

sirve para representar la suma exacta[1], también conocida como suma compensada:

$$a + b = s + e. \tag{5.2}$$

Como es importante comprender bien esto, se ilustra con la figura 5.1, en donde $t - b = e$. Esta idea fue descubierta desde 1951, cuando se aplicó en problemas ecuaciones diferenciales, área donde la estabilidad siempre es crítica.

Si δ cumple $s = a + b + \delta$ entonces $|\delta| \leq \mu\text{ufp}(a+b) \leq \mu\text{ufp}(s) \leq \mu|s|$ es un refinamiento de (5.1).

Como el error $e = ((a+b) - a) - b$ es algebraicamente cero, un compilador que simplifique estará cometiendo un serio error de punto flotante.

En [117], Knuth da un algoritmo que calcula la suma $a + b$ de otra forma, sin importar cuál es mayor:

$$x = fl(a+b) \tag{5.3}$$
$$z = fl(x-a) \tag{5.4}$$
$$y = fl((a-(x-z)) + (b-z)) \tag{5.5}$$

y $a + b = x + y$, válido incluso en presencia de underflow. Es preferible la técnica de Dekker pues solo requiere 3 operaciones, a diferencia de las 6 de Knuth.

[1] Esto es válido si el modo de redondeo es al más cercano.

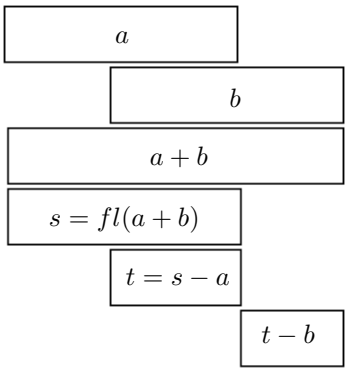


Figura 5.1: Suma compensada. Se supone que las variables a y b han sido normalizadas, utilizan toda su mantisa y que $a > b$, por lo que los bits menos significativos de b se pierden.

5.1. Suma de arreglos o conjuntos de números

Hasta raro es encontrar programas grandes donde no se requiera hacer alguna suma.

Lo que se estudia en este capítulo son los errores de redondeo acumulados en la suma de un conjunto de números. Es algo que de manera rutinaria se incluye en casi todo tipo de programas, al calcular un promedio, resultados finales de un torneo, desempeño total de un evento y muchos más. Lo ejemplos están al alcance de la mano.

Aún cuando no se requiera la suma como valor final, sí se requiere como cálculo intermedio en muchas situaciones, como en graficación, integración, operaciones con matrices, nóminas, simulaciones y una lista interminable.

En la práctica, el conjunto que se suma siempre es finito pero no necesariamente determinado. En el caso de un arreglo se sabe por anticipado cuántos elementos son, pero no en el caso de una serie (como un polinomio de Taylor), donde se requiere sumar hasta cumplir con cierto criterio. Incluso es posible que se tengan que sumar los elementos de un arreglo hasta que se cumpla alguna condición.

En escencia el problema por resolver se plantea como

$$S = \sum_{i=1}^{n} a_i \tag{5.6}$$

donde cada $a_i \in \mathbb{F}$ es un NPF. Como cada suma puede acarrear un error de redondeo por (5.1), se espera que el resultado final exacto S difiera de $fl(S)$, pues se usan formatos y operaciones nativas de los procesadores modernos y no multiprecisión emulada en software.

Los problemas de redondeo en la suma de número fueron reconocidos por Wilkinson desde los inicios del cómputo numérico, como comenta en [187]. Wilkinson demostró que la suma de n números similares en magnitud con x puede tener un error de hasta $n^2 10^{-p}$, donde p es la cantidad de posiciones en la mantisa, como de costumbre. Ese es el error máximo, pero el error más probable fue calculado en 1996 por Mikov en [135] como $n^{1.5} 10^{-p}$.

El problema resultó no ser tan trivial como debió parecer al principio, por lo que

aún a principios del siglo XXI se publican trabajos presentando nuevas formas de reducir los problemas de imprecisión al efectuar sumas.

Hay dos recursos básicos que se deben considerar al diseñar casi todo algoritmo: el tiempo y el espacio. Adicionalmente, aquí interesa lo eficiente del algoritmo, medido en términos de su precisión, exactitud, estabilidad y condicionamiento, conceptos todos definidos en la primera parte. La suma es un proceso bien condicionado pues cambios pequeños en los datos de entrada deben producir cambios pequeños en la suma final, aunque sí hay casos patológicos donde la suma da resultados muy alejados del resultado correcto.

El número de condición de la suma se define como

$$\text{cond}(A) = \frac{\sum_{i=1}^{n} | a_i |}{\left| \sum_{i=1}^{n} a_i \right|} \tag{5.7}$$

y una cota para algunos algoritmos es $\mu^2 \, \text{cond}(A)$, aunque un buen algoritmo debiera no depender de $\text{cond}(A)$.

La estabilidad, precisión y exactitud dependen del procedimiento empleado para sumar los datos y es lo que se analizará en las siguientes secciones, en las que se presenta diversos algoritmos de suma, desde el método más evidente de ir sumando los números en el orden que se presentan hasta la versión cuya cota teórica garantiza la máxima exactitud posible, bajo ciertos supuestos del ambiente de programación. Lo que se intenta es reducir el error absoluto final, representado por

$$e_S = \left| fl(\sum_{i=1}^{n} a_i) - \sum_{i=1}^{n} a_i \right|. \tag{5.8}$$

Si $n\mu \leq 1$ y $\text{máx}\{|a_i|\} \leq T$ entonces

$$e_S \leq \frac{n(n-1)}{2} \mu T. \tag{5.9}$$

que es una cota para (5.8) cualquiera que sea el orden en el que se sumen las a_i e incluso cuando nT causa overflow.

Cuando un programador utiliza funciones de alguna librería gratuita o comercial, usa objetos de una clase programada por otros o se integra soluciones pegando diversas rutinas en general, hay una pregunta importante: ¿qué tanta precisión y exactitud se requiere en el sistema que se está desarrollando? La respuesta pudiera indicar que el error relativo tolerable es suficientemente grande como para no preocuparse por la minucia numérica. Este caso se tiene al promediar edades de personas o calificaciones o procesos gráficos.

Pero si la respuesta es que se requiere la máxima precisión posible, como el caso aplicaciones científicas, simulaciones por computadora, series de tiempo, amortización a largo plazo, o sistemas financieros donde los redondeos pueden ser perjudiciales, ¿qué garantía se tiene del desempeño si se desconoce cómo se realiza algo tan sencillo como la suma de arreglos? Un desarrollador completo debe poder enfrentar este caso sin problemas.

En ambas preguntas se requiere conocer la cantidad de elementos que se van a procesar. Un grupo de 30 esudiantes o 500 derechohabientes de una clínica son poco trabajo. Pero el inventario de una gran empresa, estadísticas de encuestas de salida durante elecciones, muestreos automatizados durante una simulación a largo plazo, donde puede haber cientos de miles de elementos o hasta millones, requiere de un procesamiento más cuidadoso de la información.

Casi cada nuevo artículo sobre procesos de suma incluye un estudio sobre los métodos anteriores. A la fecha siguen siendo pocos comparados con las raíces de ecuaciones, por lo que es posible considerarlos casi todos. Ya Higham dedicó un capítulo de su libro [88] a la suma, basado en su artículo [87]. También puede consultarse el artículo [134] de John Michael McNamee, que incluye el código de dos algoritmos de suma bastante eficientes.

Al menos los más comunmente citados.

5.1.1. Interfaces para los algoritmos de suma

Ojo: `S` $\equiv fl(S)$ pues S es la suma exacta.

En este capítulo, `A` es el arreglo (o conjunto) con `n` elementos a_i, `S` o `Suma` la variable donde se acumula la suma y e el error absoluto. Se supondrá en general que se usa precisión doble. Es posible adelantar que a la fecha no hay ningún algoritmo que garantice la cota perfecta y que ninguno es superior al resto en todas las posibles situaciones.

Lo que un programador desea tener es una función que trabaje de manera transparente. Lo más sencillo es simplemente enviarle el conjunto A como arreglo y que la función regrese la suma, en algo que tendría el prototipo

```
double Sumar(double const A[], int n);
```

donde el arreglo se declara `const` para evitar que sea modificado dentro de la función. El valor n se requiere para conocer la cantidad de elementos. El programador utilizará la función como caja negra, sin saber lo que ocurre adentro. Por ejemplo, el segmento de código siguiente.

```
CapturarCalificaciones(Calif,n);
S = SumarCalificaciones(Calif,n);
Promedio = S/n;
```

Lo que no está previsto en esta interfaz es qué hacer en el caso de que la suma sea ∞ o NaN, que son posibles valores de retorno de la función `Suma`. Lo correcto sería verificar el valor de S luego de la llamada a `Suma` o bien leer las banderas del procesador. La forma más sencilla es comprobar con `if ( isnan(S) )` y `if ( isinfinity(S) )` y tomar la medida adecuada.

En el caso de saber que el conjunto A tiene un número de condición alto, entonces es prudente disponer de una funión que utilice parámetros como el error relativo máximo esperado (determinado por el programador) o el número de condición (calculado por la función).

Por otra parte, puede ser que por el tipo de aplicación que se est'a programando sea preferible que la suma se realice con la mayor exactitud posible a costa del tiempo empleado o al revés, se requiere la suma en el menor tiempo posible, quedando le exactitud en segundo término.

También es posible que para favorecer el desempeño de un algoritmo sea preferible que la función sí modifique el arreglo `A` a diferencia de lo indicado en la primera interfaz. De cualquier manera, el programador puede sacar copia del arreglo antes de la llamada a la función suma, si considera conveniente tener la información original para después.

Una librería completa requiere una función suma con varios parámetros que servirían para que internamente sea llamada la función adecuada, dependiendo de los mismos. En librería de este tipo tal vez sea preferible tener una estructura donde se incluya el arreglo, su tamaño, número de condición, la suma, error relativo, si se puede modificar y otros valores[2].

5.2. Algoritmos de suma recursiva

La suma directa es lo primero que se utilizó por ser el método natural. Consiste en el código

Listing 5.1: Algoritmo normal de suma.

```
1  double SumaNormal(double const *A, int n)
2  {
3      double Suma=0;
4
5      for ( i=0; i<n ; i++ )
6          Suma += A[i];
7
8      return Suma;
9  }
```

Como las líneas 5 y 6 se realizarán de manera ininterrumpida, la variable `Suma` siempre está en un registro, por lo que cuenta con los bits de guardia, redondeo, el sticky y posiblemente bits extra. Aún con esa ayuda, el código 5.1 tendrá problemas de precisión.

El algoritmo básico es recursivo pues puede plantearse como

$$S_1 = a_1 \tag{5.10}$$
$$S_{i+1} = S_i + a_{i+1} \tag{5.11}$$
$$S = S_n \tag{5.12}$$

por lo que también se le conoce con ese nombre: *suma recursiva*. La complejidad temporal es $O(n)$ ya que recorre el arreglo de n elementos.

En algunos lenguajes se tiene rutinas que realizan la suma, pero internamente deben aplicar algún algoritmo como 5.1 o algo más eficiente. Para comprender los problemas potenciales de una suma se presentan dos casos patológicos a continuación.

[2]Una clase con métodos sería agradable si la sobrecarga de operadores implícita en todo lenguaje orientado a objetos no causara problemas de redondeo (ver página 62).

5.3. Sumas problemáticas

El algoritmo básico presentado en el código 5.1 falla gravemente cuando se presentan arreglos con un acomodo mal intencionado. Por ejemplo, como la unidad de redondeo μ no altera al 1, es decir, $1 + \mu = 1$, sumando directamente el arreglo

$$\texttt{A} = \{1, \mu, \mu\} \tag{5.13}$$

se obtendría 1 pues al suma $1 + \mu$ se obtiene 1 y al sumar el tercer elemento se repite el fenómeno. Sin embargo, $1 + 2\mu = 1 + \varepsilon_M \neq 1$. Con sumar en sentido contrario sería suficiente para evitar este problema.

Otro arreglo con problemas involucra el mayor número de punto flotante $M = \beta^{e_{\text{máx}}}(1 - \beta^{-p})$. Por ejemplo, la suma del arreglo

$$\texttt{A} = \{M, M/2, -M\} \tag{5.14}$$

debe dar $M/2$ pues se cancelan el primero con el último elemento, pero como ocurre un overflow al sumar $M/2$ a M no se puede calcular el resultado final. De nuevo un reordenamiento de los sumandos eliminaría el problema.

Parece natural que el orden debe ser de menor a mayor en magnitud, pero si N es el menor NPF para el que ocurre que N+1=N, entonces la suma del arreglo

$$\texttt{A} = \{1, N, -N\} \tag{5.15}$$

resultará en 0 en vez de 1. En este caso, sumar en orden decreciente resulta ser lo correcto.

Tanto (5.13) como (5.14) y (5.15) son casos extremos, pero los mismos fenómenos ocurren a menor escala en programas cotidianos. El problema es que normalmente se desconoce la escala en la que ocurren.

5.4. Algoritmos básicos

Los primeros algoritmos son sencillos y hasta intuitivos de alguna manera. Han sido planteados sin autoría fácil de decidir y resultan casos particulares de un algoritmo más general. Incluso la suma recursiva es también un caso particular.

5.4.1. La suma en acumuladores separados

Esta fue la primera idea reportada en la que se intentaba minimizar los problemas de redondeo. Fue publicada en [189] por Jack Wolfe en 1964. La idea es simplemente utilizar varias variables para llevar sumas parciales separando los datos por rangos.

En ese entonces, Wolfe recomendaba utilizar una variable para todos los valores entre 0 y 9.99999, otra para valores entre 10 y 99.999999 y así sucesivamente. Si un acumulador llega al límite mínimo del siguiente rango, se suma al acumulador de ese rango y el acumulador se hace cero.

La cantidad de acumuladores depende de la aplicación y el mínimo es dos, que son los que se utilizan en el código 5.2[3].

[3] El código 5.2 es una versión minimalista pues no considera que si `S1` > `Sep`, se debe hacer `S2+=S1` y `S1`=0.

Listing 5.2: Algoritmo de suma con dos acumuladores.

```
1  double SumaAcumuladores(double const *A, int n, double Sep)
2  {
3      double S1=0, S2=0;

5      for ( i=0; i<n ; i++ )
6          if ( A[i] < Sep;
7              S1 += A[i];
8          else
9              S2 += A[i];
10     return S1+S2;
11 }
```

Incluso no representa ningún problema enviar a la función la cantidad de acumuladores que se deben usar o hacer una estimación de la cantidad adecuada dependiendo de un análisis de los datos, pero no es necesario. Este algoritmo se usa en raras ocasiones, pero significa un idea sencilla para usarse en diversas situaciones. Independientemente de la cantidad de acumuladores, este algoritmo recorre una sola vez el arreglo, por lo que su complejidad temporal es $O(n)$ y su complejidad espacial es $O(n+k)$, con k la cantidad de acumuladores.

En 1971, Malcolm propuso una extensión de la idea de Wolfe en [130]. La idea es separar cada elemento del arreglo en la suma de números cuya representación tenga los últimos bits apagados. Esto asegura que ningún número requiere toda la mantisa y de esta forma la etapa de normalización y renormalización no afectan la precisión.

El arreglo de tamaño n se convierte en un arreglo de al menos $2n$, dependiendo en cuantas partes se separe cada elemento a_i, lo que puede hacerse con técnicas como las de Dekker o Linnainma (ver la página 82) o manipulando directamente la representación de los NPF[4]. Una vez separados los números, se aplica el algoritmo de Wolfe. Lo ideal es tener al menos tantos acumuladores como partes en que se separe cada valor.

Implementaciones modernas del algoritmo de Malcolm extraen el exponente de la representación de IEEE del valor a_i, como en el código 4.12 y le aplican alguna operación para convertirlo en el índice del acumulador que le corresponde a ese valor. La operación más común es una división, pues de esta manera, varios exponentes consecutivos comparten el mismo acumulador, como se espera. En [134], esta versión resultó más eficiente que algoritmos más sofisticados.

En el código 5.3 se supone 64 acumuladores con precisión doble para un arreglo de números en precisión simple. De las líneas 13 a la 17, los números a_i son clasificaod por el rango de su exponente, al recorrer los bits de la representación el total de bits de mantisa+2 con la inteción de agrupar 4 exponentes. Eso decide a qué acumulador se agrega cada elemento, convertido a precisión doble. En el ciclo de las líneas 21 y 22 se suman los acumuladores.

Si acumuladores y datos miden igual no hay beneficio.

3 bits en vez de 2 servirían para agrupar 8 exponentes, lo que podría ser adecuado para arreglos grandes.

En las líneas 24 y 25 se obtiene el exponente de la suma y en la 26 se extrae la suma doble del acumulador correspondiente. Se suman de nuevo los acumuladores y

[4]Por supuesto, en esa época Malcolm no utilizó la representación interna porque era demasiado dependiente de la arquitectura.

Listing 5.3: Algoritmo de suma con múltiples acumuladores.

```
1  float SumCasc(float A[],int n)
2  {
3     union {
4        int n;
5        float f;
6     } U;
7     float  sum_float;
8     double Acum[64], sum_double, doublex, corr;
9     int    iexp, i;

11    for ( i=0 ; i<64 ; i++)
12       Acum[i]=0.0;
13    for ( i=0 ; i<n ; i++) {
14       U.f    =  fabs(A[i]);
15       iexp   =  U.n>>25;
16       Acum[iexp] += (double)A[i];
17    }

19    sum_double = 0.0;
20    for ( i=0 ; i<64 ; i++)
21       sum_double += Acum[63-i];
22    sum_float = (float)sum_double;

24    U.f  = fabs(sum_float);
25    iexp = U.n>>25;
26    Acum[iexp] -= sum_double;
27    corr   = 0.0;
28    for (i=0;i<64;i++)
29       corr += Acum[63-i];
30    sum_double += corr;

32    return (float)sum_double;
33 }
```

se hace la corrección final en la línea 29.

Al código anterior habría que agregarle revisiones adicionales para evitar overflow en el arreglo de acumuladores en el caso de que n sea muy grande. Lo más sencillo es pasar lo del acumulador en riesgo a un acumulador mayor[5] con la corrección correspondiente. Algo que no agrada es que la complejidad temporal del algoritmo depende del rango de los exponentes de a_i.

5.4.2. Suma en orden de magnitud creciente.

Para evitar el fenómeno de (5.13), lo correcto y más secillo es ordenar el arreglo en magnitud creciente, para comenzar a sumar los números de menor magnitud en valor absoluto primero. Implica ordenar los datos y luego aplicar el algoritmo recursivo. Ya Wilkinson había observado que en ese orden, el error (5.8) era menor. La complejidad temporal debe incluir el ordenamiento, con lo que se eleva a $O(n \log n)$ para ordenamientos como quick sort.

Una modalidad parecida es ordenarlos en el sentido contrario, pero el error por redondeo puede ser mayor. Este orden sólo funciona en casos como el arreglo (5.15). En general, se sabe que el orden decreciente es mejor cuando la suma es mucho menor que la magnitud de los elementos de arreglo, es decir, cuando hay mucha cancelación. En cualquier otro caso es mejor el orden creciente.

También se ha experimentado combinando esta técnica con la de acumuladores. Para programarlo, lo único que se requiere es agregar una instrucción al código 5.1, el ordenamiento del arreglo antes del ciclo de la línea 4, por lo que no se presenta el listado.

Cualquier algoritmo es bueno, de preferencia uno que no demerite el desempeño demasiado.

En el aspecto del mejor orden, Robertazzi y Schwartz publicaron [162], en el que determinan la influencia del orden en la exactitud del resultado. Calculan el error promedio final en términos de la media y la varianza de los datos, suponiendo distribuciones uniforme y exponencial para los órdenes y algoritmo básicos (5 en total).

El estudio anterior es interesante pues el error esperado es menor que las cotas típicas, que siempre miden lo peor que puede pasar. Entre otras cosas, revela lo que es de esperarse: el orden creciente es el que menos errores ocasiona y el orden decreciente es 2.6 veces mayor.

5.4.3. Variantes de Demmel y Hida

En 2002, Demmel y Hida presentan en un reporte técnico [49] una variante donde ordenan el arreglo en orden decreciente del exponente de la representación de IEEE, suponiendo que la suma parcial S_i posee más bits extra que cada a_i, es decir, hay registros con precisión extra. Posteriormente publicaron [50,51] con mejoras, otros detalles y aplicaciones. En la práctica esto sí aplica por la arquitectura de los procesadores en cuanto a los registros flotantes.

La variante trabaja independientemente de cond(A), acotando el error como $e_S \leq 1.5\text{ulp}(S)$, aunque si $S = 0$, la suma es exacta siempre.

Supóngase que p es la cantidad de bits de la mantisa, considerando el bit implícito y que P es la mantisa del registro flotante. Demmel y Hida demuestran que si el

Se supone que $P > p$.

[5] Al acumulador que correspondería si lo acumulado fuera el siguiente a_i.

redondeo es al más cercano y si hay undeflow gradual, entonces es posible sumar un conjunto de hasta

$$n \leq \left\lfloor \frac{2^{P-p}}{1-2^{-p}} \right\rfloor + 1 \tag{5.16}$$

números garantizando una suma correctamente redondeada hasta 1.5 ulp[6]. Si la cota se rebasa, puede perderse toda la exactitud, lo que significa un error relativo de hasta 1.

Si se utiliza como acumulador una variable de precisión doble y se suman datos en precisión simple, es posible sumar hasta

$$\frac{2^{53-24}}{1-2^{-24}} \approx 5.\,368\,709\,44 \times 10^{8}$$

elementos. En el caso de un acumulador de precisión cuádruple, se pueden sumar hasta

$$\frac{2^{113-53}}{1-2^{-53}} \approx 1.\,152\,921\,505 \times 10^{18}$$

números de precisión doble. En ambos casos se garantiza un error relativo bajo.

En el caso de confiar sólo en los registros de la computadora, para datos en precisión doble en Intel, con registros flotantes donde la mantisa utiliza 64 bits, se podrían sumar hasta

$$\frac{2^{64-53}}{1-2^{-53}} = 2048$$

números con la misma garantía.

Adicionalmente, Demmel y Hida presentan dos variantes que ahorran tiempo en el ordenamiento de las a_i pues es lo que más tiempo consume. Como los exponentes son muchos menos que el total de elementos, pues cada exponente puede agrupar muchos números, es de considerarse utilizar ordenamientos con complejidad temporal lineal, como el ordenamiento por conteo, que sólo requiere un poco más de espacio. De tratarse de precisión doble, el rango de los exponentes es de poco menos de 2000. La primera variante consiste en ordenar sin considerar los bits menos significativos del exponente, lo que resulta en un orden parcial.

La otra variante la aplican al algoritmo de la sección 5.4.1 de acumuladores. La cantidad óptima de acumuladores depende de la cantidad de bits del exponente, E, pues se utiliza la idea de Malcolm (como en el último párrafo de la sección 5.4.1). En el caso de un acumulador por exponente, la cota (5.16) se reduce a

$$n \leq 2^{P-p-E} \tag{5.17}$$

lo que es bastante pesimista.

Por ello también es posible reducir la cantidad de acumuladores a $N = \lceil P/p \rceil$ y ordenar los números que corresponden a cada acumulador para aplicar el algoritmo original (con los a_i de cada acumulador en orden), lo que aumenta la cota (5.17). Esta es la cuarta variante.

[6] En su publicación dan diversas cotas dependiendo de la relación entre p y P.

Listing 5.4: Algoritmo de suma por inserción o cascada.

```
1  double SumaCascada(double *A, int n)
2  {
3      double S=0;
4
5      while ( n>1 ) {
6          for ( i=0; i<n/2 ; i++ )
7              A[i] = A[2*i]+A[2*i+1];
8          n/=2;
9      }
10     return A[0];
11 }
```

5.4.4. Suma por inserción o de cascada

En 1970, Peter Linz propuso en [127] un algoritmo en el que los elementos del arreglo se suman por parejas, produciendo al final de la primera etapa un arreglo con la mitad de los elementos. El proceso se repite hasta que queda un arreglo de un solo elemento, que es la suma total.

Si se supone que el tamaño del arreglo es una potencia de dos por simplicidad, el código 5.4 implementa el algoritmo.

En este caso el arreglo no se recibe como `const` como en (5.1) porque es modificado. Cada iteración del ciclo `while` recorre el arreglo hasta la mitad de la iteración anterior.

Como es fácil de ver, este algoritmo puede preferirse en el caso de contar con arquitectura paralela pues el tiempo de cómputo se reduce de $O(n)$ a $O(\log n)$.

* * *

En todos los algoritmos anteriores se realizan de una u otra forma las $n-1$ sumas individuales requeridas, por lo que en cada ocasión que se repite el paso (5.11) ocurre

$$fl(S_i + a_{i+1}) = (S_i + a_{i+1})(1+\delta_i), \quad |\delta_i| < \mu.$$

por lo que la cota del error relativo en la suma final s es

$$e_S \leq (n-1)\mu \sum_{i=1}^{n} a_i \tag{5.18}$$

como máximo pesimista. Una cota más exacta la presenta Higham en [87] y es

$$e_S \leq \frac{n\mu}{1-n\mu} \sum_{i=1}^{n} |a_i| \,. \tag{5.19}$$

Algunos autores sugieren utilizar esta cantidad para decidir si se requiere un algoritmo más preciso. Si $n\mu/(1-n\mu)\sum_{i=1}^{n}|a_i| > \epsilon|S|$ para la tolerancia ϵ entonces aplicar un algoritmo más preciso.

Este es uno de los parámetros que pueden manejarse en la interfaz.

Los tres algoritmos anteriores no mejoran notablemente el procedimiento normal de suma, con excepción de las variantes de Demmel y Hida.

Es necesario aclarar que las cotas para el error relativo son pesimistas en el sentido de que es lo peor que puede ocurrir, pero no lo que se espera en el caso medio.

Listing 5.5: Método de suma de Pichat.

```
1  double SumaPichat(double const *A, int n, double Tol)
2  {
3      double Suma, Sa, SumaErr;
4      double *Err;

6      /* Memoria para el arreglo de errores */
7      if ( !(Err = (double *)malloc(n*sizeof(double)) ) )
8          return NaN;  /* No hay nada mejor */

10     Suma=A[0];
11     for ( i=1; i<n ; i++ ) {
12         Sa = Suma;
13         Suma += A[i];
14         Err[i-1] = (Sa-Suma)+A[i];
15     }
16     SumaErr = SumaPichat(Err,n,Tol);
17     if ( fabs(SumaErr)<Tol )
18         Suma += SumaErr;

20     return Suma;
21 }
```

5.5. Método de Pichat

En 1972, Pichat publicó en [152] un algoritmo que refina iterativamente hasta reducir el error debajo de un umbral selecionado. El algoritmo puede implementarse como en el código 5.5, que almacena el error de cada iteración en un arreglo, para posteriormente sumarlos y comprobar que el error total es menor que cierta tolerancia. De no serlo, se suman los errores con el mismo algoritmo y luego se restan de la suma final.

El algoritmo original de Pichat indica que la suma de errores debe repetirse recursivamente hasta no superar la tolerancia, a diferencia del código 5.5 donde se realiza una sola vez. De una manera coloquial, el algoritmo junta la "rebaba numérica" para que al final pueda agregarse de nuevo. Su complejidad temporal depende de cond(A), lo que determina la cantidad de llamadas recursivas. El mínimo peor caso es $O(4n)$ temporal y $O(2n)$ espacial, por el arreglo de errores.

Aunque es más lento que todos los anteriores, es fácilmente paralelizable. Tiene mejor precisión pero la idea básica (5.2) (ver figura 5.1) la utiliza de mejor manera el siguiente algoritmo

5.6. Suma compensada o de Kahan

William Velvel Kahan publicó en 1965 en [98] un algoritmo de *suma compensada* en el que cada iteración se conserva el error para intentar sumarlo en la siguiente iteración. La intención original de Kahan era poder simular precisión doble en los

Listing 5.6: Suma compensada o de Kahan.

```
1  double SumaKahan(double const *A, int n)
2  {
3      double Suma=0, Err=0, sa, t;

5      for ( i=0; i<n ; i++ ) {
6          s = Suma;
7          t = A[i]+Err;
8          Suma = s+t;
9          Err = (s-Suma)+t;
10     }
11     Suma = Suma+Err;
12     return Suma;
13 }
```

compiladores que solo tuvieran precisión simple[7], en particular en Fortran[8].

El algoritmo de suma compensada se presenta en el código 5.6.

Un procesador moderno cuenta con suficientes registros flotantes como para almacenar cada variable del ciclo en alguno, por lo que se dispone de los bits suficientes para garantizar un buen desempeño numérico. Claro, realiza 4 pasos cada iteración, por lo que invierte $4 \times (n-1)$ pasos en total.

Retomando la simbología de la figura 5.1, lo que ocurre en la línea 7 del código 5.6 es que la cantidad e se suma a b, que representa el siguiente término del arreglo.

Este sencillo algoritmo logra un error relativo muy bajo de $(2\mu + n\mu^2)|S|$. La cota puede mejorarse si se considera que en cada iteración es posible sumar el error de manera selectiva, a la suma parcial o al sumando siguiente, dependiendo de la magnitud.

En el planteamiento original de Kahan no aparece la línea 11 del código 5.6.

Años después se intento emplear el algoritmo de suma recursiva para recuperar todos estos errores en lugar de sumarlos en cada iteración, pero el error relativo aumenta a $(2\mu + n^2\mu^2)|S|$ si la cantidad de datos n es pequeña ($n\mu \leq .1$).

Uno de los problemas de la suma compensada es que agrega el error de la iteración anterior al nuevo término a_i, pues en general se espera que la suma parcial sea mayor que cada término siguiente. Como a_i puede ser mayor que la suma parcial S_i, a la que el error ya no pudo ser agregado, entonces con mayor razón no se sumará a ese nuevo término. Parece mejor idea un segundo intento de sumar el error a la suma parcial en espera de que los bits extra rescaten algunos de los bits perdidos. Eso significa un el cuerpo del ciclo en el código 5.6 podría cambiarse por el código 5.7.

La diferencia de desempeño entre estas dos versiones se mostrará en el experimento de la sección 5.11. Lo obvio es que las operaciones aumentan debido a la bifurcación del `if`, que es poco recomendable si se desea explotar el pipeline de los procesadores.

[7] En esa época había computadoras que truncaban o redondeaban antes de normalizar por lo que el algoritmo no funcionaba en ellas.

[8] Incluso comenta que "*la precisión doble pronto será universalmente aceptable como un sustituto para la ingenuidad en la solución de problemas numéricos*".

Listing 5.7: Variante de suma compensada.

```
1       if ( A[i] < Suma ) {
2           s = Suma;
3           t = A[i]+Err;
4       } else {
5           s = A[i];
6           t = Suma+Err;
7       }
8       Suma = s+t;
9       Err = (s-Suma)+t;
```

5.7. Destilación de Anderson

En 1999 se publicó una idea de I. J. Anderson en [10] que utiliza la idea de representar con dos variables la suma exacta, como en 5.2 (página 109), solo que aquí se trata de $a - b = s - e$. El algoritmo de Anderson es el siguiente:

1. Ordenar los números en orden decreciente en valor absoluto.
2. Encontrar los primeros dos que tengan signo opuesto. Sean a y b.
3. Eliminarlos de la lista.
4. Reescribirlos como en la ecuación 5.2.
5. Agregarlos al conjunto de números ordenados en las posiciones que corresponda.
6. Repetir desde el paso 2 hasta tener puros números con el mismo signo.
7. Aplicar el algoritmo de Kahan.

Este algoritmo es particularmente útil cuando el conjunto de números está mal condicionado y es propenso a cancelaciones graves, como el arreglo (5.15). Al llegar al último paso, el conjunto A tiene número de condición 1 pues

$$\sum_{i=1}^{n} |a_i| = \left| \sum_{i=1}^{n} a_i \right|$$

y su suma es exactamente la misma que el conjunto original. Con esto, el algoritmo es completamente estable.

En el caso de que a y b difieran enormemente en magnitud, es preferible separarlos con algo como el código 5.8 que se conoce como *reducción*. La línea 7 no sirve si $\beta \neq 2$. Entonces s y e se regresan a la lista inicial y se continúa con el algoritmo de manera normal.

Listing 5.8: Algoritmo de reducción de Anderson.

```
1       s = a;
2       while ( (a-s) != a ) {
3           e = s;
4           s = s/2;
5       }
6       s = a-e;
7       e = a-s;
```

Listing 5.9: Algoritmo de destilación simple.

```
1   /* Repetir el ciclo for hasta que
2           A[i] <= 2^-P*|A[i+1]|  para toda i */
3   for (i=0 ; i<n-1 ; i++ ) {
4       s = A[i] + A[i+1];
5       if ( fabs(A[i]) > fabs(A[i+1]) )
6           A[i] = (A[i]-s) + A[i+1];
7       else
8           A[i] = (A[i+1]-s) + A[i];
9       A[i+1] = s;
10  }
```

Según el análisis de Anderson, este algoritmo tiene una complejidad temporal de $O(n^2) + O(n \log n)$ debido al ordenamiento inicial y a las posibles aplicaciones recurrentes antes de tener un conjunto A con elementos del mismo signo.

Aprovechando que el paso 4 produce un valor muy pequeño, Anderson propone una variante de su algoritmo.

1. Separar A en dos conjuntos según su signo: positivos ($\mathcal{P}$) y negativos ($\mathcal{N}$), ambos en orden decreciente.
2. Retirar el primer elemento de cada conjunto y calcular su suma como (5.2) $s+e$.
3. Agregar s a $\mathcal{P}$ o $\mathcal{N}$ según corresponda y w a un conjunto inicialmente vacío $\mathcal{E}$.
4. Repetir pasos 2 y 3 hasta que $\mathcal{P}$ o $\mathcal{N}$ esté vacío.
5. Los elementos de $\mathcal{E}$ se agregan a $\mathcal{P}$ o $\mathcal{N}$ según corresponda.
6. Se calcula la suma (con el algoritmo de Kahan) de $\mathcal{P}$ y $\mathcal{N}$. Se calcula el número de condición $|(S_\mathcal{P} + S_\mathcal{N})/(S_\mathcal{P} - S_\mathcal{N})|$. Si no es 1 se repite desde el paso 2.
7. La suma final se calcula con el algoritmo de Kahan.

Para evitar problemas de ciclado, Anderson recomienda no separar la suma $a + b$ entre dos cantidades tan dispares como s y e, por lo que transformaciones libres de error como la de Dekker (4.2, pag. 82) se pueden emplear.

Este algoritmo obtiene mejor precisión que el de suma compensada, segú los experimentos en [10].

5.8. Suma con doble destilación o de Priest

Un algoritmo de suma que alcanza casi la cota perfecta es el de Priest en su tesis de doctorado [158], conocido como *doble destilación*, derivado a partir del de Kahan.

Para comprenderlo mejor, la idea de destilación puede verse como en el segmento de código 5.9, que es lo que se aplica en el algoritmo de Anderson.

El ciclo debe repetirse hasta que $a_i \leq 2^P |a_{i+1}|$ para toda i. Al término de la destilación, el arreglo original ha quedado sustituido por otro arreglo cuya suma es igual a la del original, pero que requiere menos bits en promedio para las a_i. El algoritmo

Listing 5.10: Doble destilación o de Priest.

```
1  double SumaPriest(double const A[], int n)
2  {
3      int i;
4      double S, e, r, y, a, t, b, c;

6              S=0;
7              e = 0;
8              for (i=1; i<n; i++) {
9                  r = A[i];

11                 y = r + e;
12                 a = r - ( y - e );

14                 t = S + y;
15                 b = y - ( t - S );

17                 c = a + b;
18                 S = t + c;
19                 e = c - (S - t);
20             }
21             return S;
22 }
```

tal cual tiene complejidad cuadrática, pero puede reducirse a $O(n \log n)$, según Priest en [158][9].

La idea de Priest es aplicar la compensación de Kahan sobre sí misma, como se muestra en el código 5.10. La idea de la suma compensada se aplica tres veces, en la misma forma (líneas 11-12, 14-15 y 17-19).

En el código 5.10, las variables `A`, `n`, `S` y `e` siguen teniendo el mismo significado. Las nuevas variables son para porder aplicar la compensación de Kahan sobre sí misma dos veces más[10]. La idea de regresar `S+e` no parece mala, tal como cuando Kahan agregó una línea similar a su planteamiento de 1964, pero debe recordarse que si la suma S ocupa toda su mantisa, $\text{ufp}(e) < \text{ulp}(S)$.

La complejidad temporal está dominada por el ordenamiento inicial, por lo que es de $O(n \log n)$.

Según demuestra Priest en su tesis, en el caso de que se cumplan las condiciones de la norma de IEEE, el error relativo estará acotado por $2\mu|S|$, que es apenas el doble de la cota ideal, independientemente del número de condición del arreglo. El error se mantiene siempre que la cantidad de elementos esté acotada por

$$n \leq \beta^{p-3}. \tag{5.20}$$

En doble precisión, (5.20) implica que se pueden sumar hasta $2^{49} \approx 5.629\,499 \times 10^{14}$ manteniendo ese error relativo.

[9] Una posible interpretación es pensar que el código 5.9 destila como el ordenamiento de burbuja, mientras que la idea de Priest aplica merge sort.

[10] Por lo que se dice que debiera llamarse *triple compensación.*

Demmel y Hida utilizan este algoritmo dentro de una de sus variantes de destilación. La cuarta de ellas de la sección 5.4.3 se aplica para reducir el tamaño del arreglo inicial y luego aplicar el método de Priest. Adicionalmente, sugieren cómo combinar sus variantes con la doble destilación para lograr el mejor resultado, dependiendo del valor de n.

Un algoritmo tan preciso como el de Priest ha logrado buenos resultados en diversas áreas de computación pues además de estable es sencillo e independiente del número de condición de A.

5.9. Método de Rump-Ogita-Oishi

El método Rump-Ogita-Oishi (ROO) se basa en la separación de valores con transformaciones libres de error pero con mayor cuidado. Antes de presentarlo es conveniente analizar su procedencia.

5.9.1. La idea de Zielke y Drygalla

En 2003, Zielke y Drygalla presentaron en [191] un algoritmo que recurre una vez más a la idea de separar los elementos a_i, pero tomando en cuenta el máx $|a_i|$, de tal manera que para sumandos pequeños, la parte mayor sea cero. La transformación libre de error debe cumplir que la suma de las partes mayores se realice sin error, lo que se repite (aplicando el mismo algoritmo una y otra vez) hasta que las partes menores son cero. Cuando esto ocurre, la suma parcial de partes mayores S_j es igual a la suma final S. Las partes superpuestas de las sumas parciales se eliminan agregándoles como acarreo en orden creciente para terminar sumando las sumas parciales en orden decreciente.

Con más detalle: primero se calcula el entero positivo k tal que máx $a_i < 2^k$ y una M tal que $n < 2^M$. Siendo p la precisión, se extraen $p - M$ bits de a_i, desde la posición k hasta $k - (p - M) + 1$, y se colocan en q_i (las partes mayores) y a un vector de acarreos. Se agrega q_i a la suma parcial S_1, que es exacta por la selección de M.

Se continúa extrayendo los $p - M$ siguientes bits de las a_i, desde el $k - (p - M)$ hasta $k - 2(p - M) + 1$ y se agregan a S_2. El proceso continúa hasta que las partes remanentes son cero. Debe ser claro que $\sum a_i = \sum S_j$. Los traslapes de S_j se eliminan agregándolos a S'_j con bits de acarreo en orden creciente. Ahora $\sum a_i = \sum S'_j$ y los bits no se traslapan pues forman una representación binaria exacta del resultado. Por último, las S'_j se suman en orden decreciente.

El algoritmo de Zielke y Drygalla adolece de varias cosas. En las 100 páginas de su artículo, se presenta un código de solo 7 líneas en MatLab. No se presenta un análisis y tampoco el caso de overflow[11]. Por último, debido a un escalamiento deficiente, el rango de las a_i está muy limitado.

[11] Rump, Ogita y Oishi presentan un arreglo de sólo 3 elementos con el que ese código no funciona, causando overflow, como en el caso del arreglo (5.14).

5.9.2. Algoritmo de ROO

Lo anterior resulta en una buena idea, pero mal concluida y presentada. Aprovechando esto, Rump, Ogita y Oishi publicaron[12], en 2007 el artículo en dos partes [164,165][13] en los que subsanan todas las deficiencias teóricas y prácticas de Zielke y Drigalla.

En la primera parte, presentan un robusto algoritmo que calcula $fl(S)$ correctamente redondeado que se adapta al número de condición de A y que no depende del rango de los exponentes de las a_i. No requiere de operaciones especiales ni supone precisión extra intermedia.

La segunda parte es particularmente útil para aritmética de intervalos pues inlcuye algoritmos con redondeo direccionado. Entre otras cosas, incluye un algoritmo especial para calcular el signo de S sin evaluar la suma final.

La mejora sobre el algoritmo de Zielke y Drygalla es indudable, tanto con la fundamentacin teórica como en la implementación. Aplican la cantidad óptima de transformaciones libres de error, mejoran el escalamiento y evitan el acarreo y sumas parciales innecesarias. Con todo eso, logran la cota perfecta: un error relativo de μ. La complejidad de su algoritmo es el logaritmo de cond(A).

Lo central en el algoritmo ROO es la separación de cada a_i con transformaciones libres de error. Como se presentó en (4.2), la técnica de Dekker es simplemente

$$\begin{aligned} x &= fl(a+b) \\ q &= fl(x-a) \\ y &= fl(b-q) \end{aligned} \tag{5.21}$$

para transformar $a+b$ en $x+y$ y resulta más eficiente que la de Knuth (5.5). El algoritmo ROO transforma sin errores el vector de valores a_i en dos NPF τ_1, τ_2 y otro vector con valores a_i', de tal manera que $\sum a_i = \tau_1 + \tau_2 + \sum a_i'$ exactamente. La suma S se redondea correctamente como $\mathrm{fl}(S) = \mathrm{fl}(\tau_1 + (\tau_2 + \sum a_i'))$.

Para esto es necesario una separación como (5.21) que no requiera que $|a| \geq |b|$. El único requisito es que no ocurra que el menor bit no cero de a sea menor que el bit menos significativo de b. POO demuestran que si (5.21) se aplica, entonces $x+y = a+b$ y $x = \mathrm{fl}(a+b)$ como se sabe, pero además

$$|y| \leq \mu \,\mathrm{ufp}(a+b) \leq \mu \,\mathrm{ufp}(x) \tag{5.22}$$

$$q = \mathrm{fl}(x-a) = x-a \tag{5.23}$$

$$y = \mathrm{fl}(b-q) = b-q \tag{5.24}$$

lo que significa que las diferencias (5.23) y (5.24) también son exactas. La separación de ROO para cada a_i es la siguiente. Sea σ una potencia de 2 no menor que máx$\{a_i\}$.

$$\begin{aligned} q &= \mathrm{fl}((\sigma + a_i) - \sigma) \\ a_i' &= \mathrm{fl}(a_i - q) \end{aligned} \tag{5.25}$$

[12] Basados en [163], un reporte técnico de 2005.

[13] Una de las mayores bondades de los artículos de Rump, Ogita y Oishi es la excelente investigación de los trabajos relacionados con el tema de suma de NPF. Aunque comentan que [88, 125] son excelentes trabajos (lo que el autor no duda), la introducción de [164] es superior, lo que se nota en las 50 referencias sobre el tema. Además, lo riguroso, detallado y extenso de su exposición (60 páginas) lo hace adecuado como material de estudio para todo analista numérico. Una introducción más sencilla es su artículo anterior [144].

Listing 5.11: Algoritmo de separación de POO para vectores.

```
1  void ExtraerVector(double const *A, int n, double Sigma,
2                     double *Ap, double *Tau)
3  {
4      int i;
5      double q;

7      *Tau = 0.0;
8      for ( i=0; i<n ; i++ ) {
9          ExtractEscalar(A+i,Sigma,Ap+i,*q);
10         *Tau = *Tau + q;
11     }
12 }
```

La transformación (5.25) separa a_i en dos partes que dependen de σ, a diferencia de (4.2, página 82) en que se separa dependiendo del exponente. Eso significa que la parte mayor puede tener todos los p bits de la mantisa. La separación no es simétrica en el sentido de que $-a_i$ puede separarse distinto que a_i, debido al valor fijo $\sigma > 0$.

Aunque la idea original de (5.25), llamada `ExtractEscalar` por los autores, ya era conocida, el análisis de ROO es nuevo y revela que si $|a_i| \leq 2^{-M}\sigma$ para algún número natural M, entonces[14]

$$a_i = q + a_i' \tag{5.26}$$
$$|a_i'| \leq \mu\sigma \tag{5.27}$$
$$|q| \leq 2^{-M}\sigma. \tag{5.28}$$

La separación 5.25 se aplica a vectores, lo que resulta en el código 5.11. En este caso, se ha preferido enviar la dirección de los vectores A' y q y no preocuparse por la memoria.

La función `ExtractEscalar` no debiera ser problema para el lector. Según observan ROO, el código 5.11 es naturalmente paralelizable, lo que lo hace adecuado para explotar el pipeline de los procesadores modernos. Hasta este punto, se tiene la transformación libre de error $\sum a_i = \tau + \sum a_i'$.

ROO demuestran que si $\sigma \in \mathbb{F}$, $\sigma = 2^k \in \mathbb{F}$ para algún entero k, $n < 2^M$ para $M \in \mathbb{F}$ y $\max\{a_i\} \leq 2^{-M}\sigma$, entonces

$$\sum a_i = \tau + \sum a_i' \tag{5.29}$$
$$\max\{a_i'\} \leq \mu\sigma \tag{5.30}$$
$$|\tau| \leq (1 - 2^{-M})\sigma \tag{5.31}$$

y la limitación $n < 2^M$ se requiere para que $|\,\mathrm{fl}(\tau + \mathrm{fl}(\sum a_i'))| \leq \sigma$ se cumpla. Sobre los valores de σ y M no se ha supuesto nada excepto su relación.

El mejor valor posible para M es $\lceil \log_2(n+1) \rceil$, pero para evitar el logaritmo binario se puede calcular con el código 5.12, que busca la siguiente potencia mayor

[14] El planteamiento de ROO fija la base numérica en $\beta = 2$, sin suponer nunca la posibilidad de aritmética decimal. Todos sus ejemplos son considerando precisión doble.

Listing 5.12: Cálculo del valor M para acotar n.

```
1  double CalcularM(double x)
2  {
3      double y, P;

5      if ( x==0 )
6          return NaN;
7      y = x/MU;
8      /* Revisar la bandera de overflow */
9      P = fabs((x+y)-y);
10     if ( P==0.0 )
11         P = fabs(x);
12     return L;
13 }
```

que x. En algunos lenguajes, particularmente los de orientación científica, no es difícil encontrar funciones de biblioteca que ya realizan este tipo de cálculos[15].

El código 5.12 trabaja bien mientras no ocurra overflow, para lo que habría que agregar algunas instrucciones más. Con el underflow no hay problema. La constante `MU` puede calcularse de muchas maneras, entre ellas la indicada en el código 2.1 de la página 31, eliminando la última línea[16].

El algoritmo trabaja aplicando la transformación libre de error `ExtractVector` y modificando el valor σ mientras que la suma parcial sea menor que $2^{2M}\mu\sigma$, que es la cota óptima para detener las iteraciones. Es en escencia lo que hace el código 5.13.

La función `SumaExacta` es simplemente la técnica de Dekker 5.21, que se presenta en el código 5.14.

Con el código 5.13, lo único que habría que hacer es hacer la llamada a la función `Transform` y luego calcular `t1+(t2+Suma(A))`, con lo que el resultado estará correctamente redondeado hasta el último bit, lo que significa un error de .5 ulp. Primer algoritmo con la cota perfecta.

El algoritmo, tiene complejidad $O(4m+n)$, donde m es la cantidad de repeticiones del ciclo `do--while`.

5.10. Algoritmo de Eisinberg y Fedele

En 2007 se publicó [57], en el que Eisinberg y Fedele presenta un algoritmo que, a costa de cambiar la representación de los NPF, obtiene un error relativo muy bajo en aplicaciones prácticas, según muestran los autores. No se requiere algún ordenamiento particular de A y es independiente de cond(A).

A diferencia de los métodos anteriores, donde se reordena A, se separan los valores a_i o se evalúan los errores en cada suma para agregarse al resultado, la idea de Eisenberg y Fedele es modificar la representación de las a_i antes de sumar, para garantizar que la suma sea correcta sin necesidad de operaciones como las mencionadas.

[15] Lenguajes científicos como Matlab o Mathematica ofrecen estas y otras facilidades.
[16] El código 2.1 calula el épsilon de máquina ε_M y no $\mu = \varepsilon_M/2$.

Listing 5.13: Transformación de a_i en (τ_1, τ_2, a_i').

```
void Transform(double A[], int n, double *Sigma
                    double *t1, double *t2)
{
     double max_ai;

     max_ai = maxAbsArr(A,n);
     if ( max_ai == 0 ) {
          *t1 = *t2 = *Sigma = 0.0;
          return 0;
     }
     M = SigPot2(n+2);
     Sigma_t = Pot2(M)*SigPot2(max_ai);
     t_t = 0.0;
     do {
          t = t_t;
          Sigma = Sigma_t;
          ExtraerVector(A,n,Sigma,A,&Tau);
          t_t = t + Tau;
          if ( t_t == 0.0 ) {
               Transform(A,n,Sigma,&Tau1,&Tau2);
               return;
          }
          Sigma_t = Pot2(M)*MU*Sigma;
     } while ( fabs(t_t)< Pot2(2*M)*MU*Sigma );
     SumaExacta(t1,t2,t,Tau);
}
```

Listing 5.14: Suma exacta de dos números (técnica de Dekker).

```
/* a+b se convierte exactamente a x+y */
void SumaExacta(double a, double b,
                double *x, double *y)
{
     double t;

     *x = a+b;
      t = x-a;
     *y = b-q;
}
```

El método aplica dos pasos:

1. Cada a_i es transformada a una nueva representación numérica equivalente, llamada $\widehat{\mathbb{F}}$. En ese conjunto $\widehat{\mathbb{F}}$, los números se representan como una combinación lineal de potencias de β, donde los coeficientes son enteros de $\mathbb{F}$.
2. Los números se suman de manera directa, ya que no hay error de redondeo al sumar enteros o potencias de β. En caso de que sean demasiados números, se recurre al algoritmo de Priest como apoyo en la última etapa.

Lo interesante del método de Eisenberg y Fedele es que la conversión se puede hacer rápidamente y luego sólo se trata de sumar enteros y multiplicar por potencias de β, lo que no consume tiempo.

5.10.1. Nueva representación para los NPF

Excluyendo por lo pronto los valores especiales

En la representación de los NPF se tiene que

$$x = (-1)^s \times m \times \beta^{e-p}$$

$e = \lfloor \log_\beta |x| \rfloor + 1$.

donde $\mathbb{F}$ queda determinado por un conjunto de parámetros (ver página 29), que es equivalente a

$$x = (-1)^s \sum_{j=1}^{q} x_j \beta^{e-E(j)} \tag{5.32}$$

donde

$$x_j = \sum_{k=E(j-1)+1}^{E(j)} d_k \beta^{E(j)-k} \tag{5.33}$$

$$E(0) = 0 \tag{5.34}$$

$$E(j) = E(j-1) + \alpha_j \tag{5.35}$$

los q números α_j suman p (la precisión contando el bit implícito), que determinan en cuántos términos se separará x. Los d_k son los dígitos binarios 0 o 1. En [57] se demuestra la equivalencia entre las dos representaciones, lo que se logra simplemente sustituyendo una en otra.

La forma de encontrar las cantidades x_j es mediante las fórmulas

$$x_n = \frac{x}{\pm\beta^e} = \sum_{k=1}^{p} d_k \beta^{-k} \tag{5.36}$$

$$x_j = \left\lfloor \beta^{E(j)} x_n - \sum_{i=j}^{j-1} \beta^{E(j)-E(i)} x_j \right\rfloor . \tag{5.37}$$

La conversión de $\mathbb{F}$ a $\widehat{\mathbb{F}}$ es casi directa. Como este método es muy distinto a los anteriores, vale la pena un pequeño ejemplo de conversión de formato.

Ejemplo. Sea $x = \mathrm{fl}(\pi) = 3.\,141\,592\,653\,589\,7932$ en doble precisión. Si se separa en 3 partes habría que dividir la mantisa de 53 bits en 17+18+18, que son los valores de las α_j. Con estos valores y las ecuaciones (5.34) y (5.35) se calculan las $E(j) = \{0, 17, 35, 53\}$.

Ahora ya se pueden calcular los x_j a partir de las fórmulas (5.36) y (5.37) de la siguiente manera:

$$\begin{aligned}
x_n &= \frac{\mathrm{fl}(\pi)}{\beta^e} = \frac{\mathrm{fl}(\pi)}{2^2} = .\,785\,398\,163\,397\,4483 \\
x_1 &= \left\lfloor 2^{17} \times x_n \right\rfloor = 102\,943 \\
x_2 &= \left\lfloor 2^{35} \times x_n - 2^{35-17} \times 102943 \right\rfloor = 185617 \\
x_3 &= \left\lfloor 2^{53} \times x_n - (2^{53-17} \times 102943 + 2^{53-35} \times 185617) \right\rfloor = 11544.
\end{aligned}$$

De esta forma, $\mathrm{fl}(\pi) = 102943 \times 2^{2-17} + 185617 \times 2^{2-35} + 11544 \times 2^{2-53}$ según la ecuación (5.32). De la misma manera, $\mathrm{fl}(\sqrt{2}) = 92681 \times 2^{-16} + 235935 \times 2^{-34} + 211916 \times 2^{-52}$ o $fl(\sqrt{2}) = 47453132 \times 2^{-25} + 109001677 \times 2^{-52}$.

El algoritmo que convierte de $\mathbb{F}$ a $\widehat{\mathbb{F}}$ es sencillo a partir del anterior ejemplo. La conversión en el sentido contrario es muy simple: solo falta normalizar cada término del formato y sumar. Como $\sum \alpha_j = p$ no hay pérdida de bits. A partir de la conversión de formato, el proceso de suma comienza.

5.10.2. La suma de NPF en $\widehat{\mathbb{F}}$

Cada a_i del arreglo que se desea sumar se ha reescrito en otro formato, como suma de enteros por potencias de β como en la ecuación (5.32), o equivalentemente

$$a_i = (-1)^s \beta^{e(i)} \sum_{j=1}^{q} x_j \beta^{-E(j)}.$$

De todos los exponentes de β en su conjunto (para todas las a_i), se localiza el menor, sea $e_{min} = \mín\{e - E(j)\}$. Todos los exponentes se pueden reescribir como la suma de e_{min} más una cantidad γ_i. Ahora cada valor a_i se reescribe como

$$a_i = (-1)^s \beta^{e_{min}+\gamma_i} \sum_{j=1}^{q} x_j \beta^{-E(j)}$$

por lo que la suma puede reescribirse de nuevo como

$$S = \sum_{j=1}^{q} f_j \tag{5.38}$$

donde

$$f_j = \beta^{e_{min}-E_j} \sum_{i=1}^{n} \mathrm{sgn}(a_i) x_j \beta^{\gamma(i)}.$$

La suma (5.38) es una suma de q enteros, por lo que su error es menor que la suma de los n enteros originales si $q < n$, que es lo esperado. Lo óptimo es que $q = 2$, lo que implica separar cada a_i en como la suma de sólo dos enteros, cada uno multiplicado por una potencia de la base. La ventaja es que el error relativo de la suma de dos NPF es la cota ideal para la suma de cualquier cantidad de números.

Esto será posible mientras se disponga de un registro donde se pueda almacenar el entero f_j con todos sus bits. El entero máximo en un registro con t bits es $2^t - 1$,

lo que limita la cantidad o magnitud de los elementos que se sumarán. En este caso, $t = p$ pues sólo los bits de la mantisa se puede utilizar. Como x_j está acotada por

$$\text{máx}\{|x_j|\} \leq \beta^{\alpha_j} - 1$$

y en el peor caso $\gamma(i) = e_{max} - e_{min}$, entonces debe cumplirse que

$$n\beta^{e_{max}-e_{min}}|x_j|_{max} \leq 2^p - 1. \tag{5.39}$$

Con $q = 2$, la ecuación (5.38) queda escrita como

$$S = \beta^{e_{min}-E_1}\sum_{i=1}^{n}\text{sgn}(a_i)x_1\beta^{\gamma(i)} + \beta^{e_{min}-E_2}\sum_{i=1}^{n}\text{sgn}(a_i)x_2\beta^{\gamma(i)} \tag{5.40}$$

que será exacta si la transformación de formato se ha realizado correctamente y si

$$n \leq \frac{\beta^p - 1}{\beta^{e_{max}-e_{min}}}\,\text{mín}\left\{\frac{1}{\beta^{\alpha_1}-1}, \frac{1}{\beta^{\alpha_2}-1}\right\} \tag{5.41}$$

que se obtiene por la ecuación (5.39). Para precisión doble, la cota es de 67 108 865 números. La cota de n requiere mayor atención. Si $q = 2$, entonces se pueden seleccionar $\alpha_1 = \lfloor p/2 \rfloor$ y $\alpha_2 = \lceil p/2 \rceil$ sin pérdida de generalidad, por lo que $\alpha_1 \leq \alpha_2$. Para que $n \geq 1$ se requiere que

$$\beta^p - 1 \geq \beta^{e_{max}-e_{min}}(\beta^{\alpha_2} - 1)$$

o equivalentemente

$$e_{max} - e_{min} \leq \alpha_2 = \lceil p/2 \rceil\,. \tag{5.42}$$

Si $e_{max} - e_{min} > \alpha_2$, el rango de los valores a_i impide una suma exacta, pero el resultado será $\text{fl}(S)$. La complejidad temporal es básicamente $O(2n)$ debido a que primero se recorre el arreglo para convertir a la nueva notación y luego se realiza la suma.

El caso en que la precisión p se queda corta.

Si el valor de n excede su cota pero se cumple (5.42) ocurre un overflow y el tratamiento es distinto. Los sumandos de (5.40) son exactos pero la suma excede el entero máximo. Eisenberg y Fedele representan la suma en términos del módulo 2^p como

$$\sum_{i=1}^{n} f_1(i) = \Delta_1 2^p + R_1 \tag{5.43}$$

$$\sum_{i=1}^{n} f_2(i) = \Delta_2 2^p + R_2 \tag{5.44}$$

por lo que (5.40) se escribe como

$$S = \Delta_1 2^{e_{min}-E(1)+p} + R_1 2^{e_{min}-E(1)} + \Delta_2 2^{e_{min}-E(2)+p} + R_2 2^{e_{min}-E(2)}. \tag{5.45}$$

Los cuatro números de (5.45) no contienen errores mientras se cumpla $e_{max} - e_{min} \leq \alpha_2$ y pueden sumarse con el algoritmo de Priest, con un error relativo de 2μ, el doble de cuando n no excede su cota. Sigue siendo muy rápido pues sólo se suman cuatro números en vez de n.

5.11. Un experimento comparativo

Cada artiículo referido en el que se presenta un método, generalmente se compara con algunos otros, por lo que se sugiere estuidiar tales experimentos. Por otra parte, en el artículo [134] John McNamee compara 14 métodos (o variantes) con 7 conjuntos de números distintos. No incluye los métodos de Eisinberg y Rum pues el artículo es de 2005, anterior a estos.

El estudio de McNamee revela que el método de Malcolm (código 5.3) es el más rápido y exacto, con el inconveniente de requerir acumuladores de mayor precisión. En el caso de que los datos tengan la más alta precisión y no contar con precisión múltiple, el de Priest resulta ser el mejor, pues su mejor competidor, Pichat, es mucho más veloz pero mucho menos preciso. Habría que agregar ahora los dos últimos métodos con cota perfecta en el estudio.

Pruebas sugeridas al lector para hacer más experimentos son arreglos:

- en desorden,
- cuya suma cause overlflow,
- con grandes cancelaciones,
- con diversos tamaos de arreglo para comparar velocidad y
- comparar con diversas distribuciones estadísticas de los datos.

Este es uno de tantos métodos de prueba para funciones de suma: un conjunto de valores cuya suma es conocida. En este caso, el conjunto de valores se forma con los términos de una serie infinita cuyo lmite es un número trascendente.

La famosa serie es

$$S(n) = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2}$$

cuyo límite es

$$\lim_{n\to\infty} S(n) = \frac{\pi}{6}.$$

Como el límite es un número irracional, es imposible aproximarlo con una computadora con exactitud. El NPF más próximo a $\frac{\pi}{6}$ correctamente redondeado es

$$\begin{aligned} \frac{\pi}{6} &= 0.523\,598\,775\,598\,298\,873\,077\,107\,230\,546\,583\,814\,032... \\ &\approx 0.\,523\,598\,775\,598\,298\,9 \end{aligned}$$

que es el resultado que se espera si se suman suficientes términos.

* * *

Pese a la cota del error perfecta de los últimos dos algoritmos, el problema seguirá estudiándose pues hay otras restricciones que son necesarias dependiendo de la aplicación. A veces se preferirá un método más veloz con un error relativo máximo predefinido o un algoritmo de ordenamiento cuyo código sea de tamaño más reducido para para incluirlo en algún circuito.

Además, el ordenamiento de números en base 10 puede ofrecer más alternativas, por ejemplo, habría que estudiar un posible adaptación del método de Eisinberg.

Capítulo 6

Raíces de ecuaciones

Obtener la raíz de una ecuación es uno de los temas que ha causado gran interés desde hace muchos siglos. Dada una expresión matemática de una variable, consiste en encontrar el (los) valor(es) que la anula(n). Por ejemplo si la expresión es $3x - 4$, hay que encontrar el valor x que hace que se cumpla $3x - 4 = 0$. Claro, este problema es sencillo y simplemente se despeja la variable x, encontrando que ese valor es $x = 4/3$. Si la ecuación es $\log x + x = 0$ entonces despejar ya no es posible y es necesario emplear otros medios.

Rara vez ocurre que la solución de un problema real termine al calcular las raíces de una ecuación. Lo que ocurre con más frecuencia es que calcular las raíces sea un paso intermedio dentro de un proceso más complejo. Eso es lo que hace importante su estudio, además de ser un tema que ilustra de manera sencilla muchos conceptos aproximación y de aritmética de punto flotante.

Son cientos de métodos conocidos para la aproximación de raíces[1] por lo que no se presenta más que una selección, determinada por los siguientes motivos.

- Métodos de importancia histórica.
- Métodos básicos, que sin ser necesariamente eficientes, introducen conceptos útiles.
- Métodos rebustos, generalmete preferidos en las librerías comerciales de software.
- Combinaciones de métodos que garantizan convergencia, a costa de eficiencia.
- Métodos que han sido muy analizados en la literatura.
- Extensiones teóricas importantes, como los que garantizan que en uno o dos pasos rebasan la precisión numérica nativa de una computadora convencional

Hay métodos que caen en varias de estas categorías. Aunque muchos tienen nombre, hay gran cantidad de variantes y propuestas diversas. No se puede pretender ser exhaustivo, a menos que se desee escribir varios tomos. Por otra parte, no se pretende abordar los métodos más sofisticados que permiten encontrar incluso raíces de

[1] Basta consultar el libro de Traub [179].

funciones especiales, donde es necesario utilizar técnicas más complejas, por ejemplo ecuaciones diferenciales o transformadas de Prüfer, lo que significa un grado mucho mayor de especialización (véanse por ejemplo [74, 153]).

De manera natural, muchos métodos se extienden y aplican a sistemas de ecuaciones, o al menos su idea básica. No se aborda el caso de las raíces de sistemas de ecuaciones, que cae de lleno en el área de optimizacin no lineal, para lo que hay gran cantidad de referencias.

La idea es complementar cursos de Métodos Numéricos, no de Investigación de operaciones.

En todos los casos se ha procurado dar las referencias que permitan al lector buscar detalles como ejemplos adicionales, demostraciones rigurosas, análisis adicionales o comentarios en pro o en contra de algunos métodos. Sólo se ha procurado mencionar el método y algunas consideraciones prácticas de su programación, desde el punto de vista de la APF.

Por otra parte, los métodos para encontrar raíces de ecuaciones se abordan en practicamente todos los cursos básicos de análisis o métodos numéricos, por lo que el lector encontrará facilmente gran cantidad de información. Algunos los presentan desde un enfoque matemático adecuado para los primeros semestres de una carrera superior, donde se prueban las condiciones de convergencia y presentan los términos de error, pero sin muchos de los detalles que en la práctica son importantes. Otros los presentan principalmente como recetas, incluendo su código, olvidando de nuevo incluir o explicar los detalles finos de programación.

Al buscar en otro libro, siempre se desea encontrar algo distinto, no lo mismo.

La intención es no repetir lo que se encuentra con mayor facilidad en los libros de análisis o métodos numéricos típicos, sino presentar una amplia variedad de métodos, dando énfasis a los detalles de programación y su fundamento matemático cuando es necesario o útil por su concepción. Basta con comparar el contenido de algunos de los textos que con mayor recurrencia se utilizan en estos cursos: Burden [25], Chapra [29], Kincaid [116] y Gerald [72], todos traducciones de su original en inglés.

Cuando resulte de utilidad, se presentará la forma de derivar los métodos pues en algunos casos es muy ilustrativa la forma de abordar y resolver un problema, como parte de la formación y entrenamiento que los futuros profesionales de la computación debe tener.

La mejor manera de aprender y comprender la teora.

Se recomienda hacer experimentos no sólo con código propio sino con paquetes como Maple, MatLab, Mathematica o cualquier otro. Incluso hojas de cálculo son útiles de vez en cuando, pues permiten realizar rápidamente experimentos numéricos antes de programarlos.

Un planteamiento más formal y propio para matemáticos, podría presentar el tema con conceptos de análisis funcional, topología, teoría de la medida y sistemas dinámicos desde la perpectiva de los espacios normados, en particular los de Banach. Aún siendo un tema excelente para introducir estos conceptos, se ha preferido enfatizar aquellos aspectos que impactan directamente en la actividad de la programación de computadoras, como los aspectos finos de la aritmética de punto flotante. Para aquellos lectores interesados, se introducen algunos conceptos más avanzados en la sección de Teoría de funciones iterativas.

Presentando el problema de manera precisa, se considera una función f real de variable real y se busca el valor $x \in [a, b]$ tal que $f(x) = 0$, suponiendo por lo general que f es invertible en $[a, b]$, es decir, que x sea un cero simple, para lo que es necesario que $f^{-1}(0) = x$. En la práctica normalmente el intervalo $[a, b]$ es pequeño pues casi todos los métodos de aproximación requieren de un valor cercano para comenzar las

iteraciones y acercarse poco a poco. Hay métodos generales con los que se busca el valor de la raíz y otros más específicos para polinomios, donde es frecuente que no se cumpla que $f^{-1}(0) = x$ debido a la existencia de raíces múltiples. Las raíces complejas sólo ocurren en el caso de los polinomios, que es un caso especial de funciones muy utilizadas en la práctica, para los que hay una gran cantidad de métodos.

Las raíces del polinomio $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ son todos aquellos valores x_i que cumplen con $p(x_i) = 0$. Por el teorema fundamental del álgebra se sabe que todo polinomio de grado n con coeficientes reales posee n raíces y que pueden ser reales o complejas, es decir, de la forma $a + ib$, donde $i^2 = \sqrt{-1}$. Siempre que $a + ib$ es raíz de p, también lo es $a - ib$. De esta manera, $p^{-1}(0) = \{x \in C, p(x) = 0\}$ es el conjunto de las raíces de p en el conjunto de los números complejos C y las raíces no son necesariamente distintas.

La limitante natural para todas las técnicas es la cercanía de las raíces. Si hay dos o más raíces entre dos números de punto flotante contiguos, como (x,`nextUp`(x)), no habrá manera de distinguirlas. A lo más las raíces pudieran aproximarse con los límites mismos del intervalo, pero considerando la sensibilidad de los polinomios, esta sería una tarea excesiva para la aritméica nativa de los procesadores.

6.1. Organización del capítulo

Se presenta primero un caso particular: la ecuación cuadrática con su famosa fórmula general . Estas ofrecen sencillez para introducir los problemas de cómputo básicos que se deben considerar en los casos generales. Un programador novato podrá darse cuenta que este problema de apariencia tan sencilla resulta bastante delicado. Además, sirve para introduce técnicas para probar si las rutinas programadas trabajan bien.

Posteriormente se presentan métodos para la aproximación de raíces de funciones generales, comenzando con los básicos: bisección y Newton-Raphson, con algunas variantes para introducir tanto problemas como conceptos.

Entonces se hace un paréntesis para estudiar de manera más detallada el fenómeno de la convergencia de funciones iterativas. Es la parte más abstracta de la aproximación de raíces de ecuaciones y será muy ilustrativa para aquellos lectores con intereses más formales e incluso podría saltarse por aquellos interesados simplemente en los esquemas algorítmicos, sin perjuicio alguno.

Luego se presentan otros métodos generales para aproximar raíces, incluyendo los métodos de carácter teórico con velocidades de convergencia de cualquier orden.

6.2. Polinomios cuadráticos

En los primeros pasos de la enseñanza del álgebra, se ve que el polinomio $x^2 - x(b + a) + ab$ posee las raíces a y b, pues se factoriza como $(x - a)(x - b)$. Pero hay una infinidad de polinomios donde no es tan fácil encontrar los factores y se necesita buscar otros métodos. Utilizar la computadora y graficar es una posibilidad pero no da mucha precisión y es inútil si las raíces se calcularán muchas veces en una rutina intermedia para usar los resultados automáticamente.

Lo más sencillo y que es enseñado desde educación secundaria, es aplicar la fórmula general. Las raíces del polinomio $ax^2 + bx + c = 0$ son

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{6.1}$$

A nivel superior no es correcto decir que no hay raíces cuando el discriminante es negativo.

Si el *discriminante* $b^2 - 4ac < 0$ las raíces son complejas. Por ejemplo, calculando las raíces del polinomio $3x^2 - 4x + 1 = 0$, se obtiene:

$$\begin{aligned} x_1 &= \frac{-(-4) + \sqrt{(-4)^2 - 4(3)(1)}}{2(3)} \\ &= \frac{4 + \sqrt{16 - 12}}{6} \\ &= \frac{4+2}{6} = 1 \end{aligned}$$

y

$$\begin{aligned} x_2 &= \frac{-(-4) - \sqrt{(-4)^2 - 4(3)(1)}}{2(3)} \\ &= \frac{4 - \sqrt{16 - 12}}{6} \\ &= \frac{4-2}{6} = \frac{1}{3} = 0.\widehat{3} \end{aligned}$$

Un ejemplo más con un polinomio con raíces complejas: $4x^2 + 2x + 3 = 0$. De la inspección visual se deduce que si se evalúa en cualquier número real positivo jamás se obtendrá cero, por lo que las raíces deben ser complejas. Sus raíces se obtienen reorganizando la fórmula (6.1) como $\frac{-b}{2a} \pm i\frac{\sqrt{-(b^2-4ac)}}{2a}$, entonces

$$\begin{aligned} x_1 &= \frac{-2 + \sqrt{2^2 - 4(4)(3)}}{2(4)} \\ &= \frac{-2 + \sqrt{4 - 48}}{8} \\ &= \frac{-2 + \sqrt{-44}}{8} \\ &= -\frac{1}{4} + i\frac{\sqrt{44}}{8} = -\frac{1}{4} + i\frac{\sqrt{11}}{4} \\ &= -.25 + .829\,156\,197\,588\,85i \end{aligned}$$

y por tanto la otra raíz es el conjugado complejo de la primera $-\frac{1}{4} - i\frac{\sqrt{11}}{4}$. En este caso, responder con NaN a la raíz de un número negativo no es correcto salvo que sólo se busquen raíces reales.

Todo ha salido muy bien, pero estos polinomios no son como los que aparecen en la práctica. En la vida real los polinomios tienen coeficientes que pueden ser como los siguientes: $-3.0014873x^2 + 328079.345x + 72673562.5673$. Normalmente se resuelven con computadora, por lo que se requiere estudiar con cuidado los problemas potenciales que pueden surgir.

Listing 6.1: Calculando las raíces de una ecuación cuadrática.

```
1  int EcGral(double a, double b, double c,
2                        double *x1, double *x2)
3  {
4      double Raiz, t, Discriminante = b*b - 4.0*a*c;}
5      int SonRaicesReales = SI;

7      if ( Discriminante<0 ) {
8          SonRaicesReales = NO;
9          Discriminante = -Discriminante;
10     }
11     Raiz = sqrt(Discriminante);
12     t = .5/a;
13     if ( SonRaicesReales ) {
14         *x1 = t*(-b+Raiz);
15         *x2 = t*(-b-Raiz);
16     } else {
17         *x1 = -t*b; /* Parte real */
18         *x2 = fabs(t*Raiz); /* Parte imaginaria */
19     }
20     return SonRaicesReales;
21 }
```

Pudiera tratar de resolverse de manera directa. Una posible interfaz para una rutina que trabaje como caja negra[2] y puede ser de la siguiente manera.

Entrada

a,b,c	los coeficientes del polinomio
x1, x2	las direcciones de memoria para las raíces

Salida

Valor de retorno		
SI	x1	Raíz menor
	x2	Raíz mayor
NO	x1	Parte real
	x2	Parte imaginaria

Esta distinción de la parte real y la parte imaginaria puede sustituirse por el tipo de dato `complex`[3] disponible en algunos compiladores, pero esta interfaz es suficientemente sencilla. Un código es el 6.1.

Un intento tan inocente como lo sencillo de la interfaz.

Si `a=0` la función `EcGral` no trabaja, por lo que debe revisarse antes de la llamada si el polinomio es cuadrático. Esto concuerda con la filosofía del lenguaje C: "el programador sabe lo que hace".

La función `EcGral` podría usarse en algo tan sencillo como el código 6.2 que muestra el resultado en la salida salida estándar.

En este caso, la condición es cierta cuando las raíces son reales y la función `EcGral` regresa `SI`. Se ha usado `%g` para que en tiempo de ejecución se decida si se usará o no notación científica.

[2] De manera autónoma, sin interacción con el usuario ni otras rutinas.
[3] Normalmente una estructura con dos campos de doble precisión.

Listing 6.2: Ejemplo de uso de la función `EcGral`.

```
if ( EcGral(a,b,c,&r1,&r2) )
  printf("r1 = %g\nr2 = %g\n",r1,r2);
else {
  printf("r1 = %g + %g\n",r1,r2);
  printf("r2 = %g - %g\n",r1,r2);
}
```

Aunque es un buen ejemplo para un primer curso de programación, esta aplicación de la fórmula general resulta un tanto inocente desde el punto de vista numérico. Se pueden identificar los siguientes problemas potenciales:

1. Si a es casi cero en valor absoluto, el cociente es propenso a pérdida de precisión.
2. Si b es mucho mayor en comparación con a y c, la raiz del discriminante será cercana a b, por lo que para una de las raíces, el numerador del cociente será $-b + (b + \epsilon)$, con ϵ muy pequeña y tenemos otro problema numérico conocido: la diferencia de cantidades casi iguales o cancelación catastrófica.
3. Si b es aproximadamente igual a $4ac$, se realizará una diferencia de cantidades casi iguales en el discriminante, perdiendo cifras significativas.
4. Alguna o varias de las operaciones pueden causar overflow o underflow.
5. Uno o varios de los coeficientes es $\pm\infty$.
6. Uno o varios de los coeficientes es NaN.
7. Los apuntadores x1 y x2 pueden ser nulos. Este es un problema no numérico, pero la función debe considerar todos los casos.

Como puede apreciarse, el código anterior debe reescribirse si se desea que funcione correctamente. Los primeros tres problemas merecen un mejor análisis y se dejan para más adelante. Los otros son más simples de resolver.

El problema 4 tiene como solución las mismas técnicas empleadas al calcular la hipotenusa como el escalamiento o el reordenamiento cuidadoso de las operaciones *a la Stewart*. Si es imposible evitarlo, debe regresarse 0 o ∞ dependiendo del caso, al igual que en el caso 5.

Los problemas 6 y 7 no tiene más solución que regresar NaN tanto en x1 como en x2. ¿Se debe regresar SI o NO? Con esta interfaz da lo mismo. En todo caso, si el programador desea defenderse de estos casos conviene cambiarla y tener valores de retorno adicionales, posiblemente incluyendo el caso $a = 0$.

Para tener confianza en una rutina es necesario probarla en todos los casos en los que puede fallar. Se continuará con los detalles de la ecuación general de segundo grado inmediatamente después de la siguiente sección.

6.3. Probando que la rutina trabaja bien

Es un momento oportuno para tratar de responder la pregunta: ¿y cómo se puede probar que el cálculo de las raíces es correcto? Probar unos cuantos casos no es sufi-

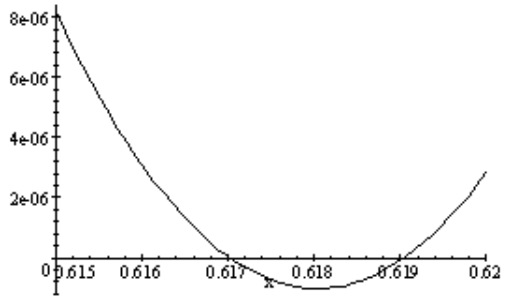


Figura 6.1: Gráfica de $p(x) = 987x^2 - 1220x + 377$. Sus dos raíces requieren una precisión mayor que la que permite la aritmética nativa de una computadora.

ciente. Una manera es evaluar las raíces encontradas en el polinomio y comprobar que el resultado es cero.

Considerando la APF de las computadoras, no es de extrañar al evaluar un polinomio en las raíces encontradas no se obtenga exactamente 0. Por ejemplo, el polinomio $p(x) = 987x^2 - 1220x + 377$, cuya gráfica se muestra en la figura 6.1 tiene dos raíces racionales, $\frac{13}{21}$ y $\frac{29}{47}$, ambas con representación infinita periódica en base 10, indicada con la llave.

$$\begin{aligned} \frac{29}{47} &= .\overbrace{617\,021\,276\,595\,744\,680\,851\,063\,829\,787\,234\,042\,553\,191\,489\,3} \\ \frac{13}{21} &= .\overbrace{619\,047} \end{aligned}$$

Aún cuando no se cometieran errores de redondeo, es necesario ajustarlas a la mantisa de la precisión seleccionada. En caso de ser de doble precisión y un error de conversión de binario a decimal menor a .5 ulp, los valores serían

$$\begin{aligned} r_1 &= .617\,021\,276\,595\,744 \\ r_2 &= .619\,047\,619\,047\,619 \end{aligned}$$

aunque esto resultaría sólo si es necesario escribir el resultado. De evaluarse directamente en el polinomio, se usarán las representaciones internas, sin el error de conversión. Aún al evaluar el polinomio en dichos valores internos que aproximan las raíces no puede esperarse obtener cero pero sería deseable obtener $|p(r_i)| < \varepsilon_M$.

Con las limitaciones de $\mathbb{F}$ se espera un error mayor, dependiendo del grado del polinomio. Por ello es necesario tener claro cuál es la tolerancia aceptable para poder considerar un candidato x_i como raíz. Utilizar el error relativo es tentador pero no es adecuado dividir entre una cantidad tan pequeña. Si se es muy exigente, se puede utilizar el número más pequeño que la norma de IEEE permite para precisión que se está empleando, en este caso doble, por lo que $2^{-52} \approx 10^{-16}$ es una tolerancia que requiere un resultado con la máxima cantidad de dígitos significativos correctos.

El caso en que el error absoluto es más adecuado que el relativo.

Probar rutinas es aún un ar
requiere de mucha práctica
dominarlo.

En general es de gran dificultad probar una rutina. Los experimentos deben ser automatizados, rápidos, orientados a los problemas que se espera encontrar, cuidando de no repetir casos y muy numerosos. En el caso que interesa, para tener la seguridad de su correctitud se tendría que probar todas las posibles situaciones: raíces alejadas, cercanas, raíz con multiplicidad, coeficientes del mismo signo, todas las combinaciones de signos, proporciones diversas, coeficientes y evaluaciones en NaN e Inf, dispersiones aleatorias, direccionar redondeo en búsqueda de anomalías, compara con librerías comerciales y otras más. Por supuesto, cada experimento debe ser reproducible para poder corregir los errores. El esfuerzo para probar bien una rutina es normalmente mucho mayor que el de programarla.

6.3.1. La prueba exahustiva

Una *prueba exahustiva* consiste en llamar a la función con todas las posibles combinaciones de entradas. Siendo simplistas y buscando una cota superior, si los parámetros son variables en precisión doble, cada uno puede tomar como valor cualquier combinación de los 64 bits de su formato, es es decir, uno de 2^{64} posibles. Siendo 3 coeficientes, el total de entradas posibles es de

$$2^{64} \times 2^{64} \times 2^{64} \approx 6.277\,1 \times 10^{57}.$$

Además, debe asegurrse de que para cada entrada la salida es correcta, o al menos tan exacta como la APF permite. La única manera es conocer por anticipado el resultado, pero si se conociera no sería necesaria la función `EcGral`.

Es obvio que incluso en un problema tan pequeño como una ecuación cuadrática, la prueba exahustiva no es viable, tanto por la cantidad de entradas como por ignorar su solución. Lo que sí se puede hacer es evaluar la ecuación cuadrática $p(x)$ en las raíces para comprobar que realmente lo sean. Si r es cualquiera de las dos raíces de p, se espera que

$$|p(r)| \leq |p(\texttt{nextUp}(x)|$$

y

$$|p(r)| \leq |p(\texttt{NextDown}(x)|$$

por monotonicidad, ya que p es de grado 2. La garantía es mayor si se asegura que la distancia entre las dos raíces es de al menos 2 ulp(r).

En polinomios de grado n, las posibles raíces no garantizan esta propiedad.

6.3.2. El método de prueba más sencillo

Siempre existe el procedimiento de fuerza bruta, sencillo pero que no garantiza mucho, por ejemplo el código 6.3, que es el esqueleto de un algoritmo así.

El método trabaja de la siguiente manera. Se generan polinomios seleccionando sus coeficientes aleatoriamente (líneas 2 a 4). Se calculan sus raíces y se evalúan en el polinomio, esperando obtener un valor cercano a cero (líneas 5 y 6). Si en algún caso la evaluación se aleja demasiado del 0 (el error absoluto necesariamente), se graban los coeficientes para analizar después. De ser raíces complejas, caso que no se incluye, puede evaluarse el módulo $\sqrt{x_1^2 + x_2^2}$ y también grabarse cuando sea mayor de cierto umbral, aunque lo más sencillo es descartarlas y sólo considerar el caso real.

Listing 6.3: Intento de prueba exahustiva para el cálculo de raíces de la ecuación cuadrática.

```
1  while ( el usuario no detenga el proceso ) {
2     a=random();
3     if ( a==0 ) continue;
4     b=random();}
5     c=random();
6     if ( EcGral(a,b,c,&x1,&x2) ) { /* Raices reales */
7        r = EvalPol(a,b,c,x1);
8        if ( fabs(r)>Tolerancia )
9           Grabar(a,b,c); /* Para investigar */
10       r = EvalPol(a,b,c,x2);
11       if ( fabs(r)>Tolerancia )
12          Grabar(a,b,c); /* Para investigar */
13    } else {
14          /* Caso complejo */
15    }
16 } /* while */
```

Si luego de dejar corriendo la prueba toda la noche el programador se levanta, detiene el proceso y el archivo de coeficientes para investigar está vacío podrá tener confianza en su función. Esta prueba es la solución para quien no quiere pensar o que no tiene tiempo para probar detalladamente, pero solo ofrece la confianza de que "casi nunca falla".

Es imposible saberlo concerteza.

Se puede defender esta técnica argumentando que es un método estadístico y que por la ley de los grandes números, entre más dure la prueba más seguridad tenemos del nivel de eficiencia de nuestra rutina. En el fondo, lo que se está haciendo es aplicar una versión empírica del método de *Monte Carlo* y sería preferible hacerlo con más cuidado. Por una parte, está bien que los coeficientes se seleccionen aleatoriamente, pues todas las combinaciones se están distribuyendo uniformemente dentro de todas las posibles. Pero por otra parte, se debe tener cuidado al usar las funciones que generan números aleatorios, pues todas son cíclicas y la longitud del ciclo depende del compilador.

¡Puede ser tan corto como 232 en el caso de algunos compiladores!

En general, es preferible dejar Monte Carlo para problemas con más parámetros donde probar es más complicado que la fórmula general. Claro, se puede utilizar como herramienta adicional a las otras pruebas, donde habrá que pensar con cuidado el camino a seguir y que sí puede permitir localizar las distribuciones de coeficientes polinomiales que dan problemas. Para los interesados en esta técnica, dentro de mucha literatura, una buena referencia es el libro de George Fishman [60] o cualquier libro introductorio sobre simulación por computadora. Otra lectura útil es [104] de Kahan sobre el análisis probabilístico de errores.

A continuación se presentan pruebas orientadas de manera más específica.

6.3.3. Polinomios de prueba

Hay muchas pruebas posibles para raíces de polinomios cuadráticos que permiten una mayor confianza. En eneral, lo mejor es construir polinomios cuyas raíces se conozcan por anticipado para poder comparar con las aproximaciones de la fórmula 6.1.

Una manera directa sería partir de las raíces para calcular los coeficientes y después aproximar las raíces. Un método es seleccionar a, x_1 y x_2 arbitrarios (o no del todo) y calcular

$$\begin{aligned} b &= (x_1 + x_2)a \\ c &= x_1 x_2 a. \end{aligned}$$

Los valores a, x_1 y x_2 se pueden generar aleatoriamente o de tal manera que comprometan la precisión de los cálculos, como

$$\begin{aligned} a &\rightarrow 0 \\ x_1 &= 1 - a \\ x_2 &= 1 + a \end{aligned}$$

Motivo por el cual no es de tanta utilidad al evaluar rutinas.

para causar conflictos adicionales o combinarlas de diversas maneras. El mayor problema de esta técnica es que las operaciones para calcular b y c introducen errores no atribuibles a la función que se prueba pero que desafortunadamente no pueden ser aislados. Igual ocurre si se toman como valores arbitrarios iniciales (b, x_1, x_2) o (c, x_1, x_2). Tendrá que tenerse gran cuidado en la forma de realizar cada operación para no agregar problemas de redondeo innecesarios.

Un mejor esquema es generar sucesiones de coeficienes para los que las raíces asociadas converjan a puntos fáciles de determinar. Por ejemplo, si F_i son los números de Fibonacci, tomando $a = F_{n+1}$, $b = -2F_n$ y $c = F_{n-1}$. El discriminante es $F_n^2 - 4F_{n+1}F_{n-1} = 4 \times (-1)^n$, por lo que se van alternando raíces reales y complejas pero que convergen a $2/(1+\sqrt{5})$, pues

$$\begin{aligned} x &= \frac{2F_n + \sqrt{F_n^2 - 4F_{n+1}F_{n-1}}}{2F_{n+1}} \\ &= \frac{2F_n + (-1)^n 4}{2F_{n+1}} \\ &= \frac{F_n}{F_{n+1}} \end{aligned}$$

ya que asintóticamente el 4 no pesa en el cociente y la parte imaginaria de la raíz compleja tiende a cero. Como se sabe, $F_n/F_{n+1} = 1/\tau$, donde τ tiene como límite la razón aurea $\frac{1+\sqrt{5}}{2}$. A la par de resolver la ecuación cudrática, se va calculando el cociente, que se sabe correcto en .5 ulp, y se puede hacer la comparación.

Uno de varios números que aparecen por todas partes.

Otro método es resolver ecuaciones también de la forma $ax^2 - 2bx + c = 0$, seleccionando c y calculando $a = c - 2$ y $b = c - 1$. Las raíces se calculan de manera exacta.

$$\begin{aligned} r_{1,2} &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\ &= \frac{2(c-1) \pm \sqrt{(2(c-1))^2 - 4(c-2)c}}{2(c-2)} \\ &= \frac{2(c-1) \pm 2}{2(c-2)} \\ &= \frac{c - 1 \pm 1}{c-2} \\ &= \begin{cases} \frac{c}{c-2} \\ 1 \end{cases} \end{aligned}$$

Entre mayor sea el valor de c, más cercanas estarán las raíces, lo que aporta otro eje de análisis. Si $c = 2^k$ llegamos a la prueba de precisión sugerida por Kahan en [105], en sus observaciones sobre la norma de 1985. Estas dos formas de ecuación cuadrática permiten evaluar con facilidad la exactitud de las respuesas de la rutina programada.

6.3.4. Evaluación de polinomios

Evaluar polinomios o funciones en general tiene muchos problemas, aún para el software profesional. Como se requiere poder evaluar el polinomio en las raíces encontradas, es necesario dedicarle un momento a los detalles. Hay muchas buenas referencias sobre este tema, como [29, 72, 88, 116, 157][4].

El primer intento de todo principiante de programación es utilizar la función `pow` en algo parecido a

```
S=0;
for ( i=0 ; i<n ; i++ )
  S += a[i]*pow(x,i);
```

Desgraciadamente, este procedimiento tan obvio falla por muchos motivos. Exceso de multiplicaciones y el uso de una función que se utiliza cuando se eleva un número real a un exponente real y no el caso de exponente entero como en los polinomios. De hecho, la función `pow` típica de cada compilador evalúa $x^y = e^{y \ln x}$ lo que resulta en un error relativo potencialmente alto, así como un exceso de tiempo de cómputo, pues para las funciones exponencial y logaritmo natural se requiere hacer una reducción de rango y otras transformaciones, terminando con iteraciones de Taylor para mejorar la aproximación. Aún si se programara una función que evaluara ineficientemente x^n, usando $n-1$ productos, sería mejor que las muchas de operaciones que requiere `pow`.

Sirve recordar de nuevo la observación de la página 99 al evaluar 2^n.

La evaluación directa no es la más adecuada pues ax^2+bx+c requiere tres productos y dos sumas, mientras que $(ax+b)x+c$ requiere sólo dos productos y dos sumas. Este es conocido como el *método de Horner*, que se aplica a polinomios de grado n en general[5].

Evaluando el polinomio del inicio del apartado anterior en Excel con este método, sí se obtiene cero en las raíces y no así con el método que usa la función `pow`. En Maple sigue igual pues la aritmética se fijó en 60 decimales de precisión.

¡Y eso que se trata de un polinomio de grado dos!

Para un polinomio de grado n, $p(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$, la evaluación en x por el método de Horner es

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

y un segmento de código para ello es el 6.4

Listing 6.4: Método de Horner para evaluar polinomios.

```
p=a[n];
for ( i=n-1 ; i>=0 ; i--)
   p = p*x + a[i];
```

[4] En [157] se presentan técnicas curiosas que optimizan aún más, pero que son poco empleadas en la práctica (ver también [117]).

[5] Se dice que esta técnica era ya utilizada por los chinos antigüamente, e incluso por Newton, solo que W. G Horner la popularizó en 1819.

y el resultado de la evaluación queda en `p`. Por ejemplo

$$2x^3 - 5x^2 + 3x + 2 = ((2x - 5)x + 3)x + 2$$

donde son 3 sumas en ambas notaciones, pero las multiplicaciones pasan de 6 a 3 en el peor caso y de 5 a 3, en el caso de que para calcular x^3 se use x^2.

El método de Horner puede utilizarse también para evaluar la derivada del polinomio como en el código 6.5 que se muestra a continuación.

Listing 6.5: Método de Horner para evaluar la derivada de un polinomio.

```
d=0.0;
p=a[n];
for ( i=n-1 ; i>=0 ; i--) {
    d = d*x + p;
    p = p*x + a[i];
}
```

El algoritmo para la división entre $x-c$ es el 6.6. En el arreglo `q` quedan los coeficientes.

Listing 6.6: Realizando una división sintética.

```
Residuo=p[n];
q[n]=0.0;
for (i=n-1 ; i>=0 ; i-- ) {
    s = p[i];
    q[i] = Residuo;
    Residuo = s+Residuo*c;
}
```

Los polinomios son entidades sencillas de manipular algebraicamnete. Su integral, producto y hasta el cociente de polinomios son fáciles de programar.

En 1967, Duane Adams presentó en [3] una variante del algoritmo de Horner que además evalua la cantidad $\in$ tal que $|p - p(x)| \leq \in$, donde p es el valor aproximado al final de las iteraciones y $p(x)$ el valor exacto.

Listing 6.7: Método de Horner-Adams para evaluar polinomios con cota del error.

```
p=a[n];
e=fabs(p)/2;
absx=fabs(x);
for ( i=n-1 ; i>=0 ; i--) {
    p = p*x + a[i];}
    e = absx*e + fabs(y);
}
e=u*(2.*e-fabs(p));
```

Se usa el error absoluto necesariamente.

La variable `u` se define como la precisión de la computadora o unidad de redondeo, que es alrededor de 10^{-16} para la doble precisión de IEEE. El valor final `e` cumple con $|\mathtt{p} - p(x)| < \mathtt{e}$. En polinomios de grado alto, este algoritmo es útil para detener la evaluación en cuanto ocurra que $p \approx e$. En ese caso se podrá tener confianza de que x es un cero de $p(x)$, dentro de las limitaciones de $\mathbb{F}$.

Si la evaluación se realiza con números complejos, la última línea debe cambiarse a la cota mayor `e=2.*u*(2.*e-fabs(p))/(1.-2.*u)`. Si se trata de aritmética compleja, en vez de la norma L_2 puede usarse la norma L_1 si el tiempo es un recurso crítico.

Uno de tantas técnicas para acelerar los procesos, cuando la precisión puede relajarse.

6.3.5. Terminando con la ecuación cuadrática

Ya hay una idea más clara de cómo probar una rutina y de cómo evaluar un polinomio de una manera óptima. Por fortuna hay resultados teóricos que nos ayudan a minimizar los errores potenciales en una ecuación cuadrática. Las tres observaciones de problemas de precisión numérica, en forma resumida eran:

- $a \approx 0$
- $b >> 4ac$
- $|b - 4ac| <\in$

Los problemas son salvados con de manipulación algebraica conveniente para la aritmética de punto flotante y algo más. Por ejemplo, el primer punto puede ser abordado multiplicando numerador y denominador por el conjugado del numerador.

$$\begin{aligned} \frac{(-b+\sqrt{b^2-4ac})}{2a}\frac{(-b-\sqrt{b^2-4ac})}{(-b-\sqrt{b^2-4ac})} &= \frac{(-b)^2-(\sqrt{(b^2-4ac)})^2}{2a(-b-\sqrt{b^2-4ac})} \\ &= \frac{2c}{-b-\sqrt{b^2-4ac}} \end{aligned} \tag{6.2}$$

Esta fórmula es muy comunmente utilizada y reduce el primer problema.

Para el segundo, el mejor esquema es calcular la raiz que no da problemas de cancelación en el numerador con $-(b + signo(b)\sqrt{b^2 - 4ac})$. La otra raíz se calcula despejando de la fórmula $x_1x_2 = c/a$.

Un remedio sencillo.

En el caso del tercer problema, el único remedio es utilizar precisión extra si está disponible o hacer uso de precisión múltiple, representando cada valor como la suma de dos variables para duplicar la precisión. De esta manera, será posible calcular $b^2 - 4ac$ hasta con el último bit correctamente redondeado.

a la Dekker [48]

Por lo pronto, una manera sencilla es reducir riesgos de overflow y underflow cuidando la precisión, como el código 6.8 muestra. Esto es exclusivamente para el paso crítico de calcular el discriminante. En este caso se prefiere utilizar la función `scalb` para facilitar algunas cosas.

En caso de que el primer problema se combine nefastamente con alguno de los otros dos, hay que combinar las soluciones planteadas para reducir los problemas.

Como puede verse, la versión inicial de la función `EcGral` debe modificarse bastante para considerar estos casos. En la mayoría de las ocasiones, los programas legibles y sencillos de entender no son tan robustos como aquellos cuyo propósito es la eficiencia.

El costo es la legibilidad.

6.4. Polinomios de grado 3 y 4

Tras haber presentado la solución de los polinomios de grado 2, se podrían presentar las soluciones de los de grado 3 y 4, cuyas fórmulas fueron publicadas por Girolamo

Listing 6.8: Calculando el discriminante con escalamiento.

```
1  double EcGral(a,b,c,x1,x2)
2  {
3      Max = log(MaxDouble)
4      Min = log( MinSubnormal/eps )
5      /* Casos especiales, a=0 y todo eso ... */
6      m = ( Max+Min-max(1,log(a))-max(1,log(c)) )/2;
7      A = scalb(a,m);
8      C = scalb(c,m);
9      P = A*C;
10     Save(ExceptionFlags);
11     SetExceptionFlags(0);
12     B = scalb(b,m);
13     R = B*B;
14     D = R-P;
15         if ( D==R )
16             Recalcular
```

Cardano (1501-1576) en 1545 en su *Ars Magna*, pero son atribuidas a Ludovico Ferrari (1522-1565), Scipione Ferro (1465-1526) y principalmente a Nicolo Fontana Tartaglia (1500-1557), en una época en la que había torneos donde se calculaban raíces de polinomios. Por ello se dice que Cardano y Tartaglia terminaron peleándose por la autoría.

Se llenarían varias hojas más con los detalles de eficiencia numérica debido a las extensas fórmulas, además de los códigos, pero no se ve la necesidad pues la ecuación cuadrática ha sido muy útil para introducir los problemas básicos de las raíces de funciones y polinomios. Sólo se presentan versiones prácticas de las fórmulas. Para mayores detalles consúltese [17] o

Todo polinomio cúbico puede reescribirse de la forma x^3+bx+c con la transformación $x = mx+n$. Las fórmulas para esta forma de polinomio cúbico son las siguientes. Sean

$$\begin{aligned} \gamma &= \sqrt{\frac{c^2}{4}+\frac{b^3}{27}} \\ \alpha &= \sqrt[3]{\frac{-c}{2}+\gamma} \\ \beta &= \sqrt[3]{\frac{-c}{2}-\gamma}. \end{aligned}$$

Las raíces de x^3+bx+c son

$$\begin{aligned} r_1 &= \alpha+\beta \\ r_{2,3} &= -\frac{r_1}{2} \pm i\sqrt{3}\frac{\alpha-\beta}{2}. \end{aligned}$$

Para la ecuación de grado cuatro de la forma $x^4+ax^3+bx^2+cx+d$, sea w una de las raíces del polinomio $w^3-bw^2+(ac+4d)w-a^2d+4bd-c^2$ y

$$\rho = \sqrt{\frac{a^2}{4}-b+w}.$$

Si $\rho \neq 0$ se calculan

$$\alpha, \beta = \sqrt{\frac{3a^2}{4} - \rho^2 - 2b \pm \frac{4ab - 8c - a^3}{4\rho}}$$

y si $\rho = 0$ se calculan

$$\alpha, \beta = \sqrt{\frac{3a^2}{4} - \rho^2 - 2b \pm 2\sqrt{w^2 - 4d}}.$$

Las raíces de $x^4 + ax^3 + bx^2 + cx + d$ son

$$\begin{aligned} r_{1,2} &= -\frac{\alpha}{4} + \frac{\rho}{2} \pm \frac{\alpha}{2} \\ r_{3,4} &= -\frac{\alpha}{4} + \frac{\rho}{2} \pm \frac{\beta}{2}. \end{aligned}$$

De las observaciones del caso cuadrático se desprende que para los casos de grado 3 y 4 se requiere de mucho cuidado para conseguir una buena rutina que calcule las variables.

6.4.1. Polinomios de grado 5 o mayor

En 1799, Ruffini probó que no había fórmulas posibles para los polinomios de grado mayor a 4, pero no tuvo suficiente difusión.

Paolo Ruffini (1765–1822), matemático italiano.

Fue hasta 1824 cuando Abel demostró (de nuevo) que no había fórmulas para grados superiores, por lo que los métodos de aproximación cobraron importancia.

Niels Henrik Abel (1802–1829), matemático noruego.

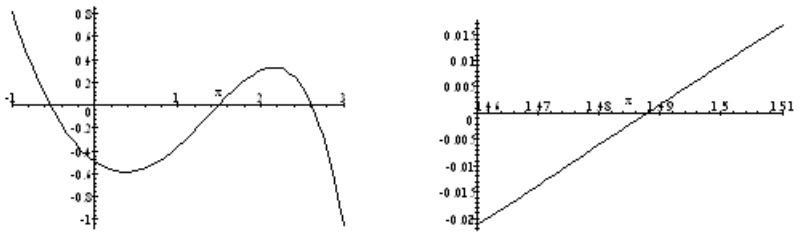
La prueba sólo establece que no existen fórmulas con operaciones elementales y radicales, dejando abierta la posibilidad de otras operaciones. De hecho, sí existen fórmulas para grados mayores que 4, por ejemplo, utilizando funciones hipergeométricas o en términos de las funciones modulares de Siegel. Tales formulaciones no se presentan en este texto.

6.5. Ecuaciones no lineales en general

Para el caso de polinomios generales hay diversos métodos y en general todos son iterativos, es decir, las raíces se van aproximando una por una o todas a la vez. Los métodos más sencillos y generales no son específicos para polinomios, sino que trabajan con funciones de todo tipo. Ningún método es la panacea y que incluso en algunos de ellos, la convergencia no se ha probado en el caso general. En realidad es fácil probar la convergencia y sus condiciones para los algoritmos más sencillos, que en la práctica se utilizan menos.

6.5.1. Método gráfico

La aproximación más sencilla es la del método gráfico. Por ejemplo, $f(x) = x^2 - \frac{e^x}{2}$. Su gráfica se presenta en la figura 6.2a.



a. Función $f(x)$. **b.** Acercamiento de la raíz central.

Figura 6.2: Gráfica de $f(x) = x^2 - \frac{e^x}{2}$ y acercamiento en una de las raíces.

Es posible intentar graficar cada uno de los cruces para aproximar visualmente el resultado. Por ejemplo, en el caso de la función f, podemos graficar sólo el cruce del intervalo $[1.46, 1.51]$, como la figura 6.2b y hay una mejor aproximación. Este método también es útil para saber si la función tiene un cambio de signo en la raíz o si sólo hace contacto tangencialmente con el eje x, lo cual es de gran importancia al momento de decidir el método que se empleará, ya que las raíces con multiplicidad dan muchos problemas a algunos métodos, además de aquellos que sólo trabajan cuando ocurre en cambio de signo.

Problemas del método gráfico

Además de no funcionar con raíces complejas, el método gráfico tiene otros problemas adicionales.

1. Es poco exacto. Principalmente si la raíz se necesita con muchas cifras de precisión.
2. Pueden ocurrir ciertos engaños visuales.

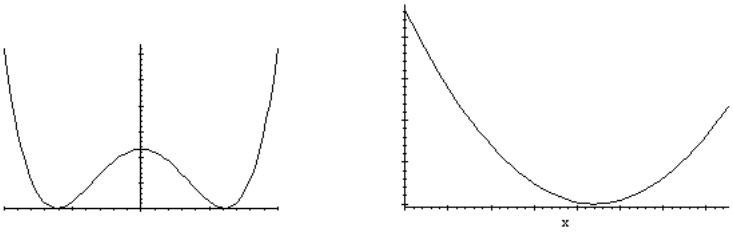
Ejemplo. $x^4 - 3.02x^2 + 2.3101$ es el polinomio de la gráfica 6.3. Aparentemente hace contacto en dos puntos (gráfica 6.3a), pero graficando la función en un intervalo pequeño centrado en uno de los contactos, se ve que no es así (gráfica 6.3b).

O funciones problemáticas como $(x - 3.15)^7 = x^7 - 22.05x^6 + 208.3725x^5 - 1093.955625x^4 + 3445.960218\,75x^3 - 6512.864\,813\,437\,5x^2 + 6838.508\,054\,109\,375x - 3077.328\,624\,349\,218\,75$, cuya gráfica en $[2.6, 3.6]$ es la de la figura 6.4.

De la figura 6.4a no es claro que 3.15 sea la raíz, aunque sólo por la forma se espera que sea múltiple. Más aún, aunque los cálculos son precisos y que las raíces se obtienen con exactitud, la gráfica en $[2.95, 3.35]$ se muestra en la figura 6.4b y la raíz pareciera estar cerca 3.02, lo que no corresponde con la función. Esta gráfica puede resultar distinta si se calcula con otra computadora u otro graficador[6].

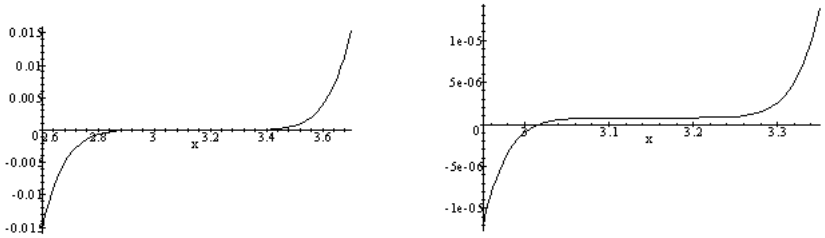
Por todos los motivos anteriores, la gráfica de una función no es el mejor método para localizar raíces. Su uso se restringe a tener una idea inicial de la posición de las

[6]En este caso se uso un motor de Maple.



a. En $(-2, 2)$. **b.** En (1.22,1.235).

Figura 6.3: Función con dos raíces aparentes. Un acercamiento deja ver la realidad.



a. En $(2.6, 3.6)$. **b.** En $(2.95, 3.35)$.

Figura 6.4: Raíz con multiplicidad. Representan un verdadero problema por su sensibilidad. El graficador empleado se equivoca con la verdadera raíz ($r = 3.15$).

raíces reales, en caso de que existan, pues puede haber sólo raíces complejas, como en la función $f(x) = x^2 + 1$. Si obtener las raíces de un polinomio es una parte intermedia en un programa de cómputo, este método no tiene nada qué hacer.

6.5.2. Geometría de las raíces de ecuaciones

¿Cómo cruzan las ecuaciones el eje de las abscisas? En el caso de la matemática continua con exactitud infinitamente precisa no hay duda: exactamente en un punto es cero y alrededor toma valores positivos o negativos. Pero, ¿qué ocurre con aritmética de punto flotante? No puede esperarse que en todos los casos se cumpla que para algún valor x la ecuación sea 0 (o -0.0) exactamente, aunque sí puede ocurir para ciertas ecuaciones y raíces. Vale la pena consderar estos detalles un poco más para comprender mejor la exactitud y precisión que se puede exigir a cada método de aproximación de raíces.

Supóngase primero el caso más simple. Una raíz simple aislada (suficientemente alejada de otras raíces). En el caso ideal ocurrirá que, siedo $r \in \mathbb{F}$ un NPF, $f(r) = 0$ y que en cualquier otro punto $x \neq r$ en una vecindad de r ocurra que $f(x) \neq 0$. En ese caso ocurrirá que

$$\begin{aligned} |f(\texttt{nextUp}(x))| &> 0 \\ \text{y} & \\ |f(\texttt{NextDown}(x))| &> 0. \end{aligned}$$

Evidentemente este es un caso discreto ya que en medio de x y $\texttt{nextUp}(x)$ no existe ningún otro. Se espera que $|f(\texttt{nextUp}(x))| \geq t$, pudiendo ser t al menos el menor número subnormal, lo que dependerá de parámetros internos y externos al problema, como los de la siguiente lista.

- La pendiente o inclinación de f, es decir, del valor de $f'(r)$[7],
- el número de condición de f,
- la estabilidad del algoritmo empleado para evaluar f,
- la precisión interna (intermedia) de los cálculos y
- que sí ocurra la fortuna de este caso ideal.

f con poca inclinación.

En el caso de que $f'(r) < 0.5$ (y la condición de f, la estabilidad en su evaluación y la precisión intermedia empleada) es de esperarse que también $f(\texttt{nextUp}(x)) = 0$. Estrictamente se puede decir que $\texttt{nextUp}(x)$ es una raíz pues cumple con la condición suficiente, pero numéricamente se sabe que se debe a la precisión limitada de la computadora.

La raíz $\texttt{nextUp}(x)$ es artificialmente producida por las imperfecciones nativas de la APF.

X X X Completar X X X

6.5.3. Métodos iterativos: Generalidades

Existe una gran cantidad de algoritmos para aproximar raíces de polinomios y de ecuaciones en general. Algunos son de propósito general, otros especializados en cierto

[7] También se le llama geometría de la función.

Valor de k	Convergencia
1	Lineal
$1 < k < 2$	Superlineal
2	Cuadrática

Tabla 6.1: Órdenes de convergencia más comunes.

tipo de raíces y otros provienen de combinar métodos sencillos. Su eficiencia varía y están igualmente limitados por la precisión numérica de la computadora. Se presenta a continuación un concepto con el que se califica a los métodos de aproximación.

Velocidad de convergencia

Además de la precisión y estabilidad con que un algoritmo numérico debe trabajar, hay un concepto igualmente importancia, que es el de la complejidad temporal, es decir, qué tan rápido se acerca un algoritmo a la solución. No hay duda de que es un tema importante, pero desafortunadamente en muchos textos se hace demasiado énfasis en esto[8], como si fuera más importante que la precisión o la estabilidad. De cualquier forma es una medida más de la eficiencia de todo algoritmo y debe ser tomada en cuenta.

Se define entonces el concepto de *velocidad de convergencia*. Si $\{x_i\}$ es una sucesión de números cuyo límite es r, y si se cumple que

$$|x_{n+1} - r| \leq \alpha \, |x_n - r|^k \tag{6.3}$$

para algunas constantes $0 < \alpha < 1$ y $k > 1$, se dice que la sucesión tiene orden de convergencia k. Algunos autores además reportan la convergencia de $|x_{n+1} - x_n|$ o de la longitud[9] del intervalo $[a_n, b_n]$ que contiene a x_n, pero (6.3) es más común.

Los órdenes de convergencia más comunes son los de la tabla 6.1.

A manera de ejemplo, varios algoritmos con órdenes de convergencia 1.4142, 1.4892, 1.5537, 1.5874 y 1.618 se presentan en [5]. La velocidad de convergencia superlineal es la que normalmente se espera obtener en la práctica, como afirma Kahan en [108].

Existen algoritmos con órden de convergencia superior, como cúbica, cuártica y demás, pero en general no tienen aplicaciones prácticas, salvo distinguidas excepciones. Algunos se presentan en las siguientes secciones.

Tipos de Algoritmos

Sin pretender hacer una clasificación completa de los algoritmos que buscan ceros de funciones o raíces de polinomios, sí se pueden determinar algunas características que algunos tienen en común. Además de agruparlos por su orden de convergencia, se puede hablar de algoritmos

[8] Este es un fenómeno social. Un síntoma más familiar es la forma de vender (o comprar) una computadora, pues preocupa su velocidad y tamao de memoria, nadie preguntar si cumple o no con las especificaciones de la norma de punto flotante. Hasta algunas empresas que realizan comparaciones entre equipo (benchmarks) adolecen de este mal.

[9] Casi siempre se refieren al diámetro del intervalo, entendido como una bola de radio $|b_n - a_n|$, que tiene una concepción más general.

- con y sin memoria,
- que van encerrando la raíz,
- con convergencia global o local,
- simples y compuestos y
- especialistas en raíces múltiples.

En el caso especial de polinomios:

- con coeficientes reales y complejos, y
- aproximacón de una o varias raíces a la vez.

Puntos fijos

Dado el problema de resolver la ecuación $f(x) = 0$, es decir, encontrar r tal que $f(r)$ sea cero, lo más común es buscar alguna función ϕ que cumpla que

$$\phi(r) = r \tag{6.4}$$

y a r se le conoce como punto fijo[10]. Encontrar ϕ a partir de f siempre es posible, de hecho, el problema es que ϕ no es única por lo que se han desarrollado diversas técnicas para encontrar la más adecuada.

Lo que se sabe es que si $f : [a, b] \to [a, b]$ y f es continua, entonces existe un punto x_0 que cumple con (6.4).

Este concepto lo formalizó Brouwer en 1912 en el teorema que lleva su nombre, aunque su formulación original es para el caso más general de conjuntos compactos convexos no vacíos en $\mathbb{R}^n$.

Se presenta a continuación la teoría básica para poder continuar discutiendo métodos de aproximación de raíces con más herramientas.

Criterios de paro

Si se alcanza la convergencia, es obvio que las evaluaciones

$$\begin{aligned} |x_{n+1} - x_n| &< \varepsilon_1 \qquad \text{y} \\ |f(x_{n+1})| &< \varepsilon_2 \end{aligned} \tag{6.5}$$

son determinantes pues al cumplirse alguna de ellas se detienen las iteraciones con la confianza de que x_{n+1} es la mejor aproximación posible de r, ante las limitaciones de la precisión de las computadoras y el hecho de que es preferible el error relativo que el absoluto.

Otra forma menos deseable de terminar las iteraciones es llegar al máximo permitido. Por ejemplo, al iterar con Newton, si en 50 iteraciones no se ha alcanzado alguno de los criterios 6.5. Pero, ¿Por qué no detenerse a las 30? ¿O a las 20? Después de todo, Newton es rápido cuando converge.

Revisar trabajo de Oleksandr Mikolaiovich Sharkovsky,1936 Kiev, Ukraine.

cen en todos los cursos y li-
e Métodos numéricos.

Los dos más básicos son el de bisección y el de Newton-Raphson. De estos derivan una gran cantidad de métodos y por ello se discuten con mayor detalle que otros.

6.5.4. Método de bisección

Este es un método útil en muchas situaciones, aunque su convergencia no es muy alta.[11] Es importante notar que este algoritmo (y muchos de los que siguen) se presenta sin considerar que las raíces sean de polinomios.

Si f es una función continua en el intervalo $[a, b]$ y si $f(a)f(b) < 0$, entonces f debe tener un cero en (a, b) por el teorema del valor medio. Se calcula el punto medio $x = \frac{1}{2}(a + b)$ y se revisa si $f(a)f(x) < 0$. Si es así, se hace $b = x$, de lo contrario, se hace $a = x$ y el proceso se repite. Hay dos criterios de paro importantes:

1. $|f(x)| < \varepsilon$, con $\varepsilon > 0$ un número que representa nuestra tolerancia o error aceptable. Este criterio surge de la definición misma de raíz pues lo que se busca es un número que cumpla con $f(x) = 0$.
2. $|b - a| < \delta$, con $\delta > 0$ otro número pequeño que determina el tamaño del menor intervalo donde buscaremos a la raiz. Eso significa que obtendremos una aproximación a la raíz verdadera r, en el intervalo $(r - \frac{\delta}{2}, r + \frac{\delta}{2})$. Al mismo tiempo, este criterio fija la cantidad mínima de iteraciones que se necesitan, pues se puede calcular considerando que

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

y despejando se obtiene que n debe ser tal que

$$n > \left| \log_2(\frac{|b - a|}{\delta}) \right|.$$

En los dos criterios, ε y δ no son necesariamente distintos. Pueden estar determinados por un error relativo para el intervalo inicial.

Orden de convergencia de la bisección

Si $[x_n]$ es la sucesión de aproximaciones a la raíz exacta r, entonces se cumple

$$|x_{n+1} - r| \leq \frac{1}{2} |x_n - r|$$

por lo que el orden de convergencia es lineal.

[10] Es un concepto central en sistemas dinámicos, alrededor del cual hay gran cantidad de resultados, desde puramente teóricos hasta control, pasando por ecuaciones diferenciales y sistemas lineales.

[11] Esta anotación de convergencia "lenta" es principalmente de carácter teórico. Con la velocidad de las computadoras moderas y la eficiencia de los compiladores, al aproximar un cero de una función las iteraciones terminan antes de levantar el dedo del ¡ENTER¿ o del botón del mouse. El problema es cuando se requieren muchos miles de aproximaciones dentro de un proceso de cómputo mayor. Además, no hay criterios automáticos para calcular los parámetros iniciales en el caso general. De otra manera, este sigue siendo un método robusto.

Listing 6.9: Método de bisección para aproximar ceros de ecuaciones (versión simple).

```
1  double Bisec(double a, double b)
2  {
3     double r;
4     if ( f(a)*f(b) > 0 )
5        return 0;
6     while ( 1 ) {
7        r = (a+b)/2;
8        if ( fabs(f(r))<1e-13 )
9           break;
10       if ( f(a)*f(r)<0 )
11          b=r;
12       else
13          a=r;
14    }
15    return r;
16 }
```

No es un mal orden de convergencia si consideramos la velocidad de las computadoras modernas, pero no termina de agradar que la sucesión de aproximaciones $\{x_i\}$ no se acerca a r en cada iteración. Por ejemplo, si la raíz es $r = 5$ y se comienza en el intervalo $[1, 10]$, $x_0 = 5.5$ y el siguiente intervalo será $[1, 5.5]$. La primera aproximación quedó a .5 de distancia de r. En la segunda iteración se hace $x_1 = (1 + 5.5)/2 = 3.25$, con lo que la segunda aproximación queda a 1.75 de distancia de la raíz.

En algunas aplicaciones esto es más serio que la convergencia lineal ya que se usa el criterio de continuar iterando mientras mejore la aproximación no se puede aplicar.

Aspectos de la programación

El código 6.9 muestra directamente el algoritmo presentado. Se supone que f es la función cuya raíz r interesa y que se sabe está en el intervalo $[a, b]$.

La condición de la línea 4 es para comprobar que el cambio de signo se cumple. Este intento es muy legible pero tiene muchas cosas que criticarle, como el exceso de evaluaciones[12] de la función f o el hecho de la precisión fija 10^{-13}. ¿Y si la raíz que se aproxima está en el intervalo $[0, 10^{-15}]$? ¿O en $[10^{11}, 10^{12}]$? La precisión fija no resulta adecuada.

En el caso de que por descuido se envíe un intervalo donde no hay raíz, es decir, que se cumpla el primer `if`, la función regresa de imediato pero no hay ninguna señal que indique lo que ocurrió. Además, no hay un límite en la cantidad de iteraciones, en caso de que el algoritmo quede ciclado, que no sería raro tomando en cuenta la aritmética imprecisa de la computadora.

Otra de las primeras cosas que se debe notar es que no hay necesidad de calcular el producto $f(a)f(x)$, sino sólo revisar que los signos sean opuestos. Eliminar un producto

[12] La cantidad de evaluaciones de la función de interés durante cada iteración es un parámetro importante al comparar los métodos iterativos. La convergencia en la mitad de las iteraciones puede perder sentido si se realizan más del doble de evaluaciones.

Listing 6.10: Método de bisección para aproximar ceros de ecuaciones (versión no tan simple, pero aún incompleta).

```
double Biseccion(double a, double b, int MaxItera)
{
    int i;
    double fa=f(a), fr, L, Epsi, r;
    if ( Signo(fa) == Signo(f(b)) )
      return NaN; /* Para no usar la raiz */
    if ( b<a) {
      int t=a;
      a=b;
      b=t;
    }
    L = b-a;
    Epsi = Epsilon((b+a)*.5);
    for ( i=1 ; i<MaxItera ; i++ ) {
      L *= .5;
      if ( L < Epsi ) /* Intervalo muy pequeno */
         break;
      r = a + L;
      fr=f(r);
      if ( fabs((fr-fa)/fr)<Epsi )
         break;
      if ( Signo(fr) != Signo(fa) )
         b = r;
      else {
         a = r;
         fa = fr;
      }
    }
    return r;
}
```

en cada iteración (o sustituirlo por una verificación de signo) es una reducción de tiempo significativa, principalmente en el caso de funciones complicadas.

La mejor manera de calcular el punto medio de un intervalo es sumar al límite inferior la mitad de su longitud actual, como $a + t/2$, donde t es la longitud del intervalo. Ambos números pueden almacenarse en variables adecuadas desde la primer iteración. Un mejor intento es el código 6.10.

Se han resuelto algunos de los problemas de la primera versión y las cosas lucen mejor, aunque de 11 instrucciones se pasó a 22 y ya no es tan legible. Se agregó a los criterios de paro un contador para la cantidad de iteraciones, por si acaso algo sale realmente mal y el método se queda en un ciclo infinito. Además, no se repiten evaluaciones, se introdujo el error relativo como medidad de paro y la función f se evalúa una sola vez cada ciclo. Al regresar NaN no hay malos entendidos cuando la condición inicial no se cumple.

Pero los dos primeros argumentos de la función son valores flotantes en el formato de IEEE754R, ¿qué pasa si $b = \infty$ o NaN? Los parámetros normalmente vendrán de algún otro procedimiento y pocas veces los escribirá el usuario desde el teclado. Por la norma 754 de IEEE, es posible que la evaluación de una función de estos

resultados, por lo que al comparar el signo de $f(a)$ y $f(b)$ pudiera obtenerse una respuesta aparentemente adecuada.

Aún así, al intento anterior se le puede criticar su interfaz. Para efectos de evaluación, pudiera ser útil conocer el estado en que terminaron las evaluaciones, si se llegó al máximo de iteraciones o cuántas fueron, la precisión calculada o si en algún momento la condición de cambio de signo dejó de cumplirse.

Por otra parte, el error relativo es muy útil pues asegura una cantidad fija de cifras significativas correctas independientemente de la magnitud del intervalo, pero si la raíz es cercana al cero, la división se vuelve sumamente inestable y falta un criterio para ese caso. Como puede verse, programar bien un algoritmo numérico no es tan sencillo. Una versión mejorada de la anterior que considerara todas las posibilidades que dicta la norma casi triplica la cantidad de instrucciones y no se incluye en el texto.

La función `Epsilon`, que calcula una tolerancia adecuada para el intervalo inicial, puede ser algo como el código 6.11, donde se asegura una precisión de 16 decimales para el valor `A`.

Listing 6.11: Cálculo de un épsilon adecuado.

```
1  double Epsilon(double A)
2  {
3     double act=A;
4     while ( act/A > 1e-16 )
5        act *= .5;
6     return ant;
7  }
```

En caso de trabajar con simple precisión, la constante puede ser 10^{-8} en vez de 10^{-16}. Si se trabaja con cuádruple precisión, se puede emplear 10^{-34}. Si se supone que la computadora respeta la norma de IEEE, podría indicarse el número sin hacer cálculos.

Un buen ejercicio para el lector.

Bisección tendrá problemas en una función aparentemente simple como la de la figura 6.5. La forma parece indicar que hay una raíz cerca del cero con multiplicidad ¿0 y que el cambio de signo ocurre, por lo que con cierta confianza se puede a comenzar a iterar en el intervalo $[0, 2]$.

Una vez que falle el algoritmo y amalice con mayor cuidado la función, sale a la luz la realidad. Primero en la figura 6.6a, donde se aprecia que son en realidad tres raíces, una en $[.6, .8]$ otra cerca de 1 y otra cerca de 1.5. Ahora pudiera creerse conocer mejor a f e iniciar con un intervalo seguro. No habrá problemas con las raíces de los extremos, pero de nuevo fallará el método en la central. En la figura 6.6b, la gráfica en $[1, 1.0004]$ evidencia otras dos raíces más[13]. Es por ello que seleccionar adecuadamente el intervalo incial es tan crítico .

Aprovéchese para observar en la gráfica 6.6b la repetición de abcisas por redondeo del graficador empleado.

Aunque bisección sea un método robusto, verificar que las condiciones de convergencia se cumplen requiere cierto cuidado.

Mejorando la eficiencia de bisección

Puede pensarse que la división del intervalo en dos partes es muy limitada y que tal vez sea mejor idea subdividir en tres o más, para luego buscar en qué subintervalo

[13] Podría ser un cuento de nunca acabar: el método gráfico ataca de nuevo.

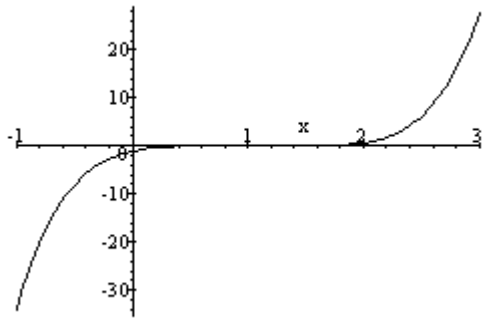


Figura 6.5: Gráfica de una función de apariencia inocente.

ocurre el cambio de signo. La mejora en la aproximación más veloz de la raíz se pierde por la necesidad de bucar el subintervalo indicado. La única forma aparente de mejorar el desempeño del método de bisección es utilizando paralelismo.

Si se cuenta con $m+1$ procesadores, se puede subdividir el intervalo en m partes y delegarle a cada uno de los procesos la verificación de cambio de signo. Se espera que uno solo responda. La idea suena sencilla pero en la práctica reslta un tanto inocente, debido a que en la típica arquitectura con muchos procesadores, cada procesador trabaja con su propia memoria local.

Si el intervalo $[a, b]$, se subdivide en m partes (suponiendo un procesador que dirige o controla a los demás), al procesador i le tocará el subintervalo

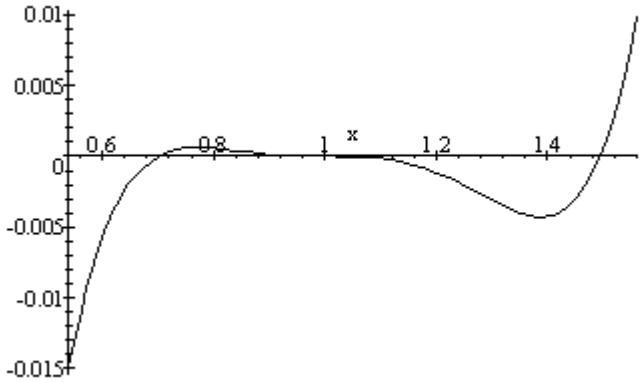
$$[a + (i-1)\frac{b-a}{m}, a + i\frac{b-a}{m}]$$

y deberá evaluar a la función en ambos límites para verificar el cambio de signo. Como cada procesador utiliza su propia memoria local, cada uno repetirá la evaluación considerando como límite inferior el límite superior del anterior, pues es más veloz que transferir el resultado.

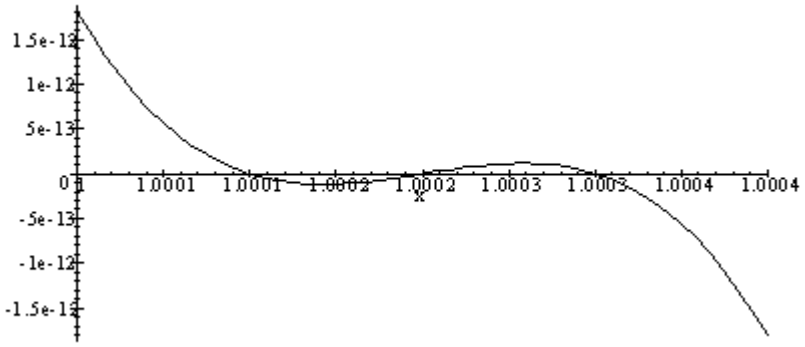
El procesador pricipal debe enviar a cada proceso los límites del intervalo y la función que debe evaluar, por lo que si son muchos, cuando el último procesador esté recibiendo el intervalo $[a+(m-1)\frac{b-a}{m}, b]$, posiblemente muchos habrán terminado su pequeña tarea de dos evaluaciones y una comprobación de signo.

Una forma más eficiente puede lograrse si los procesadores comparten memoria la memoria principal. Las operaciones pueden organizarse de tal manera que al inicio de las iteraciones, a cada procesador se le asigna su propia i que indica qué subintervalo le corresponde, independientemente del intervalo en que se esté. Cada procesador i evalúa la función sólo en su límite superior y almacena el resultado, señala que evaluó con una bandera y queda en espera de la bandera del procesador $i-1$. En cuanto se puede comprobar el cambio de signo, se envía un mensaje al procesador prinncipal, que deberá cambiar el intervalo inicial. De esta manera no se repiten evaluaciones.

Es posible que exista más de un cero en el intervalo inicial, ya sea porque la función es engañosa (como la figura 6.5) o por inexactitudes numéricas, por lo que



a. En $(0.6, 1.52)$.



b. En $(1, 1.0004)$.

Figura 6.6: El acercamiento a las raíces de la función de la figura 6.5 en la página 159 aclara la situación. No se puede confiar en el método gráfico.

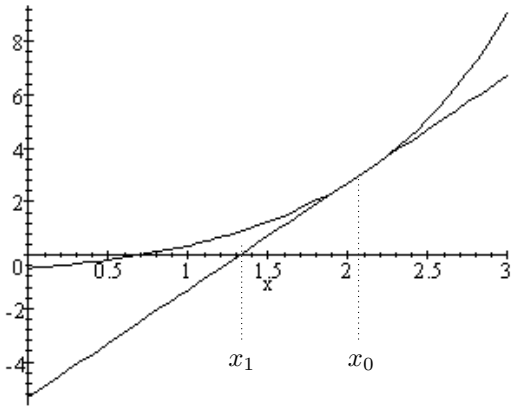


Figura 6.7: Método de Newton: el punto de cruce de la recta tangente, $(x_0 - \frac{f(x_0)}{f\prime(x_0)}\ ,\ 0) = (x_1, 0)$ es la siguiente aproximación.

se debe preveer estos casos. Se puede aplicar el algoritmo a cada subintervalo donde aparentemente hay una raíz o reportar el evento y detener las iteraciones.

6.5.5. Método de Newton-Raphson

Este es un método poderoso para la búsqueda de raices, que se presenta sin considerar que se trate de polinomios. Fue propuesto por Raphson en 1690, en su libro "Analysis aequationum universalis".

Joseph Raphson, (1648 – 1715), matemático inglés.

Se publicó también como trabajo de Newton en 1736, pero escrito en 1671. Se dice que Newton se limitó a aplicarlo a polinomios[14]. Posteriormente fue estudiado por otros matemáticos, por ejemplo Simpson, en 1740, quien vio la conexión con las derivadas, según [168], o Mourraille en 1768, que se percató de lo sensible y difícil de determinar una buena aproximación inicial.

Isaac Newton, (1648 – 1715), matemático inglés.

Consiste en iniciar con una aproximación x_0 cercana a la raíz r y calcular el cruce en el eje x de la recta tangente a la función en x_0, como muestra la figura 6.7.

Una forma de deducirlo es a partir del polinomio de Taylor. Si la función f posee una raíz en el valor r y que es continua hasta al menos la segunda derivada en una vecindad de r. La aproximación de $f(x)$ para una x en la vecindad de r viene dada por

$$\begin{aligned} 0 &= f(r) = f(x+h) \\ &= f(x) + hf'(x) + O(h^2). \end{aligned}$$

De aquí se deduce que $h = r - x$. Al eliminar los términos de segundo grado, obtenemos que $h = -f(x)/f'(x)$. Ahora, si x es próximo a r, entonces $x - h$ debiera estar más próximo aún. De esta manera, obtenemos un procedimiento para aproximar

[14] De hecho, usa la función $x^3 - 2x - 5 = 0$ como ejemplo en su *"Methodus fluxionum et serierum infinitarum"* por 1669, sin utilizar explcitamente el concepto de derivada.

la raíz a partir de un primer candidato:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{6.6}$$

obtenida con el supuesto de que la derivada no es cero en la vecindad de la raź y suponiendo que h es pequeña, es decir, la aproximación inicial es buena. También puede verse (6.6) como la función $g(x) = x - f(x)/f'(x)$, que es una función de $\mathbb{R}$ en $\mathbb{R}$, es decir, un endomorfismo.

Otra manera de deducir la misma fórmula (6.6) es geométricamente a partir de la figura 6.7.

Para simplificar expresiones, de ahora en adelante se usará también la notación (común en muchos textos, como el de Traub [179])

$$u(x) \;=\; \frac{f(x)}{f'(x)} \tag{6.7}$$

o simplemente

$$u \;=\; \frac{f}{f'}$$

ya que aparecerá constantemente en muchas expresiones. Este término es conocido como *corrección de Newton*. La iteración de Newton se escribe en forma más simple:

$$x_{n+1} \;=\; x_n - u(x_n).$$

Una desventaja del método es que requiere la evaluación de la derivada de f en cada iteración y es muy común que las derivadas sean funciones mucho más complicadas que la original. Una táctica útil en estos casos es actualizar la derivada cada k iteraciones para no realizar tanto cálculo.

Condiciones de convergencia

Las aproximaciones generadas por el esquema iterativo (6.6) son una sucesión de puntos en $\mathbb{R}$ que cumple $|x_{n+1} - r| < |x_n - r|$ si x_0 está suficientemente cerca de r. Retomando la terminología de la sección 1.4, si se encuentra un intervalo $[a, b]$ donde la única raíz sea r y una aproximación inicial $x_0 \in [a, b]$, si el esquema iterativo (6.6) es contractivo entonces es estará asegurando la convergencia. Hay diversas maneras de lograr esto.

Una es exigiendo que $|f'(r)| > 0$, que f sea doblemente diferenciable en una vecindad de r y si para algún $\delta > 0$ ocurre que

$$\left| \frac{f''(\bar{x})}{f'(x)} \right| < \frac{2}{\delta}$$

para $\bar{x}$ y x en el intervalo $[r-\delta, r+\delta]$ entonces Newton converge y lo hace cuadráticamente.

Mínimamente habría que exigir que f'' sea continua, r sea un cero simple, con lo que siempre será posible encontrar una aproximación inicial, o en otras palabras, un intervalo $[a, b]$ tal que $r \in [a, b]$ y una constante L tal que

$$|x_{n+1} - r| \leq L|x_n - r|^2$$

comenzando desde cualquier punto inicial

La única circunstancia en la que el método converge desde cualquier punto inicial es cuando la función es creciente, convexa y que tenga un cero. Es el caso de funciones como $e^x - 2$.

Una prueba más formal y rigurosa de que este método (y otros similares) converge se basa en el *Teorema de Kantorovich*[15].

Leonid Vitalyevich Kantorovich, (1912-1986), matemático ruso ganador del premio Nobel de Economía en 1975 por sus trabajos en optimización.

El teorema enuncia que si f tiene al menos hasta la segunda derivada continua y satisface la condición de Lipschitz

$$\left| \frac{f'(x) - f'(\bar{x})}{f'(r)} \right| \leq \alpha \left| x - \bar{x} \right|$$

para toda x y $\bar{x}$, con

$$|x - \bar{x}| + |r - \bar{x}| \leq \frac{1}{\alpha}$$

y además $\beta = |u(r)|$, si $\alpha\beta \leq 1/2$ entonces la sucesión x_n generada por $x_{n+1} = x_n - u(x_n)$ (método de Newton Raphson) converge a la solución única r en el intervalo cerrado $[r - \frac{1}{\alpha}, r + \frac{1}{\alpha}]$.

Y todavía más, Kantorovich asegura que si

$$\gamma = \frac{1 - \sqrt{1 - 2\kappa}}{\alpha} \leq \frac{1}{\alpha}$$

entonces la aproximación se alcanza para valores iniciales en el intervalo abierto $(r - \gamma, r + \gamma)$, donde para el método de Newton[16], $\kappa < 1/2$. Otra buena referencia es [190].

Este método ha sido muy estudiado y ha permitido que se determinen condiciones menos estrictas que también garanticen la convergencia. Por ejemplo, mientras que el teorema de Kantorovich exige la condición de Lipschitz

$$|f'(x) - f'(y) \leq L|x - y| \tag{6.8}$$

en [80], Gutiérrez y Hernández muestran que es suficiente con

$$|f'(x) - f'(x_0)| \leq L|x - x_0| \tag{6.9}$$

lo que implica alterar algunas de las otras condiciones, particularmente el tamaño del intervalo donde puede seleccionarse x_0.

Los interesados en estos temas no deben dejar de buscar los teoremas de Miranda, Moore y Borsuk, que imponen criterios distintos para la existencia de ceros de funciones, generalmente formulados en espacios de Banach. Kantorovich resulta un caso especial de una generalización del de Miranda y este del de Borsuk, según Alefeld, Frommer, Heidl y Mayer. En particular los de Moore y Miranda son útiles como pruebas en toda rutina que busque raíces de funciones de una o más variables. Miranda es particularmente útil en el caso de utlizar aritmética de intervalos.

[15] Este teorema ha encontrado aplicacaciones en muchas ramas de las matemáticas, como control, elemento finito, ecuaciones integrales y muchos más, (consúltese [155]) al proporcionar condiciones para la existencia y unicidad de soluciones de operadores no lineales en espacios de Banach, aunque en esta presentación sólo interesa el caso particular de los números reales.

[16] La prueba puede consultarse en [156].

Velocidad de Convergencia de Newton

Se parte de nuevo de la aproximación de Taylor. Sea e_n al error de la aproximación en el paso n, es decir, $e_n = |x_n - r|$, con r la raíz que se busca, $e_0 < 1$. La aproximación con el polinomio de Taylor es

$$\begin{aligned} 0 &= f(r) = f(x_n - e_n) \\ &= f(x_n) - e_n f'(x_n) + \frac{1}{2!} e_n^2 f''(\delta), \end{aligned}$$

con δ algún número entre x_n y r. De aquí se obtiene la relación

$$e_n f'(x_n) - f(x_n) = \frac{1}{2!} e_n^2 f''(\delta). \tag{6.10}$$

Ahora lo que interesa es ver cómo se comporta el error de un paso a otro para poder determinar el orden de convergencia.

$$\begin{aligned} e_{n+1} &= x_{n+1} - r \\ &= x_n - \frac{f(x_n)}{f'(x_n)} - r \\ &= x_n - r - \frac{f(x_n)}{f'(x_n)} \\ &= e_n - \frac{f(x_n)}{f'(x_n)}. \end{aligned}$$

Esta expresión se puede escribir como $e_{n+1} = \frac{e_n f'(x_n) - f(x_n)}{f'(x_n)}$ y, considerando la ecuación 6.10 y que r es cercano a x_n, ahora se puede escribir como

$$\begin{aligned} e_{n+1} &= \frac{1}{2} \frac{e_n^2 f''(\delta)}{f'(x_n)} \\ &\approx \frac{1}{2} \frac{f''(r)}{f'(x_n)} e_n^2 \\ &= C e_n^2 \end{aligned}$$

para alguna constante C, por lo que se concluye que el método de Newton posee convergencia cuadrática. Esto significa que en cada iteración se duplica la cantidad de cifras significativas correctas.

La C es la constante de Lipschitz (véase la página 23) si se ve la sucesión de errores como los puntos generados por una función. Es suficiente con que $\frac{1}{2} \frac{f''(r)}{f'(x_n)} < 1$ para que sea un mapeo contractivo y sea posible la convergencia.

Además, si f y g tienen como raíz a r, son convexas en una vecindad de r y son dos veces diferenciables con primera derivada no nula en r, entonces la iteración de Newton (6.6) converge más rápido en la función con mayor convexidad logarítmica L_f. Al no tener la raíz desde el inicio de las iteraciones, L_f puede evaluarse con las aproximaciones x_n.

Listing 6.12: Método de Newton-Raphson para aproximar ceros de ecuaciones.

```
1  double Newton(double x, int MaxItera, int *Itera)
2  {
3    int i;
4    double Epsi, fx, dfx;
5    Epsi = Epsilon(x);
6    for ( i=1 ; i<MaxItera ; i++ ) {
7      fx = f(x);
8      dfx = df(x); /* la derivada */
9      x = x - fx/dfx;
10     if ( fabs(fx) < Epsi )
11       break;
12   }
13   *Itera = i;
14   return x;
15 }
```

Aspectos de la programación

El código más directo de una función como Newton es algo como el código 6.12, en el que la función recibe la aproximación inicial, la cantidad máxima de iteraciones y la variable donde quedará cuántas iteraciones fueron necesarias.

Es claro que el código aplica el método de Newton tal y como fue presentado, pero generalmente esto no es del todo eficiente o seguro. Por ejemplo, la fórmula de actualización pudiera dar un cociente entre cero no previsto, por lo que hay que tomar precauciones y cambiar la fórmula por al siguiente segmento de código.

Puede considerarse algo arriesgado utilizar `Epsi`, pero el otro remedio es simplemente abortar las iteraciones indicando que no se llegó a la aproximación requerida.

```
if ( fabs(dfx) > Epsi )
  x = x - fx/dfx;
else
  x = x - fx/Epsi;
```

Por otra parte, el único criterio de paro de si la función es casi cero no es suficiente en el caso general, conviene ir revisando si la actualización actual es muy cercana a la anterior para detenerse cuando ya no se está mejorando sensiblemente. Es equivalene al criterio del tamaño del intervalo en bisección, donde si el intervalo es suficientemente pequeño se detiene el proceso. Para esto, es necesario mantener una variable con la aproximación anterior para poder comparar.

Comparando los métodos de Bisección y Newton, la tabla 6.2 muestra las iteraciones para la función $(x-1)(x-10)$, buscando aproximar la raíz 10. Bisección comienza con el intervalo $[9, 10.5]$ mientras que Newton parte de la aproximación inicial 10.5. Se muestra la aproximación correspondiente.

Como puede verse, Bisección tardó 32 iteraciones en llegar a la precisión relativa indicada en el programa, mientras que Newton la superó en sólo 4 pasos. En este caso, Bisección ocupó el 96 % del tiempo de ejecución mientras que Newton sólo el 4 %, sin tomar en cuenta el resto del código[17]. El método de Newton comienza a tener

[17] Los porcentajes hacen referencia a la proporción de tiempo de ejecución utilizado por cada una de las dos rutinas, sin contar el resto del programa.

Iteración	Bisección	Newton
1	9.75	10.0250000000000004
2	10.125	10.0000690607734803
3	9.9375	10.0000000005299245
4	10.03125	10
5	9.984375	-
6	10.0078125	-
7	9.99609375	-
⋮	⋮	-
30	10.0000000004656613	-
31	9.99999999976716936	-
32	10.0000000001164153	-

Tabla 6.2: Tabla comparativa de las iteraciones de Bisección y Newton-Raphson.

Iteración	Bisección	Newton
1	9.75	10.3362068965517242
2	10.125	10.2254672162094131
3	9.9375	10.1509187900252726
4	10.03125	10.100887568334679
5	9.984375	10.0673821864663324
6	10.0078125	10.0449769574305048
7	9.99609375	10.0300094473814365
8	10.001953125	10.0200173671539492
9	9.9990234375	10.0133498436609738
10	10.00048828125	10.0089020916573741
11	9.999755859375	10.0059357048437469
12	-	10.0039575711505133
⋮	-	⋮
18	-	10.0003475106001272

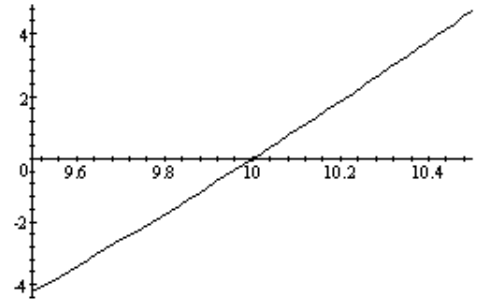
Tabla 6.3: Comportamiento de Bisección y Newton-Raphson ante una raíz múltiple.

problemas cuando las raíces tiene multiplicidad. Por ejemplo, si se cambia la función a $(x-1)(x-10)^3$ el resultado es el de la tabla 6.3. La gráfica de ambas funciones se muestra en la figura 6.8.

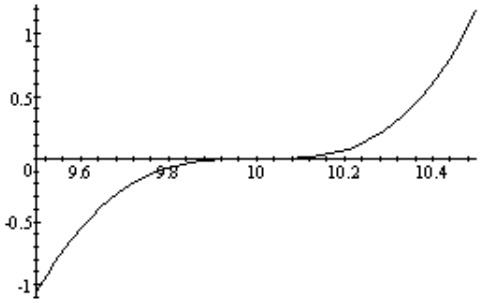
Ahora es Newton el que tarda más iteraciones en llegar a la precisión exigida, aunque los tiempos de ejecución son similares: 48 % de bisección contra 52 % de Newton. Es de esperarse que la precisión sea menor debido a la forma de esta función cerca de la raíz, que ocasiona una reducción de la convergencia de cuadrática a lineal.

Aproximación de la derivada

Uno de los problemas de este veloz método es que requiere de la derivada. Como se sabe, al derivar una función es posible que la expresión resultante sea más complicada. Para evitar este problema, puede utilizarse una aproximación. La forma más simple



a. $(x-1)(x-10)$.



b. $(x-1)(x-10)^3$.

Figura 6.8: Comportamiento de una raíz múltiple. El riesgo de que la derivada sea 0 aumenta con la multiplicidad.

es aproximar la derivada con el cociente de diferencias

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

Este es el origen del método de la Secante, que se verá a continuación.

utilizando cada par de aproximaciones sucesivas. Con este esquema, el denominador se va haciendo cada vez más pequeño y puede haber problemas. No es posible fijar un valor h y utilizar la fórmula

$$f'(x_n) \approx \frac{f(x_n + h) - f(x_n)}{h} \tag{6.11}$$

pues el resultado dependerá del punto donde se calcule. Por el teorema de Taylor, la ecuación 6.11 tiene un error de truncamiento de $-\frac{h}{2}f''(\xi)$, con $\xi \in (x, x+h)$.

Aproximando $f(x+h)$ y $f(x-h)$ y calculando su diferencia se obtiene una mejor aproximación de la derivada:

$$f'(x_n) \approx \frac{f(x_n + h) - f(x_n - h)}{2h}$$

y el error de truncamiento es de sólo $-\frac{h^2}{6}f'''(\xi)$. La iteración de Newton queda

$$x_{n+1} = x_n - \frac{2hf(x_n)}{f(x_n + h) - f(x_n - h)} \tag{6.12}$$

y debe ser claro que la constante $2h$ debe calcularse por anticipado. Esto significa que en vez de encontrar una recta tangente se utiliza una secante que pasa por dos puntos muy próximos a x_n.

No es posible determinar el mejor valor de h para cada x, por lo que se utiliza un proceso iterativo, reduciendo el valor de h hasta llegar a la mejor aproximación posible de $f'(x)$.

Al aproximar la derivada, se está cambiando la fórmula utilizada en el análisis de la velocidad de convergencia, por lo que esta se ve perjudicada. Al hacerdos evaluaciones de función por iteración el orden de convergencia baja de 2 a $\sqrt{2}$, aunque esto depende de qué tan buena sea la aproximación. Lo mejor es tener la expresión de la derivada, siempre que sea posible.

Problemas externos a los métodos

La fuente constante de problemas es la aritmética de punto flotante. Por ejemplo, con el polinomio $(x-1)(x-10)^5$, iterando con bisección con el intervalo inicial $[9.1, 11]$ se llega en 10 iteraciones al intervalo $[9.998048875, 9.99990234375]$. ¡La raíz ya quedó fuera del intervalo! Todo porque la evaluación de un número menor que 10 dio positivo en vez de dar negativo.

Peor aún, los polinomios $p_1(x) = (x-1)(x-10)^3$ y $p_2(x) = (x-1)(x-10)^5$ son negativos en $[9, 10)$ y positivos en $(10, 11]$, por lo que $p(10-h) < 0$ y $p(10+h) > 0$ para toda $0 < h < 1$. ¡Pero ocurre que $p_i(10-h) > 0$ y $p_i(10+h) < 0$ cuando $h =$ .00000 381 469 726 562 5!

Analizando el fenómeno anterior con más cuidado, es evidente que no fallan los métodos de aproximación de raíces, sino la aritmética de la computadora. Una pregunta

interesante es si valdría la pena agregar código a bisección para verificar si en cada iteración se satisface que $f(a)f(b) < 0$. En caso de no cumplirse se podría abortar la aproximación indicando el motivo y regresar la actual mejor aproximación, Por supuesto, a costa del tiempo extra en cada iteración. Esto sólo si se desea que la rutina sea muy robusta y decida las respuestas de manera autónoma.

Continuando con Newton, a menos que se programe una función muy general, es adecuado tratar de simplificar el cociente $u(x) = f(x)/f'(x)$. Si la derivada se aproxima a 0 cerca de la raíz pueden surgir problemas con la convergencia. De hecho, evidentemente no es posible aplicar el algoritmo si f no tiene derivada en una vecindad de r.

Variantes de Newton

Una variante que presentan algunos libros para cuando la derivada es muy compleja es no actualizar la derivada cada iteración, sino decir cada cuántas iteraciones hacerlo. Así, si se actualiza cada tres, el método evaluará la derivada en las iteraciones 1, 3, 6, ... hasta lograr la convergencia. Esto significa que en la primera iteración, se calcula la x_1 a partir de x_0 con una tangente y x_2 y x_3 se aproximan con una secante como en la gráfica siguiente.

Luego, al actualizar la pendiente en la aproximación que está cerca del 1.1, la nueva recta tangente es

y el nuevo corte ocurre en el intervalo $[.4, .6]$.

Parece una buena idea, pero la convergencia no es cuadrática. Además, si la convergencia original es cuadrática, la cantidad de cifras correctas se duplica, siendo al inicio 1, 2, 4, 8, 16 y 32, lo que significa que 6 iteraciones se podría rebasar la precisión de la computadora. Entonces sólo se actualizará la derivada una vez además del cálculo inicial. Pareciera demasiado detalle de programación para un resultado tan cuestionablemente mejor, debido a la velocidad de las computadoras modernas.

La convergencia en general queda como algún número entre 1.6 y 2, excepto en la presencia de raíces múltiples, que se revisa a continuación.

Raíces múltiples

Este es el caso en el que Newton y la mayoría de los métodos tienen más problemas. En general se dice que una función f tiene una raíz r con multiplicidad m si se cumple que

$$f(x) = (x - r)^m g(x)$$

donde $g(r) \neq 0$, $f'(x) = f''(x) = \cdots = f^{(m-1)}(x) = 0$ y $f^{(m)}(x) \neq 0$. Además es bien conocido que

$$\lim \frac{u(x)}{x - r} = \frac{1}{m}$$

En r ya no se puede calcular la convexidad logarítmica de f y la velocidad de la convergencia ya no puede evaluarse.

La iteración de Newton definida como

$$x_{n+1} = x_n - m\, u(x_n) \tag{6.13}$$

x_0	$m=1$	$m=2$	$m=3$	m
11	29	11	3	4
10.90625	29	11	3	4
10.8125	29	11	3	4
10.71875	28	11	3	4
10.625	28	11	2	4
10.53125	27	10	2	4
10.4375	27	10	2	4
10.34375	26	10	2	4
10.25	25	10	2	4
10.15625	24	9	2	4
10.0625	22	8	2	4
Tiempo invertido:	58 %	23 %	7 %	10 %

Tabla 6.4: Iteraciones necesarias con multiplicidad fija y calculada para la función $f(x) = (x-1)(x-10)^3$ partir de diversos valores iniciales.

donde u se define como en 6.7 en la página 162, permite una convergencia cuadrática a r. El problema es que puede ser que m no se conozca por anticipado y si el valor estimado no coincide con la multiplicidad real, entonces los resultados no son adecuados y la convergencia puede estar comprometida (típcamente no se alcanza). La idea sería encontrar m en tiempo de ejecución. Utilizando el hecho de que los errores sucesivos de Newton, en el caso de una raíz múltiple, cumplen con

$$e_{n+1} = \frac{m-1}{m} e_n$$

es posible encontrar una aproximación de la multiplicidad como

$$m = \left\lfloor \frac{e_n}{e_n - e_{n+1}} \right\rfloor \tag{6.14}$$

cuando $x_0 > r$. De esta manera, con las dos primeras iteraciones de Newton es posible encontrar una estimación de la multiplicidad de la raíz que se está aproximando.

Aplicando lo anterior a $f(x) = (x-1)(x-10)^3$ como en el ejemplo anterior, se obtiene la tabla 6.4. Se utilizó el método de Newton normal y además se probó 6.13 con $m = 2$, $m = 3$ y el cálculo dinámico 6.14, indicado solo por m. Es claro que $m = 2$ se usó para experimentar una multiplicidad no adecuada. La tabla muestra la cantidad de iteraciones en que se alcanza la convergencia con una tolerancia de 10^{-15}. No se muestra el resultado con $m = 4$ pues ningún método nunca logró la convergencia en 100 iteraciones.

Los porcentajes de la última fila son el tiempo de ejecución empleado por cada método. El cálculo dinámico de m emplea dos pasos "lentos" para estimar la multiplicidad y luego en otras dos iteraciones se acerca cuadráticamente al resultado desde una aproximación bastante buena. De allí la diferencia con el caso de $m = 3$.

También se experimentó con la ecuación a $f(x) = (x-1)(x-10)$ como en el primer ejemplo comparativo de Bisección y Newton. En esta función sin raíz múltiple, para $m > 1$ no se logra la convergencia, mientras que con la estimación de m en tiempo de ejecución, se iguala el caso de Newton normal pues m se evalúa como 1.

La evaluación dinámica de m es un resultado interesante pues permite una convergencia cuadrática ante los casos de raíces simples y múltiples con una evaluación muy sencilla.

Existe otra manera de acelerar el método de Newton y consiste en construir una función donde la raíz r tenga multiplicidad 1. Sea $f(x) = (x-r)^m g(x)$, la dicha función y $g(r) \neq 0$. Entonces u tiene una sola raíz ya que

$$\begin{aligned} u(x) &= \frac{f(x)}{f'(x)} \\ &= \frac{(x-r)^m g(x)}{m(x-r)^{m-1}g(x) + (x-r)^m g'(x)} \\ &= \frac{(x-r)g(x)}{mg(x) + (x-r)g'(x)} \end{aligned}$$

y es claro que u tiene una sóla raíz en r pues $g(r) \neq 0$. Como

$$\begin{aligned} u'(x) &= \frac{[f'(x)]^2 - f(x)f''(x)}{[f'(x)]^2} \\ &= 1 - \frac{f(x)f''(x)}{[f'(x)]^2}, \end{aligned}$$

la función de iteración que aproxima la única raíz de u es

$$\begin{aligned} x_{n+1} &= x_n - \frac{u(x_n)}{u'(x_n)} \\ &= x_n - \frac{f(x_n)f'(x_n)}{[f'(x_n)]^2 - f(x_n)f''(x_n)} \\ &= x_n - \frac{f(x_n)}{f'(x_n) - L(x_n)/f'(x_n} \end{aligned} \tag{6.15}$$

donde L es la aceleración convexa. Como la multiplicidad de r para la función u es 1, la convergencia será cuadrática, como el caso normal de Newton-Raphson.

El inconveniente de la iteración (6.15) es que se requiere la segunda derivada. Se puede aproximar con

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

pero el esquema iterativo sigue siendo mucho más complicado que el original. De nuevo, para poder aproximar adecuadamente la segunad derivada se requiere un proceso iterativo pues no es posible conocer por anticipado el mejor valor posible de h.

La raíz cuadrada

Una aplicación inmediata es el algoritmo para calcular la raíz cuadrada. Si se desea obtener la raíz de m, se construye la función $f(x) = x^2 - m$ y se aplica el método de Newton, obteniendo la recurrencia

$$x_{n+1} = \frac{1}{2}(x_n + \frac{m}{x_n})$$

que fue empleada ya hace siglos por Herón.

Esta es la base del procedimiento que se emplea en las computadoras para la raíz cuadrada, aunque en la práctica, sólo se utiliza una cantidad fija de iteraciones para el refinamiento final, luego de las etapas de reducción de rango y aproximación polinomial, pues el método es muy estable.

6.5.6. Aceleración de Aitken

Alexander Craig (Alec) Aitken, (1895-1967), matemático inglés.

Como el método original de Newton-Raphson es lento cuando la raíz r es múltiple, Aitken diseñó un método conocido como δ^2 de Aitken con el que acelera cualquier proceso con convergencia lineal. Dada la sucesión de aproximaciones $\{x_n\}$, se define $\delta x_n = x_{n+1} - x_n$. Las potencias de δx_n se definen recursivamente:

$$\delta^k x_n = \delta^{k-1}(\delta x_n), \text{ para } k > 1.$$

Para $k = 2$, la fórmula $\delta^2 x_n = x_{n+2} - 2x_{n-1} + x_n$ es útil. Aitken demostró que si existe un número $|\alpha| < 1$ tal que

$$\lim_{n\to\infty} \frac{r - x_{n+1}}{r - x_n} = \alpha$$

entonces la sucesión $\{z_n\}$ definida como

$$z_n = x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n} \tag{6.16}$$

converge más rápido, en el sentido de que $\lim_{n\to\infty} \left|\frac{r-z_n}{r-x_n}\right| = 0$. La iteración del método de Aitken se construye calculando z_n cada paso de Newton. Otras dos formas de la aceleración matemáticamente equivalentes son

$$\begin{aligned} z_n &= x_{n+1} - \frac{(x_{n+1} - x_n)(x_{n+2} - x_{n+1})}{x_{n+2} - 2x_{n+1} + x_n} \qquad \text{y} \\ z_n &= x_{n+1} + \frac{1}{\dfrac{1}{x_{n+2} - x_{n+1}} - \dfrac{1}{x_{n+1} - x_n}} \end{aligned}$$

y la tercera es la más estable desde el punto de vista numérico.

La deducción anterior se debe a Aitken, pero es posible encontrar el mismo resultado considerando dos aproximaciones sucesivas de una sucesión con convergencia lineal. Sea $|c| < 1$. Se cumplen

$$\begin{aligned} |x_{n+1} - r| &= c\,|x_n - r| \\ |x_{n+2} - r| &= c\,|x_{n+1} - r| \end{aligned}$$

y despejando c de ambas e igualando las expresiones se obtiene

$$\frac{x_{n+1} - r}{x_n - r} = \frac{x_{n+2} - r}{x_{n+1} - r}.$$

Despejando r se obtiene de donde se obtiene la relación

$$r = x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n}$$

Iteración	Newton	Steffensen
1	10.3362068965517242	10.3362068965517242
2	10.2254672162094131	10.2254672162094131
3	10.1509187900252726	9.99431762991743078
4	10.100887568334679	9.99621215224659743
5	10.0673821864663324	9.99747494539873749
6	10.0449769574305048	9.9999992019996391
7	10.0300094473814365	-
8	10.0200173671539492	-
⋮	⋮	
27	10.0000090634125449	-

Tabla 6.5: Comparación de Newton y Steffensen ante una raíz múltiple.

que es la aceleración presentada.

El proceso de aceleración de Aitken es útil ante raíces múltiples, pero sólo puede emplearse en el caso de convergencias lineales. Si la convergencia es cuadrática, no hay ninguna aceleración.

Basado en este proceso de aceleración, surge el siguiente método de aproximación.

6.5.7. Método de Steffensen

El eficiente método de Newton se ve seriamente afectado ante la presencia de raíces múltiples. La técnica de aceleración de Aitken fue utilizada por el matemático norteamericano Arnold Steffensen para ayudar al método de Newton-Raphson en casos donde la convergencia es más lenta. El método consiste en iterar dos veces con Newton-Raphson (6.6) y luego acelerar con Aitken[18]. De esta manera, en cada iteración se hacen tres aproximaciones, partiendo de tres iniciales. De la iteración anterior se tienen x_i y ahora se calculan

$$\begin{aligned} x_{i+1} &= x_i - u(x_i) \\ x_{i+2} &= x_{i+1} - u(x_{i+1}) \\ x_{i+3} &= x_i - \frac{(x_{i+1} - x_i)^2}{x_{i+2} - 2x_{i+1} + x_i} \end{aligned} \tag{6.17}$$

y se garantiza convergencia cuadrática aún cuando Newton-Raphson tiene problemas.

La principal desventaja de Steffensen es la potencial cancelación de cifras significativas en el denominador en cuanto las aproximaciones se van acercando más y más, por lo que hay que tomar precauciones.

Comparando los métodos de Newton y Steffensen, utilizando de nuevo el polinomio $(x-1)(x-10)^3$, que fue donde Newton tuvo problemas de convergencia. Utilizando ahora un error relativo más pequeño de 10^{-15}, los resultados de la comparación son los de la tabla 6.5.

[18] No todos los autores coinciden en atribuirle este método a Steffensen. De hecho, hay otro esquema adicional atribuido al mismo autor que se presenta en la página 176.

x	$f(x)$	$f\prime(x)$
10.35	.01718...	.29648...
10.29	.00576...	.119...
10.24	*Paso de Aitken*	
9.9982...	$-4.945... \times 10^{-12}$	$-4.547... \times 10^{-13}$
10.875...		

Tabla 6.6: Iteraciones del método de Steffensen con $(x-1)(x-10)^5$ comenzando en 10.35, con un error relativo de 10^{-15}.

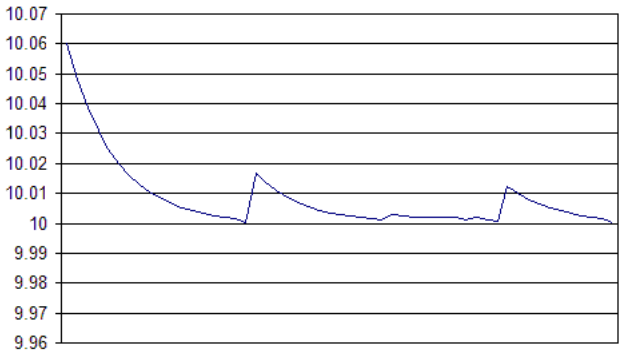


Figura 6.9: Problemas de Newton-Raphson al acercarse a una raíz múltiple. Se presentan las iteraciones de la 11 a la 64 para la función $(x-1)(x-10)^5$.

En la tabla 6.5 se están contando como uno solo los tres pasos que se da en cada iteración de Stefensen. Considerando sólo estos dos métodos, los porcentajes de tiempo de ejecución son de 72 % para Newton y 28 % para Steffensen. Aún así, es posible encontrar condiciones donde el método de Steffenson se comporte mal. Por ejemplo, con la función $(x-1)(x-10)^3$ y el punto inicial 10.55, Newton y Steffenson realizan 32 y 33 iteraciones respectivamente.

En el caso de la función $(x-1)(x-10)^5$, ambos métodos agotan 100 iteraciones sin llegar a la aproximación exigida. El problema es escencialmente la poca estabilidad de los polinomios y la raíz múltiple en 10.

No es difícil encontrar condiciones donde la derivada cerca de la raíz es tan pequeña que la tangente corta el eje x demasiado lejos de la raíz. Por ejemplo, con un error relativo de 10^{-15}, el polinomio $(x-1)(x-10)^5$ y el método de Steffensen comenzando en 10.35, se tienen iteraciones de la tabla 6.6.

Este trastabilleo es típico en raíces múltiples.

Otro ejemplo de mal comportamiento es el de la figura 6.9, donde se muestra de la iteración 11 a la 64 y se ve con claridad que los defectos numéricos de la evaluación del polinomio ocasionan que no se acerque más a la raíz. De hecho, en la iteración 65 la aproximación se pasó a -5.999, lo que ocasionó la convergencia a la raíz más alejada 1.

Ese comportamiento errático también llega a ocurrir en raíces simples y debe detectarse

Iteración	Steffensen	Modificado
1	10.37662338	10.37662338
2	10.2834036	9.995245988
3	9.995245988	9.996434649
4	9.996434655	9.999999452
5	9.99732608	-
6	9.999999466	

Tabla 6.7: Comparación de Steffensen con la modificación del autor. En este caso la mejora es evidente.

6.5.8. Steffensen modificado

Steffenson da tres pasos en cada iteración, dos con Newton y uno con Aitken. Con algo de manipulación algebraica se puede reestructurar y mezclar los pasos 2 y 3. La modificación consiste los dos siguientes pasos.

Esta es una modificación del autor.

$$\begin{aligned} x_{i+1} &= x_i - u(x_i) \\ x_{i+2} &= x_i - \frac{(u(x_i))^2}{u(x_i) - u(x_{i+1})} \end{aligned} \tag{6.18}$$

Esto se obtiene de contraer la tercera fórmula del algoritmo original 6.17, sustituyendo x_{i+2} y x_{i+1} por sus definiciones[19]. Esto significa que se dan los dos pasos finales de Steffensen en uno solo, con menor costo computacional, que tiene como ventaja las siguientes:

1. Ya no se requieren las evaluaciones de x_{i+2}, $f(x_{i+2})$ ni de su derivada. Debe recordarse que la cantidad de evaluaciones de funciones y operaciones elementales es en perjuicio del desempeo de los métodos.
2. La única cantidad nueva es $f(x_{i+1})/f'(x_{i+1}) = u^{-1}(x_{i+1})$ pues las demás ya se calcularon.

En algunos casos, la convergencia teórica reduce de orden cuando el esquema iterativo es muy complicado.

La modificación del algoritmo de Steffensen invierte menos pasos. Por ejemplo, para la función $(x-1)(x-10)^3$, con precisión relativa de 10^{-15}, la comparación con el método original se muetra en la tabla 6.7 con el punto inicial 10.5, por poner el mismo caso de la comparación con Newton.

Como puede apreciarse, hay una ganancia en la cantidad de iteraciones, aunque en este caso, las evaluaciones del método modificado hacen que este consuma más tiempo que el original, para las condiciones iniciales establecidas.

Comparando los tres métodos en varios puntos iniciales, la tabla 6.8 muestra la cantidad de iteraciones que invierte cada uno en alcanzar la precisión exigida, para cada aproximación inicial a la raíz 10 de la misma función anterior.

De la tabla 6.8 se ve que la modificación 6.18 obtiene ventaja cuando Steffensen se desvía y tarda demasiadas iteraciones en alcanzar la aproximación. Por otra parte, es claro que entre más cercano esté x_0 de la raíz mejora la convergencia y que la modificación tiene menos problemas que el método original.

No se puede esperar gran cosa de tan solo una simplificación algebraica

[19] Atreverse a presentar una variante de un método conocido, requiere más pruebas que las aquí presentadas. El autor promete un análisis más exaustivo de la propuesta de modificación.

x_0	Newton	Steffenson	Modificado
11	39	27	10
10.899	31	6	4
10.798	34	6	4
10.697	31	15	4
10.596	31	6	4
10.495	58	21	4
10.394	33	6	4
10.293	38	6	4
10.192	25	6	4
10.091	23	3	2

Tabla 6.8: Comparación de Newton, Steffensen y la modificación del autor para diversos valores iniciales.

No fue difícil encontrar un experimento donde la modificación tuviera problemas. Por ejemplo, para la función $(x-1)(x-10)^5$, el algoritmo modificado no fue el único pero sí el que más se desvió a la raíz 1, en vez de localizar la raíz 10. Cuando se redujo la precisión de 10^{-15} a 10^{-12} ningún método se desvió a la otra raíz, lo que deja ver la dependencia de la precisión exigida para el buen comportamiento de los algoritmos. Queda claro por simple sentido común: no es conveniente exigir más precisión que la realmente necesaria.

En la literatura se encuentra la iteración que también recibe el nombre de Steffensen, que consiste en

$$x_{n+1} = x_n - \frac{[f(x_n)]^2}{f(x_n + f(x_n)) + f(x_n)} \tag{6.19}$$

y aparece en una gran cantidad de referencias de todo tipo.

Falta la variante de Multiple-precision zero-finding methods and the complexity of elementary function evaluation - Brent.

Variantes con convergencia cúbica

Al aplicar Aitken, el método de Newton o Steffensen se acelera la débil convergencia lineal en los casos de raíces múltiples o muy cercanas, logrando de nuevo una convergencia cuadrática.

Hay métodos de convergencia cúbica casi tan antiguos como el de Newton-Raphson, que algunos autores los vieron como generalizaciones del original.

Si se desea una convergencia superior, se pueden utilizar diversos métodos, donde en general se requiere continuidad hasta derivadas superiores (en proporción al orden). Más adelante se presentan algunos esquemas donde no se requieren derivadas (como el de bisección).

6.5.9. Método de Halley

Edmond Halley (1654-1742), famoso por el cálculo de la trayectoria y periodicidad del cometa que lleva su nombre.

Uno de los primeros métodos con convergencia cúbica es el método de Halley, publicado en 1694. Según comenta Traub en [179], Halley encontró el método tras

generalizar una técnica que le impresionó de Fautet Lagny de 1692 para encontrar raíces (en particular cúbicas) de potencias. Lo que lo impresionó fue que $\sqrt[3]{a^3+b}$ se encuentra siempre entre

$$a+\frac{ab}{3a^3+b} \qquad \frac{a}{2}+\sqrt{\frac{a^2}{4}+\frac{b}{3a}}$$

cuando $a^3 \gg b > 0$.

Hay dos versiones del método, llamadas racional e irracional. Como Newton aproxima la raíz con una recta, utilizando un término más en la expansión de Taylor se está utilizando una parábola tangente en x_0, que se ajusta mejor a la función de interés.

La *forma irracional* parte de la aproximación polinomial de la función f

$$p(x) = f(x_n) + (x-x_n)f'(x_n) + \tfrac{1}{2}(x-x_n)^2 f''(x_n) \tag{6.20}$$

e igualando a cero y despejando $x - x_n$ como

$$x - x_n = \frac{-f'(x_n) \pm \sqrt{[f'(x_n)]^2 - 2f''(x_n)f(x_n)}}{f''(x_n)}.$$

Esta fórmula tiene dos problemas: 1- qué signo tomar del radical y 2- que cuando x_n se acerca a la raíz r, $f(x_n)$ es casi cero, por lo que el discriminante es casi $f'(x_n)$, ocasionando la diferencia de dos cantidades casi iguales en el numerador.

El signo del radical debe tomarse de tal manera que el numerador sea lo menor posible, pero depende de $f'(x_n)$, asi que se puede dividir arriba y abajo entre $f'(x_n)$, quedando

$$x - x_n = \frac{-1 \pm \sqrt{1 - \frac{2f''(x_n)f(x_n)}{[f'(x_n)]^2}}}{f''(x_n)/f'(x_n)}$$

pero sólo se resolvió el problema del signo y falta la potencial diferencia entre dos cantidades casi iguales a 1. Pero haciendo uso de que $(1-\sqrt{1-a})(1+\sqrt{1-a}) = a$, que es la transformación usada usada en la página 147 en (6.2), la fórmula queda

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'(x_n) \pm \sqrt{[f'(x_n)]^2 - 2f(x_n)f''(x_n)}} \tag{6.21}$$

que tiene parecido con la iteración de Newton, con unos términos adicionales. Este esquema tiene convergencia cúbica.

En la *forma racional* del método de Halley, se parte de nuevo de la aproximación (6.20)

$$f(x_n) + (x-x_n)f'(x_n) + \tfrac{1}{2}(x-x_n)^2 f''(x_n) = 0$$

y despejando $x - x_n$ como en la ecuación general de segundo grado:

$$x - x_n = -\frac{f(x_n)}{f'(x_n) + \frac{1}{2}(x-x_n)f''(x_n)}.$$

Ahora, partiendo de la aproximación $x = x_n - u(x_n)$, se sustituye $x - x_n$ por $-u(x_n)$, dando:

$$\begin{aligned} x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n) - \frac{1}{2}u(x_n)f''(x_n)} \\ &= x_n - \frac{f(x_n)f'(x_n)}{[f'(x_n)]^2 - \frac{1}{2}f(x_n)f''(x_n)} \end{aligned} \tag{6.22}$$

y se tiene la forma racional, que es la más conocida de las dos[20], también llamada *método de hipérbola tangente* debido a que cuando $x_{n+1} = 0$ se tiene la intersección con el eje x de una hipérbola que es osculatoria a $y = f(x)$ en x_n. En el trabajo [168] de Scavo se da una interpretación geométrica del método de Halley, presumiblemente similar a los razonamientos de finales del siglo XVII.

Es curioso que Taylor dedujo su teorema (ver página 16) alrededor de 1712 tras identificar las derivadas en la formulación original de Halley, según [59] (véase (1.3) en la página 16).

La convergencia se garantiza si se cumple que

$$2\left|x_1 - x_0\right|\left|g''(x)\right| \leq \left|g'(x_0)\right|$$

y que todas las x_k estarán en el intervalo $[x_0, x_0 + 2h]$ si $h = x_1 - x_0 > 0$ o en $[x_0 + 2h, x_0]$ si $h < 0$. En caso de no complirse estas condiciones, la concavidad de la parábola aproximante la curvará antes de tocar el eje x, con lo que no ocurrirá el corte (como el que sí se ve en la figura 6.10a) y por lo tanto no habrá raíz real que sea la siguiente aproximación.

Ejemplo. Sea la función $f(x) = e^x - 8$. Si se comienza a iterar en $x_0 = 3$ se obtienen la recta de Newton y la parábola de Halley que se muestran en la figura 6.10. Es evidente que el corte de la parábola de Halley pasa más cerca de la raíz.

En [4], el matemático alemán Götz Alefeld muestra (entre otras cosas) que el método de Halley (6.22) converge, partiendo de que es posible derivarlo tras aplicar Newton a

$$g(x) = \frac{f(x)}{\sqrt{f'(x)}}.$$

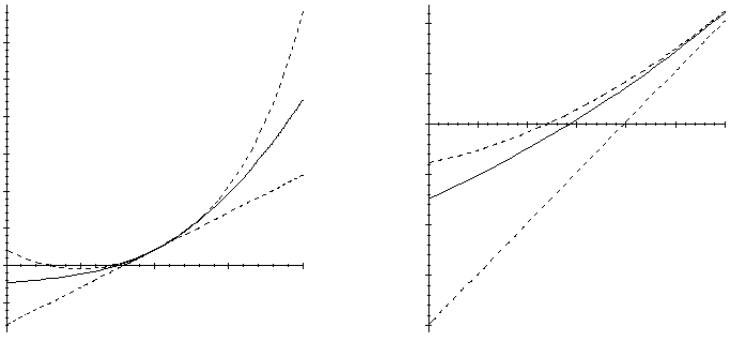
Como la derivada de g es

$$g'(x) = \sqrt{f'(x)} - \frac{1}{2}\frac{f(x)\,f''(x)}{\sqrt{f'(x)}^3},$$

aplicando la iteración de Newton se obtiene

$$\begin{aligned} x &= x - \frac{\dfrac{f(x)}{\sqrt{f'(x)}}}{\sqrt{f'(x)} - \frac{1}{2}\frac{f(x)f''(x)}{\sqrt{f'(x)}^3}} \\ &= x - \frac{f(x)}{f'(x) - \frac{1}{2}\frac{f(x)f''(x)}{f'(x)}} \\ &= x - \frac{f(x)f'(x)}{[f'(x)]^2 - \frac{1}{2}f(x)\,f''(x)} \end{aligned}$$

[20] Halley prefería la forma irracional argumentando mayor precisión, pero ahora se sabe que no encontró los contraejemplos adecuados.



a. De lejos ... **b.** ... y de cerca.

Figura 6.10: Comparación gráfica de los métodos de Newton y Halley. La línea sólida es la función $f(x) = e^x - 8$, la recta es la aproximación de Newton.

que es el método de Halley.

Para probar la convergencia del método de Halley, se utiliza una variante del teorema de Kantorovich, debida a Ostrowski, que puede consultarse en el artículo de Dong Cheng, donde además se demuestra su convergencia cúbica.

Revisar tambien Review of iterative methods, sobre super-Halley.

6.5.10. Variantes con cuadratura

Se han desarrollado diversos esquemas iterativos de punto fijo que evitan el uso de derivadas. Los ejemplos clásicos son bisección, secante y la regla de falsa posición, pero hay muchos otros. Otra forma de no usar derivadas es mediante aproximaciones, por ejemplo, el algoritmo LZ4 de Le, en [122], que utiliza la iteración de Chebyshev (pag. 195) como paso principal sin utilizar explícitamente ni f' ni f''.

Por otra parte, Amat presenta en [9] un método llamado de dos pasos atribuido a Potra y Ptak, que también tiene convergencia cúbica pero sin usar la segunda derivada. El esquema es

$$x_{n+1} = x_n - \frac{f(x_n) + f(x_n - u(x_n))}{f'(x_n)} \tag{6.23}$$

y durante mucho tiempo fue el único conocido con estas características.

Una interesante manera de deducir el método de Newton se basa en una conocida y sencilla técnica de integración, llamada Newton-Cotes. Como la integral

$$\int_{x_n}^{x_{n+1}} f(x)\,dx$$

es el área bajo la curva de f en el intervalo $[x_n, x_{n+1}]$, entonces una aproximación muy burda del área es el rectángulo cuya base es $x_{n+1} - x_n$ y su altura es $f(x_n)$ o $f(x_{n+1})$.

Con cualquiera de las dos alturas se obtiene una burda aproximación a la integral que interesa.

Aprovechando esto, se puede notar que si se usa el teorema de Newton

$$f(x) = f(x_n) + \int_{x_n}^{x} f'(t)\,dt$$

y se aproxima la integral con un rectángulo de altura $f'(x_n)$ y base $x - x_n$, se obtiene

$$\int_{x_n}^{x} f'(t)\,dt \approx (x - x_n)f'(x_n).$$

Haciendo $f(x) = 0$ y despejando x se obtiene $x_{n+1} = x_n - u(x_n)$, que es el método de Newton.

Esta manera de deducir se deriva de la aproximación de la integral con un rectángulo, conocido como la regla de Newton-Cotes de orden cero, que es la aproximación con mayor error. Aproximando la integral con un rectángulo de altura $\frac{1}{2}(f'(x) + f'(x_n))$ se obtiene una mejor aproximacón que lleva a la iteración de punto fijo

Conocido como la regla del trapecio.

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'(x_n) + f'(x_{n+1})}$$

que es una ecuación implícita en x_{n+1}, es decir, no se puede despejar x_{n+1}. Lo que sí se puede hacer es sustituir en el argumento de f' con $x_{n+1} = x_n - u(x_n)$ y reescribir

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'(x_n) + f'(x_n - u(x_n))}. \tag{6.24}$$

Este método iterativo, presentado por Weerakoon en [186], tiene convergencia cúbica, como el método de Halley, pero sin requerir la segunda derivada. Otra forma que también tiene convergencia cúbica se obtiene si en vez de utilizar un trapecio se aproxima la integral con la regla del punto medio, como hicieron Frontini y Sormani en [65, 66]. En ese caso, se parte de la aproximación

$$\int_{x_n}^{x} f'(t)\,dt \approx (x - x_n)f'(x_n/2 + x/2)$$

y la iteración queda

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n - \frac{1}{2}u(x_n))}. \tag{6.25}$$

No es difícil ver que tanto (6.24) como (6.25) tienen cierto parecido, por el argumento $x_n - k\,u(x_n)$, donde k vale 1 y $\frac{1}{2}$ respectivamente.

Es natural pensar que si se utilizan reglas de Newton Cotes de orden mayor, como Simpson, Simpson 3/8 o la regla de Bule se llegará a métodos iterativos con convergencia mayor, pero ya en 2003, Frontini y Sormani demostraron en [66] que todas las derivaciones de Newton-Cotes de orden 1 en adelante alcanzan convergencia cúbica como máximo.

Eso sin considerar el fenómeno de Runge, donde en algunos casos, los polinomios de grado alto van degradando cada vez más la aproximación de la integral debido a su sensibilidad.

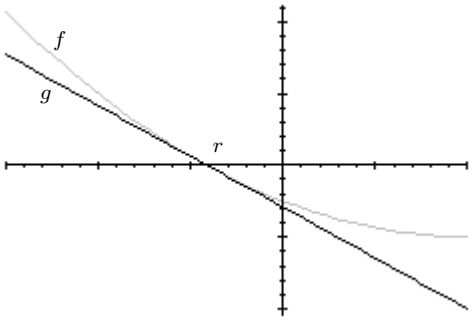


Figura 6.11: Acceleración convexa de Newton.

6.5.11. Aceleración convexa - Super Halley

En el caso de que la función f cumpla con ciertas condiciones de convexidad (similares a las de la página 23) es posible acelerar el método de Newton, como presentan Gutiérrez y Hernández en [81][21]. Ellos demuestran que su iteración es una aceleración de Newton si f satisface $f'(x) < 0$ y $f''(x) > 0$ para $a \leq t \leq r$, con $f(r) = 0$. Si la segunda derivada de f es no decreciente en $[a, r]$ entonces la convexidad logarítmica $L_f \leq \frac{1}{2}$ en ese intervalo.

Si x_n es la n-ésima aproximación con Newton, entonces puede obtenerse una sucesión z_n acelerada con el esquema iterativo

$$z_n = x_n - \left[1 + \frac{L_f(x_n)}{2(1 - L_f(x_n))}\right] \frac{f(x_n)}{f'(x_n)} \tag{6.26}$$

que a diferencia de la aceleración de Aitken (6.16) hace uso de la geometría de f. La restricción que debe cumplirse es $f'''(x) \geq 0$ para toda $x \in [a, r]$.

El esquema iterativo (6.26) lo obtienen de hacer $g(z) = f'(r)(z - r)$, que es la recta tangente a f en r, por lo que cruza el eje x en la raíz, como muestra la figura 6.11. Como en toda recta, $L_g = 0$. De esta manera, $g(z)$ define una sucesión que converge a r más rápido.

Como r no se conoce, se aproxima g con el polinomio de Taylor

$$g(z) = f(z) - \frac{f''(r)}{2}(z - r)^2$$

reescribiendo como

$$g(z_n) = f(z_n) - \frac{f''(z_n)}{2}(z_n - z_{n+1})^2$$

de donde se deduce la iteración $z_{n+1} = z_n - \frac{g(z_n)}{g'(z_n)}$ que se desarrolla y obtiene (6.26).

Según muestran Gutiérrez y Hernández en [81], la iteración (6.26) es de orden cúbico para raíces simples y degrada a orden lineal en el caso de raíces múltiples. Para polinomios cuadráticos este método alcanza convergencia cuártica en raíces simples, siendo una iteración de (6.26) equivalente a dos iteraciones de (6.6).

[21] Gutiérrez y Hernández generalizan sus resultados en espacios de Banach y los utilizan en un ejemplo para resolver un sistema de ecuaciones no lineales.

Métodos que no requieren derivada

El mayor problema de los métodos anteriores es que hacen uso de la derivada. Como ya se dijo, normalmente la derivada es mucho más compleja que la función original, por lo que afecta el desempeño de los métodos.

Además, en el caso de que se tenga un sistema autónomo, donde las funciones se generen dinámicamente, habrá que tener un módulo que calcula la derivada en una forma numéricamente efectiva, lo que es bastante difícil.

Los siguientes métodos no requieren usar ni la primera derivada.

6.5.12. Método de la secante

El método de la secante es una variación del método de Newton, útil cuando no hay derivada cerca de r o que su cálculo es demasiado complejo. La idea básica es sustituir la recta tangente por una secante. Por ello, se requieren dos puntos iniciales. La derivación es muy simple, radica en escribir la ecuación de la recta secante a partir de dos puntos de la función f, por donde pasa la secante, como en la siguiente gráfica.

Según la ecuación punto pendiente y suponiendo que x_{n-1} y x_n son los puntos iniciales, la secante que pasa por los puntos $(x_{n-1}, f(x_{n-1}))$ y $(x_n, f(x_n))$ es

$$y - f(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x - x_n).$$

Si ahora se hace x_{n+1} el corte de la recta con el eje x y se escribe

$$f(x_n) + \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x_{n+1} - x_n) = 0$$

de donde se despeja x_{n+1}, obteniéndose el esquema iterativo del método:

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n) \tag{6.27}$$

que es similar al método de Newton[22]. Ya se había utilizado esta idea en la página 168 (ecuación (6.12)).

Convergencia del método de la secante

Si las dos aproximaciones iniciales son suficientemente buenas, se sabe que el método tiene un orden de convergencia de

$$\frac{1 + \sqrt{5}}{2} \approx 1.618 \tag{6.28}$$

y se dice que tiene convergencia superlineal. Esta convergencia es un cálculo empírico a partir del hecho de que la definición de convergencia6.3 en la página 153 es $|e_{n+1}| \leq \alpha |e_n|^k$. Al despejar k se obtiene

$$k = \frac{\log(e_{n+1})}{\log(\alpha e_n)} \tag{6.29}$$

[22] Véase más adelante el método de Laguerre, para una extensión de estas ideas.

que converge a (6.28).

Esta técnica (6.29) se puede aplicar a cualquier método iterativo. Es un buen ejercicio comprobar las convergencias teóricas con las reales pues depende de la condición de cada función.

Secante se puede aplicar sin problemas si f es dos veces diferenciable y la raíz r es simple, es decir, no tiene multiplicidad. Al igual que el método de Newton, hay problemas cuando la derivada se acerca a 0 en valor absoluto.

6.5.13. Método de falsa posición o regula falsa

En el método de la secante las dos aproximaciones en cada iteración están ambas a un mismo lado de la raíz. Una variante es aplicar el teorema del valor intermedio, utilizando la idea del método de bisección y poner un punto a cada lado de la raíz, calculando la secante. El corte de la secante con el eje x es la nueva aproximación, de donde se tiende la nueva secante. Esto asegura que el corte en el eje x de cada secante sucesiva pasa más cerca de r.

Lo único que se requiere es calcular el corte en el eje x de la secante en cada iteración. La ecuación de la secante a f en los extremos del intervalo $[a_n, b_n]$ en la n-ésima iteración es

$$y - f(b_n) = \frac{f(b_n) - f(a_n)}{b_n - a_n}(x - b_n).$$

De hecho es la misma ecuación de la secante pero con la notación del método de bisección. Por lo que la aproximación Se selecciona c_n como el corte en el eje x, por lo que sustituyendo y despejando se obtiene

$$f(b_n) + \frac{f(b_n) - f(a_n)}{b_n - a_n}(c_n - b_n) = 0$$

de donde se despeja c_n:

$$c_n = b_n - \frac{b_n - a_n}{f(b_n) - f(a_n)} f(b_n) \tag{6.30}$$

que es igual que la iteración del método de la secante, pero se debe estar revisando que $f(a)f(b) < 0$ como en el método de bisección. También se acostumbra escribirlo de la siguiente forma

$$c_n = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}$$

para pensar que se calcula un punto en el intervalo $[a, b]$, como en bisección.

En este método, en ocasiones siempre es el mismo extremo el que se aproxima a r para algunas ecuaciones, por lo que el intervalo no se va reduciendo a la mitad como en el método de bisección, lo que resulta en una convergencia pobremente lineal.

J. A. Ford presentó en 1995 un reporte técnico [62] donde estudia varias maneras de mejorar algoritmos para funciones no lineales, entre ellos el de la regla falsa. Dentro de otras cosas, propone evaluar si el mismo extremo ha cambiado en dos iteraciones consecutivas para forzar un cambio en el otro extremo, con las fórmulas

$$c_n = \frac{\frac{1}{2}a_n f(b_n) - b_n f(a_n)}{\frac{1}{2}f(b_n) - f(a_n)} \quad \text{ó} \quad c_n = \frac{a_n f(b_n) - \frac{1}{2}b_n f(a_n)}{f(b_n) - \frac{1}{2}f(a_n)}$$

dependiendo el extremo repetido. Esto garantiza una convergencia superior a la del método original. Esta variante, llama Método Illinois, asegura una convergencia de 1.442 y es similar a la conocida como Pegasus, que es todavía un poco mejor con 1.642. En esta última, el $1/2$ se sustituye con $y_i/(y_i + y_{i+1})$.

Un refinamiento aún mejor es sustituyendo el $1/2$ por α, definida como

$$\begin{aligned} \alpha &= \begin{cases} \beta, & \beta > 0 \\ \frac{1}{2}, & \beta \leq 0 \end{cases} \\ \text{con } \beta &= \frac{(x_i - x_{i-1})(f(x_{i+1}) - f(x_i))}{(x_{i+1} - x_i)(f(x_i) - f(x_{i-1}))} \end{aligned}$$

que de hecho se obtiene aplicando el concepto de diferencias divididas. El orden de convergencia del refinamiento es de al menos 1.68, apenas superior a secante.

6.5.14. Método de Müller

Como la secante es una burda aproximación a la función con una recta, en 1956 Müller agregó un punto más para utilizar una parábola[23] y obtener una mejor aproximación en [139] que coincidiera con la función en tres puntos. Este método puede requerir aritmética compleja por los motivos que se expondrán. Ahora se parte de x_0, x_1 y x_2 al inicio del algoritmo.

Aunque Müller lo presenta para el acso de polinomios, el método fue aplicado desde su nacimiento a funciones generales, en particular a funciones analíticas. Las raíces múltiples también son localizadas, aunque con convergencia menor, pero superlineal aún, a diferencia de Newton-Raphson. Una ventaja es que aunque requiere varios cálculos auxiliares, sólo requiere una evaluación nueva de f en cada iteración y no usa derivada alguna.

Es conveniente introducir el concepto de *diferencias divididas*. Sea el conjunto de n puntos $\{(x_i, f(x_i))\}_{i=0}^{n-1}$. Las diferencias divididas se definen recursivamente como

$$\begin{aligned} [x_k] &= f(x_k) \\ [x_k, ..., x_{k+j}] &= \frac{[x_{k+1}, ..., x_{k+j}] - [x_k, ..., x_{k+j-1}]}{x_{k+j} - x_k} \end{aligned}$$

y son utilizadas para la evaluación de funciones dadas en tablas, en especial para la construcción de los coeficientes del polinomio de interpolación de Newton. Usando esta notación, en el método de falsa posición se podría reescribir la β del último refinamiento como $\beta = [x_{i+1}, x_i]/[x_i, x_{i-1}]$.

Regresando al método de Müller, se utilizan los tres valores iniciales para construir el polinomio interpolante de Newton, que pasa por los puntos iniciales $\{(x_i, y_i)\}_{i=0}^{2}$ como

$$y = f(x_2) + (x - x_2)f[x_2, x_1] + (x - x_2)(x - x_1)f[x_2, x_1, x_0].$$

Escribiendo $m = f[x_2, x_1] + f[x_2, x_0] - f[x_1, x_0]$ se puede escribir la iteración de Müller como

$$x_3 = x_2 - \frac{2f(x_2)}{m \pm \sqrt{m^2 - 2f(x_2)f[x_2, x_1, x_0]}}$$

[23] La diferencia con el método de Halley es que allí la parábola se busca tangente a f en cada nueva aproximación.

y el signo se escoge de tal manera que el denominador sea lo mayor posible para evitar errores. En general se aplican las mismas observaciones que en el caso de la fórmula general.

Como las aproximaciones pueden ser números complejos, es necesario realizar las operaciones con aritmética compleja. Es por ello que este método normalmente se utiliza sólo con polinomios, aunque sí se aplica a funciones analíticas, como se dijo antes.

Otra idea es que si se buscan raíces reales, a cualquier aproximación intermedia se le puede anular la parte imaginaria y continuar los cálculos con números reales. Demerita la convergencia pero facilita la programación.

Aunque la convergencia de (6.5.14) es local, padece de problemas similares a otros métodos, como cuando se aproxima una raíz y se llega a otra. Claro, siempre es posible acotar cada paso que se da en la aproximación e impedir un salto excesivo, principalmente cuando ocurre por una impresición numérica.

La convergencia es superlineal, con la ventaja mencionada: converge a raíces reales o complejas a partir de una aproximación real.

Programación

Normalmente se prefiere seguir los siguientes pasos, que han mostrado ser suficientemente estables. Sea la iteración k.

$h_k = x^{(k)} - x^{(k-1)}$

$r_k = h_k/h_{k-1}$

$a_k = r_k f(x^{(k)}) - r_k(1+r_k)f(x^{(k-1)}) + r_k^2 f(x^{(k-2)})$

$b_k = (2r_k+1)f(x^{(k)}) - (1+r_k)^2 f(x^{(k-1)}) + r_k^2 f(x^{k-2})$

$c_k = (1+r_k)f(x^{(k)})$

$d_k = \dfrac{-2c_k}{b_k \pm \sqrt{b_k^2 - 4a_k c_k}}$ (raíz de la parábola, debe tomarse la que garantice el mayor denominador en valor absoluto)

$x^{(k+1)} = x^{(k)} + h_k d_k$

Y estas operaciones se repiten hasta alcanzar alguno de los criterios de paro convencionales. Claro, la codificación puede optimizarse con unas cuántas operaciones intermedias, como el cálculo de $1+r_k$ que debe hacerse una vez. El discriminante debe calcularse con el cuidado de costumbre, escalando de ser necesario.

Detalles de implementación

Los criterios normales deben aplicarse: máximo número de iteraciones en caso de que la convergencia sea lenta y detenerse cuando las iteraciones están en la tercera etapa: el error de las aproximaciones no disminuye. Normalmente debe establecerse el paro cuando

$$\left| \frac{z^{(k+1)} - z^{(k)}}{z^{(k+1)}} \right| < \varepsilon.$$

En el cálculo de la raíz de la parábola debe cuidarse que d_k no alcance un valor demasiado grande, pues en ese caso la aproximación puede saltar a otra raíz, ocasionando convergencia lenta o inconsistencias. Esto puede evitarse acotando el incremento relativo de d_k entre una iteración y otra.

Para evitar un problema de overflow, se puede calcular

$$n \log_{10} \left| z^{(k+1)} \right|$$

y si el resultado es resultado es demasiado grande, aproximar con

$$x^{(k+1)} = x^{(k)} + \frac{h_k d_k}{2}$$

y repetir hasta que no ocurra el overflow. El efecto será acercarse cada vez más al valor anterior $x^{(k)}$. Por supuesto, el criterio $\left| f(x^{(k+1)} \right| < \varepsilon$ debe imperar.

Consultar Lan1993Non9ANewandEff.

6.5.15. Método de Ridder

Una variante del método de falsa posición fue propuesta por C. J. F. Ridder en 1979 (IEEE Transactions on Circuits and Systems, vol. CAS-26, pp.979-980). En esta variante se calcula el punto medio c_n del intervalo $[a_n, b_n]$ y con los tres puntos se calcula el cuarto punto

$$d_n = c_n + (c_n - a_n) \frac{signo(f(a_n) - f(b_n)) f(c_n)}{\sqrt{f(c_n)^2 - f(a_n) f(b_n)}} \tag{6.31}$$

que es la raíz de una parábola que aproxima a f como en el caso del método de Müller.

Como d_n está en el intervalo $[a_n, b_n]$, ahora se evalúa dónde ocurre el cambio de signo, para proseguir con el intervalo adecuado, ya sea $[a_n, d_n]$ si $f(a_n)f(d_n) < 0$ o con $[d_n, b_n]$ si $f(d_n)f(b_n) < 0$. Su convergencia es cuadrática, aunque por la evaluación de funciones, el desempeño real es de convergencia $\sqrt{2}$.

Pese a ello, (6.31) tiene cualidades que se esperan en los buenos métodos de aproximación de raíces. Por ejemplo, cada aproximación d_n se mantiene siempre dentro del intervalo $[a, b]$. El discriminante en el denominador jamás es negativo por las restricciones de cambio de signo, por lo que la aritmética siempre es real.

Lo que hace es resolver la ecuación exponencial

$$e^{2t} f(b_n) - 2e^t f(c_n) + f(a_n) = 0$$

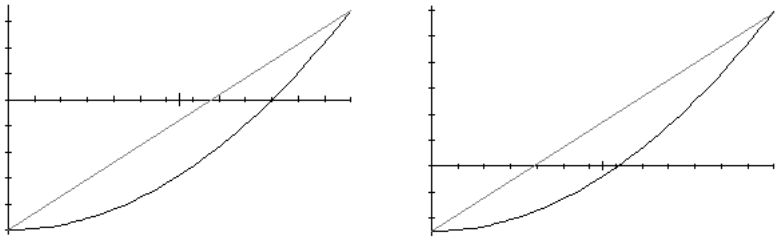
que consiste en una función cuadrática para la nueva variable $w = e^t$. Como en el caso del método de Müller, se aplica la fórmula general de segundo grado.

De nuevo, los cuidados comentados al inicio del capítulo en la sección 6.2 deben considerarse, como el escalamiento del discriminante y las banderas de estado.

*** * ***

Los siguientes tres métodos son ejemplo de que la combinación de estrategias es beneficiosa pues cuando un algoritmo rápido pierde el rumbo la correción se obtiene de un algoritmo lento pero seguro.

* * *



a. La secante es mejor aproximación a r.

b. El punto medio m es mejor aproximación.

Figura 6.12: Aproximaciones del método de Dekker.

6.5.16. Método de Dekker

El método de Dekker, publicado en 1969, combina bisección con falsa posición o secante y en cada iteración determina con cuál calcular la siguiente aproximación. Dekker llama a la secante interpolación lineal en su publicación. El método de Dekker trabaja de la siguiente manera.

Theodorus Jozef Dekker, matemático holandés, famoso por un algoritmo que permite a dos procesos compartir un recurso no divisible usando sólo memoria compartida.

Si se busca la raíz de la función continua f, se seleccionandos puntos a y b tales que $f(a)f(b) < 0$, como en bisección. La existencia de la raíz está garantizada si se cumplen estas condiciones. Como en bisección y secante, un tercer punto está involucrado, ya sea el punto medio de bisección o la siguiente aproximación por secante. Se espera que el tercer punto esté más cerca de la raíz que las aproximaciones anteriores. Ahora se definen las siguientes cantidades.

Sea b_n el mejor candidato a raíz es decir, que se cumpla que $|f(b_n)| < |f(a_n)|$.

Sea a_n es el punto contrario, en términos del signo, $f(a)f(b) < 0$.

Sea b_{n-1} el candidato anterior. En el caso de la primera iteración se hace $b_{-1} = a_0$.

Se calculan dos valores provisionales[24]:

$$\underbrace{m = \frac{a_n + b_n}{2}}_{\text{bisección}} \quad \text{y} \quad \underbrace{t = b_n - \frac{b_n - b_{n-1}}{f(b_n) - f(b_{n-1})} f(b_n)}_{\text{secante}} \tag{6.32}$$

Si $t \in (m, b)$, entonces t se convierte en el nuevo candidato, $b_{n+1} = t$. Si no es así, entonces $b_{n+1} = m$. Claro, debe verificarse que el cambio de signo ocurra. En la figura 6.12 se muestran los dos casos posibles.

Los otros casos se dan por simetría.

Ahora se determinan el *punto contrario*. En caso de que $f(a_n)f(b_{n+1}) < 0$, a_n permanece igual, $a_{n+1} = a_n$. En caso contrario, $a_{n+1} = b_n$.

Si $|f(b_n)| < |f(a_n)|$ entonces los puntos contrarios se mantienen iguales. En otro caso, como a_{n+1} parece mejor candidato a raíz, se intercambia con b_{n+1}.

La iteración del método de Dekker termina cuando se cumple alguna de las condiciones de paro que el programador determine, como $f(b_n) < \epsilon$, $|b_n - a_n| < \epsilon$ o haber

[24] Dekker calcula un tercer valor adicional que consiste en acercar el extremo más prometedor del intervalo hacia la raíz pero con un paso tan pequeño como la tolerancia aceptable, es decir, hace $b' = b + \texttt{signo}(a - b) \times \epsilon$.

llegado a una cantidad máxima de iteraciones. Dekker indica que se termina de iterar cuando se cumple la segunda de las condiciones, aunque eso revela la aritmética de punto flotante de la época.

La convergencia teórica es superlineal, heredada más de secante que de bisección. Al mismo tiempo, hereda los problemas con ciertas funciones.

En 1975, Dekker y Bus publican [28] donde diseñan otros dos algoritmos a partir de este, agregando una posibilidad más a 6.32. En vez de solo interpolar con una recta como es secante o falsa posición, además de la seguridad de bisección, utilizan lo que se conoce como interpolación racional, que consiste en

$$\phi(x) = \frac{x - w}{px + q} \tag{6.33}$$

donde los parámetros p, q y w se determinan de tal manera que $\phi(x) = f(x)$ en a y b. Para ello se requiere un tercer punto además de los límites del intervalo, normalmente se toma b_n o a_n, dependiendo de cómo se actualizó en la iteración anterior. Es simplemente una nueva función que se espera se parezca más a f que una simple secante, como lo sería ajustar una parábola a partir de 3 puntos, pero sin necesidad de aplicar radicales para encontrar la aproximación.

Para el segundo método, la diferencia consiste en asegurar que no se apliquen más de dos pasos del mismo algoritmo, que es una idea basada en el comportamiento asintótico de los métodos, con lo que se logra una mejora teórica de una convergencia de orden 1.842.

Ambas variantes no han vuelto a ser referidas con frecuencia y sigue conociéndose al método de Dekker como el explicado al principio.

6.5.17. Método de Brent

Richard Brent (1946 –), matemático asustraliano.

Este método es realmente un híbrido más, desarrollado en conjunto por van Wijngaarden y Dekker en Amsterdan en 1969 y posteriormente Brent en 1973, quien le dio su forma final presentada en [18]. Este método se emplea en la NetLib con ligeras modificaciones, además de MatLab. Está basado en los de bisección y falsa posición, agregando la posibilidad de usar *interpolación cuadrática inversa* (ICI), lo que requiere de explicación adicional.

Cuando se interpola un conjunto de puntos $\{(x_i, f(x_i))\}_{i=0}^{n-1}$, se busca poder aproximar $f(x)$ para valores de x que no existen en la tabla pero sí dentro de su rango. Por ejemplo, en el caso más simple, si se tienen los dos puntos $\{(2, 8), (3, 10)\}$ se puede interpolar con una recta que pase por ambos. Entonces se puede contestar cuál es el valor de la función interpolada en $x = 2.7$, por decir algo.

En vez de eso, la interpolación inversa supone el valor de $f(x)$ y busca el valor correspondiente x. En el ejemplo anterior, la pregunta sería en qué valor del intervalo $[2, 3]$ se obtiene la ordenada 9.1 y se despejaría para encontrar la abscisa[25]. La ICI interpola con un polinomio de Lagrange de grado dos (6.34) y se busca la abscisa para la que el polinomio debiera valer cero, es decir, $f^{-1}(0)$.

[25] Algunos le llaman interpolación en reversa. El método de la secante es una interpolación lineal inversa en el fondo pues aún intercambiando abscisas con ordenadas se obtiene la misma recta.

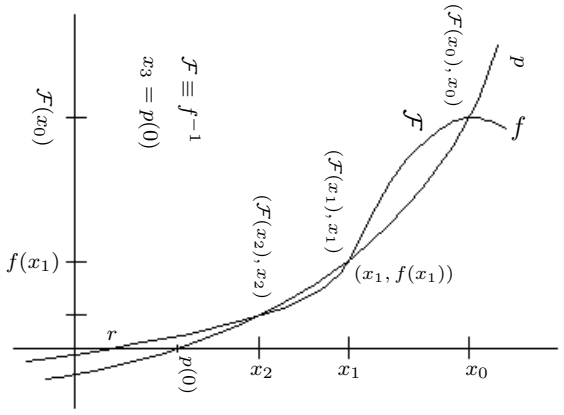


Figura 6.13: Los puntos $(\mathcal{F}(x_i), x_i)$ determinan la parábola p, cuya ordenada en el origen es la siguiente aproximación. En esta orientación las abscisas están en sentido inverso.

Figura 6.13: Interpolación cuadrática inversa. En esta orientación, la forma de determinar x es con la fórmula general, lo que da dos raíces.

La figura 6.13 puede verse en dos orientaciones. En la orientación vertical se ve la gráfica convencional de f y las coordenadas $(x_i, f(x_i)$. Cuando se rota el texto se ve como la gráfica de la función inversa, $\mathcal{F} \equiv f^{-1}$. El polinomio p interpola los tres puntos $(\mathcal{F}(x_i), x_i)$ y la nueva aproximación es $x_3 = p(0)$.

No debe haber confusión: puede ser que la función que interesa no tenga inversa en la raíz r, pues pudiera ser raíz múltiple, pero la parábola interpolante p construida en la ICI siempre tendrá inversa en su cruce con cero por la forma de construcción. Lo peor que pueda pasar es que los puntos son tan distantes de r que la parábola no cruza el eje.

Si los tres puntos iniciales son $\{(x_i, f(x_i))\}_{i=0}^{2}$, el paso de ICI es

$$\begin{aligned} x_3 \quad = \quad & \frac{f(x_0)f(x_2)x_1}{[f(x_1)-f(x_0)][f(x_1)-f(x_2)]} + \frac{f(x_1)f(x_2)x_0}{[f(x_0)-f(x_1)][f(x_0)-f(x_2)]} \\ & + \frac{f(x_0)f(x_1)x_2}{[f(x_2)-f(x_0)][f(x_2)-f(x_1)]} \end{aligned} \tag{6.34}$$

que tiene convergencia más rápida que bisección. Ya Ostrowski en 1966 había probado que con pura interpolación lineal se alcanzaba convergencia de orden $\frac{1}{2}(\sqrt{5}+1) = 1.618\ldots$, que es lo mínimo que se espera de este método.

La pregunta aquí es cuál de los tres métodos utilizar en cada iteración. El más robusto es bisección, aunque más ineficiente. El más eficiente es ICI, pero con riesgos de desorientación. En general la idea es utilizar ICI cada vez que se pueda y en caso de algún problema utilizar el método de la secante. Si secante no obtiene una aproximación suficientemente buena, se da un paso seguro con bisección.

Los puntos x_0 y x_1 son el intervalo inicial $[a_0, b_0]$ y se generará la sucesión de intervalos $[a_i, b_i]$ donde b_i es el mejor candidato a raíz. Por ello, si al inicio ocurre que $f(a_0) < f(b_0)$ entonces hay que intercambiar los valores.

Puede ocurrir que la aproximación del método de la secante se repita en iteraciones sucesivas, con la desventaja de que para algunas funciones, la diferencia $|b_n - b_{n-1}|$

es demasiado pequeña, por lo que la convergencia es lenta. Para evitar este problema, Dekker propone lo siguiente.

1. Si en la iteración anterior se utilizó el método de la secante, verificar que $|x_{n+1} - b_n| < \frac{1}{2}|b_n - b_{n-1}|$. Si esto no se cumple, la aproximación actual t no se acepta y se prefiere el punto medio (bisección), que reduce con certeza el intervalo a la mitad.
2. Si en el paso previo se utilizó ICI, se verifica que $|x_{n+1} - b_n| < \frac{1}{2}|b_{n-1} - b_{n-2}|$ y si no se cumple también se utiliza bisección. Es decir, se le exige más a la ICI.

Como el paso de ICI es el mejor, se aplica siempre que $f(b_n)$, $f(a_n)$ y $f(b_{n-1})$ son distintos, en un intento de mejorar la eficiencia. Esto ocasiona un cambio ligero en el criterio para aceptar la nueva aproximación x_{n+1}. Se acepta x_{n+1} si está entre $(3a_n + b_n)/4$ y b_n.

Detalles de programación

Con todo lo anterior, el algoritmo de Brent sin optimizar se programaría como el código 6.13. Sea dado el intervalo $[a, b]$ donde f tiene un cambio de signo. Bisección se agrega para cuando los otros dos no ofrecen una buena aproximación, así que será la última opción por considerar. Primero se asegura de que b sea el límite del intervalo dónde la función f es más cercana a cero, lo que da cierta posibilidad de que sea la mejor aproximación.

Este código que reescribe el algoritmo descrito, es bastante ineficiente. Primeramente, al inicio de cada iteración se tienen que calcular los valores $f(a_n)$, $f(b_n)$ y $f(c_n)$ y guardar los valores en tres variables, por ejemplo `fa`, `fb` y `fc`. Sólo debe actualizarse la que cambie.

Por otra parte, como la ICI requiere de una gran cantidad de evaluaciones, hay que optimizarlas. La fórmula 6.34 puede reescribirse como $x = b + \frac{p}{q}$, donde p y q se definen de la siguiente manera.

$$\begin{aligned} p &= s\left[t(r-t)(c-b) - (1-r)(b-a)\right] \\ q &= (t-1)(r-1)(s-1) \end{aligned}$$

y las cantidades restantes se definen como sigue.

$$\begin{aligned} r &= \frac{f(b)}{f(c)} \\ s &= \frac{f(b)}{f(a)} \\ t &= \frac{f(a)}{f(c)} \end{aligned}$$

En cada momento se espera que b sea una buena aproximación a la raíz y que el término p/q sea una correción no muy grande. De esta forma la instrucción que calcula la aproximación de la iteración de la ICI se cambiaría por los cálculos de r, s, t, p y q, para luego aplicar la fórmula $x = b + \frac{p}{q}$.

Como cada iteración incluye un potencial paso con bisección, se deben utilizar las optimizaciones que ya se discutieron para este método. Una vez que se combinan los

Listing 6.13: Método de Brent para aproximar ceros de ecuaciones.

```
if ( fabs(f(a))<fabs(f(b)) ) {
   w=a;
   a=b;
   b=w;
}
c=a;
B=SI;
for (i=1; i<MaxIteraciones ; i++ ) {
   if ( f(a)!=f(c) && f(b)!=f(c) ) {
      x =  a*f(b)*f(c)/((f(a)-f(b)*(f(a)-f(c)))+
      f(a)*b*f(c)/((f(b)-f(a)*(f(b)-f(c))) +
      f(a)*f(b)*c/((f(c)-f(a)*(f(c)-f(b)));
   } else
      x = b-f(b)*(b-a)/(f(b)-f(a));
   if( x<min((3*a+b)/4,b) || x>max((3*a+b)/4,b) ||
      B==SI && fabs(x-b)>=fabs(b-c)/2 ) ||
      B==NO && fabs(x-b)>=fabs(c-d)/2 )     }
      x=(a+b)/2;
      B=SI;
   } else
      B=NO;
   d=c;
   c=b;
   if ( f(a)*f(x)<0 )
      b=x;
   else
      a=x;
   if ( fabs(f(a))<fabs(f(b)) ) {
      w=a;
      a=b;
      b=w;
   }
}
```

tres métodos de manera óptima, el código no queda tan legible como el presentado, pero funciona con mayor eficiencia. Este es un excelente método para una librería de funciones matemáticas.

La ICI no es la única opción que han tomado algunos autores, por ejemplo, en [5] Alefeld combina ICI con interpolación cúbica inversa, lo que acelera un poco más la convergencia, pero sin llegar a cuadrática. En este caso, se requiere un punto más que en la ICI para construir el polinomio de grado tres que pasa por los cuatro puntos $\{(\mathcal{F}(x_i), x_i)\}$.

Se esperaría una mejor convergencia, pero son una cantidad considerable de operaciones adicionales.

La librería gsl del proyecto GNU incluye el método de Brent dentro de sus buscadores de ceros, pero usa una variante donde la interpolación inversa es cúbica. El códigos fuente puede obtenerse en Internet.

6.5.18. Método de Crenshaw

Fué desarrollado al inicio por IBM en la década de 1960 y distribuido dentro de su librería científica, hasta ser descubierto por Jack Crenshaw, físico reconocido en el desarrollo de sistemas empotrados. El método de Crenshaw (modestamente llamado The World's Best Root Finder, CTWBRF, por su autor), es conocido ahora como TNIEBRF (The New, Improved, and Even Better Root Finder). En general alterna una iteración de bisección con otra de ICI, en vez de seleccionar la mejor entre ambas, como el de Dekker.

Lo que Crenshaw hizo fue modificar los criterios de convergencia y cambiar la estructura del método, para hacer pruebas a la interpolación antes de usarla. En el caso de los criterios de convergencia, cambió el error absoluto usado por IBM por un criterio relativo.

Para las abscisa lo determina como $\in_{\overline{x}} \in |x_n - x_{n-1}|$. En el caso de las ordenadas, como no se tienen los valores de $y = f(x_i)$ por adelantado, recurre a un esquema dinámico en el que va actualizando el mínimo y el máximo de f. en cada iteración. Con eso, es posible calcular $\in_{\overline{y}} \in |y_{\text{máx}} - y_{\text{mín}}|$.

Según Crenshaw, tras intentar combinar los dos criterios en uno sólo encontró que era suficiente con el segundo de ellos. Además, para mantener estructurado el código y actualizar sin problemas los valores límites de la función, separó la evaluación del código, mediante una función intermadia que administra $y_{\text{máx}}$ y $y_{\text{mín}}$. El código se muestra a continuación.

Listing 6.14: Función que controla la convergencia en el método de Crenshaw.

```
1 double Eval_f(double x, double Epsi, double *ymax,
2                double *ymin, bool *convergencia)
3 {
4    double y = f(x);
5    ymax = max(ymax, y);
6    ymin = min(ymin, y);
7    convergencia = (abs(y) < eps*(ymax-ymin));
8    return y;
9 }
```

Como puede verse, esta función no solo mantiene el mínimo y el máximo, sino verifica si la convergencia ya se alcanzó.

En cuanto a la estructura del método, no se intercambian siempre bisección e interpolación. Más bien se utiliza bisección y si se puede también interpolación. Al esquema básico de IBM, Crenshaw agregó más comprobaciones en la estructura, haciendo en términos generales lo siguiente.

Listing 6.15: Esquema general del método de Crenshaw.

```
for ( i=0 ; i<MaxItera ; i++ ) {
  Aplicar biseccion: m=(a+b)/2
  if ( f(b)-f(m) != 0 )
    Aplicar interpolacion
}
```

Por supuesto, cada iteración tiene muchas más operaciones. Por ejemplo, cada evaluación de la función requiere una llamada a `Eval_f()`, además de una posterior revisión de si se alcanzó la convergencia.

Aunque las bases son las mismas que en los algoritmos anteriores, Crenshaw asegura que el método es superior debido a la estructura general de alternar bisección con interpolación cuadrática inversa y a la mejora de los criterios de paro. Realmente no se alternan bisección e ICI, sino que luego de aplicar bisección, se evalúa la conveniencia de ICI antes de realizarla.

My first step in that direction has been to post the long-awaited new version of TWBRF, which should now rightly be called (TNIEBRF). The changes are both subtle enough and important enough that I need to explain them. So I'm digressing once more, but I promise to get back on topic soon.

* * *

Los dos siguientes métodos sirven de ejemplo de órdenes de convergencia más altos, que aunque ponen de condición que la primera derivada no se anule, en un intervalo que contenga a la raíz.

* * *

6.5.19. Método de Sharma-Goyal

Buscar King, R.F., 1973,Afamily of fourth-order methods for nonlinear equations. SIAM Journal on Numerical Analysis, 10, 876879.

Anonymous, 1980, Problems. BIT, 20, 261262.

En 2006, Sharma y Goyal presentan un esquema iterativo de orden sexto, donde cada iteración consta de dos pasos. El esquema es

$$\begin{aligned} x_n^* &= x_n - \frac{m\left[f(x_n)\right]^2}{f(x_n + mf(x_n)) - f(x_n)} \\ x_{n+1} &= x_n^* - \frac{mf(x_n)f(x_n^*)}{f(x_n + mf(x_n)) - f(x_n)} \frac{f(x + mf(x))\left[f(x_n) + f(x_n^*)\right]}{f(x_n^*)f(x)} \end{aligned} \tag{6.35}$$

y parte de los esquemas iterativos derivados Newton donde la derivada se aproxima de dos maneras distintas, como Steffensen en la ecuación (6.19).

El valor de m puede ser ± 1. El esquema presentado en [169] es algo más complejo pues depende de un parámetro más, una función racional en x. Sin embargo, (6.35) es el esquema iterativo más sencillo, aplicado en los ejemplos de [169].

El método de Sharma y Goyal resulta menos preciso que buscar un polinomio de grado 4 tangente a f en x_n, pero al menos no requiere las derivadas que el polinomio de Taylor necesitaría.

P Jarratt, Some effic ent fourth-order multlpo nt methods for solvmg equations, BIT 9 (1969), 119-124

R chard F Klng, A fifth-order family of modlfied Newton methods, BIT 11 (1971), 409-412

6.5.20. Método de Neta

Beny Neta utilizó el programa de cálculo simbólico MACSYMA[26] para conseguir un método iterativo de orden 6. Sólo hace uso de la primera derivada pues al inicio requiere un paso de Newton Raphson.

El esquema iterativo que Neta presenta en [142] es

$$\begin{aligned} w_n &= x_n - \frac{f(x_n)}{f'(x_n)} \\ z_n &= w_n - \frac{f(w_n)}{f'(x_n)} \frac{f(x_n) - \frac{1}{2}f(w_n)}{f(x_n) - \frac{5}{2}f(w_n)} \\ x_{n+1} &= z_n - \frac{f(z_n)}{f'(x_n)} \frac{f(x_n) - f(w_n)}{f(x_n) - 3f(w_n)} \end{aligned} \tag{6.36}$$

que obtuvo a partir de las fórmulas más generales

$$w_n = x_n - \frac{f(x_n)}{f'(x_n)} z_n = w_n - \frac{f(w_n)}{f'(x_n)} \frac{f(x_n) + af(w_n)}{f(x_n) + bf(w_n)}$$
$$x_{n+1} = w_n - \frac{f(z_n)}{f'(x_n)} \frac{f(x_n) + cf(w_n) + df(z_n)}{f(x_n) + ef(w_n) + g(z_n)}$$

Expresiones bastante grandes que son muy tardadas de obtener si no se dispone de paquetería para cálculo simbólico.

y utilizó MACSYMA para obtener las expresiones de error de w_n y z_n. A partir de las expresiones, ajusta los parámetros a, b, c, d, e y g para anular los términos de orden 5 y menores, obteniendo (6.36).

En cuanto a la cantidad de evaluaciones, el método es equivalente a dar un paso de Newton y dos de secante, lo que alcanzaría una convergencia de orden 5.2 como máximo. En el caso de (6.36), en escencia se dan tres pasos en cada iteración, ajustando la aproximación de Newton con funciones racionales.

Dar tres pasos de Newton sería más barato, por lo que el método queda mejor como ejemplo de una técnica que se puede utilizar para deducir esquemas iterativos a partir de fórmulas generales.

Consultar libros de Brent y Forsyth

[26] El proyecto Mac's SYmbolic MA'nipulation system, desarrollado hace décadas en lenguaje Lisp y utiizado principalmente para cálculo simbólico, aunque también permite algo de cálculo numérico, pero sin usar las bondades de la aritmética de punto flotante moderna.

Convergencias superiores

Los métodos descritos, tanto para polinomios como para funciones en general trabajan con órdenes de convergencia bajos, principalmente 2 y 3, aunque algunos llegan a 6.

Es posible encontrar procedimientos con orden mayor. De hecho, no hay limitante teórica con respecto al orden de convergencia que se puede alcanzar con un proceso iterativo de punto fijo (ver 6.4). Estos descubrimientos no son nada nuevos y se deben a personajes como Chebyshev, Schröder y Householder.

No tienen gran utilidad práctica, pero son buena aportación teórica. El hecho de que actualmente sea complicado

6.5.21. Iteración de Chebyshev

Este método iterativo también es conocido como Chebyshev o Super Newton:

Pafnuty Lvovich Chebyshev (1821 – 1894), gran matemático ruso, su nombre aparece en muchas áreas de matemáticas.

$$x_{n+1} = x_n - u(x) - u^2(x_n)\frac{f''(x_n)}{2f'(x_n)} \tag{6.37}$$

y puede derivarse del polinomio de Taylor. Su orden de convergencia también es cúbico, pero adolece de lo mismo que (6.15) y otros métodos cúbicos, pues requiere de la segunda derivada.

La iteración (6.37) es un caso particular de una regla más general descubierta por Chebyshev. Se basa en la representación de la inversa de $f(x) = y$, como $x = \mathcal{F}(y)$ mediante la serie de Taylor. Si x_n está suficientemente cerca de una raíz de f y si $c_0 = \beta = f(x_n)$ y $f'(x_n) \neq 0$, entonces

$$x_{n+1} = \mathcal{F}(y) = x_n + \sum_{n\geq 1} d_n(y-b)^n \tag{6.38}$$

y los coeficientes d_n se definen recursivamente de la identidad $x \equiv \mathcal{F}(f(x))$ usando los coeficientes de Taylor c_n de la función $f(x) = \sum_{n\geq 0} d_n(y-\beta)^n$. Haciendo $y = 0$ en (6.38), se obtiene la iteración

$$\begin{aligned} x_{n+1} = x_n - u(x) - u^2(x_n)\frac{f''(x_n)}{2f'(x_n)} + \\ - u^3(x_n)\left[\frac{1}{2}\left(\frac{f''(x_n)}{f'(x_n)}\right)^2 - \frac{f^{(3)}(x_n)}{6f'(x_n)}\right] + \\ - u^4(x_n)\left[\frac{5}{8}\left(\frac{f''(x_n)}{f'(x_n)}\right)^3 - \frac{5f''(x_n)f^{(3)}(x_n)}{12f'(x_n)} + \frac{f^{(4)}(x_n)}{24f'(x_n)}\right] - \cdots \end{aligned}$$

El orden de convergencia de (6.39) depende de la cantidad de términos que se empleen[27]. Si se usan dos, se tiene el caso de la iteración de Newton. De usar tres, se obtiene Super Newton (6.37).

[27] Chebyshev derivó esta fórmula a los 17 años, lo que le valió una medalla de plata y ser considerado como el candidato más sobresalientte.

6.5.22. Fórmula de Schröder

Ernst Schröder (1841 - 1902), algebrista alemán, primero en usar el término lógica matemática.

Aunque en estas fechas resulta un tanto exagerado, podía decirse.

En 1870, el Schröder publicó uno de los artículos que mayor impacto han causado en la teoría de la solución de ecuaciones no lineales, funciones iterativas. Tanto que Householder decía que si una nueva publicación no hacía referencia al artículo de Schröder, entonces posiblemente había redescubierto algo incluido allí. En su artículo, Schröder clasificó los métodos en dos grandes clases. A la primera pertenecen los métodos iterativos que aplican alguna fórmula de punto fijo como 6.4 de la página 154, de los que el método de Newton es un buen representante. La segunda clase consiste de los métodos que construyen una sucesión de funciones $F_i(x)$ cuyo límite es una raíz r que depende de la selección del valor x. El método de Bernoulli pertenece a esta clase.

El artículo fue traducido en 1992 por G. W. Stewart, quien comenta al inicio que la

Revisar tambien Review of iterative methods.

Alexander Markowich Ostrowski (25 September 1893, Kiev, Ukraine - 20 Nov 1986, Montagnola, Lugano, Switzerland).

Housholder

Householder, A. S. The Numerical Treatment of a Single Nonlinear Equation. New York: McGraw-Hill, 1970.

De hecho, Householder, en su libro *The Numerical Treatment of a Single Nonlinear Equation* de 1970, presenta un esquema más general. Si f y sus derivadas son continuas.

Tanto el método de Newton como el de Halley son casos particulares de una regla más general encontrada por Householder. Sea $g(x) = 1/f(x)$ con derivadas continuas hasta la $n+2$. Entonces la iteración definida como

$$x_{n+1} = x_n + (n+1)\frac{g^{(n)}(x_n)}{g^{(n+1)}(x_n)} \tag{6.39}$$

converge localmente a una raíz con orden de convergencia $n+2$. Newton se obtiene con $n = 0$ mientras que Halley con $n = 1$. Si ocurre que las primeras $n+2$ derivadas son continuas y no se anulan en una vecindad de la raíz, se puede obtener un método que, a partir de un valor inicial x_0 adecuadamente aproximado, en un solo paso alcance la exactitud máxima para variables en doble precisión, salvo errores de redondeo. Claro la fórmula queda más complicada que el método de Halley. Por ejemplo, para $n = 2$ se tiene la iteración

$$x_{n+1} = x_n + \frac{\frac{1}{2}(f(x_n))^2 f''(x_n) - f(x_n)(f'(x_n))^2}{(f'(x_n))^3 - f(x_n) f'(x_n) f''(x_n) + \frac{1}{6}(f(x_n))^5}$$

que se podría utilizar en un programa verificando que el denominador no es cero. Para $n = 3$ se tendría

$$x_{n+1} = x_n + 4\frac{6f^2f'f'' - 6f(f')^3 - f^3f'''}{8f'f''' + 6(f'')^2 - f^3f'''' - 36f(f')^2f'' + 24(f')^4}$$

y se ha omitido el argumento (x_n) por razones de espacio.

De esta manera se pueden generar iteraciones de cualquier orden de convergencia, pero que en la práctica tienen muchos inconvenientes.

Alexander M. Ostrowski (1893-1986)

$$x_{n+1} = x_n + 3\frac{-f(x)f''(x) + 2\left[f'(x)\right]^2}{-f(x)f^{(3)}(x) + 6f'(x)f''(x) - 6\dfrac{(f')^3(x)}{f(x)}}$$

6.5.23. Iteración de Householder

Un conocido método con convergencia cúbica se debe al matemático norteamericano Alston Scott Householder (1904-1993), más conocido por haber puesto orden en 1950 a la entonces caótica área del algebra lineal numérica. El esquema iterativo es:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\left[1 + \frac{f(x_n)f''(x_n)}{2\left[f'(x_n)\right]^2}\right] \tag{6.40}$$

y puede obtenerse de la forma racional del método de Halley, reescribiéndolo como

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\left[1 - \frac{f(x_n)f''(x_n)}{2\left[f'(x_n)\right]^2}\right]^{-1}$$

tras reemplazar $(1-a)^{-1}$ por la aproximación $1+a+O(a^2)$. Como ya es de esperarse, la tasa de convergencia baja al enfrentarse a raíces múltiples.

Otra forma de iteración puede derivarse despejando del polinomio de Taylor.

Capítulo 7

Programación de funciones: exponencial

La programación de funciones elementales es toda una rama del cómputo numérico que desde los inicios de las computadoras cobró importancia. Ya desde siglos atrás, la necesidad de hacer cálculos manuales fomentó el desarrollo de técnicas eficientes que facilitarán la publicación de tablas trigonométricas y logarítmicas.

Los logaritmos son desde entonces un recurso invaluable al momento de hacer cálculos pues reducen problemas complicados a versiones ms sencillas debido a sus propiedades. Las tablas con logaritmos se popularizaron hasta la llegada de las calculadoras de bolsillo.

Los logaritmos fueron inventados por John Napier (1550–1617), matemático escocés.

Desde finales de la década de 1940 a la fecha, ha habido grandes avances en la programación de funciones elementales, por los trabajos de excelentes matemáticos como Hirondo Kuki[1], James Cody, William Kahan y más recientemente Jean Michel Muller. Al inicio, se tenía que hacer todo el trabajo con software pues no haía un estándar como el 754 de IEEE que comprometiera ciertos requisitos mínimos en los procesadores.

En los 70s se discutió la posibilidad de que las funciones elementales estuvieran en el procesador mismo y no en la biblioteca de funciones del compilador, por ejemplo [148]. La carencia de normas que facilitaran las decisiones dio pie discusiones como la publicada en [15] que abogaban por una norma para funciones elementales[2]. La sugerencia no se cumplió pues al año siguiente se aprobó la norma 754 de IEEE y la historia cambió.

De cualquier manera, en 2004 se publicó [47], aprovechando la revisión de la norma de punto flotante. Los autores (Defour, Zimmermann y Muller entre otros) proponen las bases de una norma para la programación de funciones matemáticas, considerando gran cantidad de aspectos. Lo central radica en los tres niveles de redondeo en que se agrupan las funciones.

[1] Vale la pena la lectura del *In memoriam* del padre de la aritmética de punto flotante tras la muerte de Kuki [99]. Kuki hizo de la mediocre librería matemática de Fortran la mejor del momento, ayudado por Cody, trabajo reportado en [119].

[2] En ese entonces, un error de 2 ulp era admisible para las funciones elementales.

Desde el principio los programadores se dieron cuenta de que las rutinas que aproximaran funciones trascendentes tenían que hacerse con muchos cuidados. La exactitud teórica de polinomios como el de Taylor son un verdadero desastre cuado se realizan mal. Muchas estrategias, como las presentadas en el capítulo 4 fueron empleadas, algunas con gran eficiencia.

Aunque se pensó en la posiblemente alta complejidad de rutinas eficientes, comparadas con las operaciones aritméticas, la experiencia ganada en los primeros años mostró que no era del todo as. En 1984, Alt demostró en [7] que las funciones elementales eran tan complejas como la división, en términos del tamaño del circuito lógico necesario para evaluarlas, para una salida de n bits la complejidad es $O(\log n)$ en la profundidad del circuito. Su resultado es para punto fijo, pero aplica igual para punto flotante.

Cuatro años después, publicó otro artículo [8] donde acota aún más la complejidad para el caso de las funciones algebraicas, mostrando que es equivalente a la multiplicación (la misma profundidad de $O(\log n)$ con un tamaño de $O(n \log n \log\log n)$ simultaneamente[3]).

Cada función elemental tiene características particulares, que motivaron diversas técnicas de programación. En general pueden agruparse dependiendo de sus propiedades y el resultado ha sido un gran conjunto de cada vez mejores aproximaciones en busca de la cota ideal: el $\text{ulp}(f(x))$ correctamente redondeado.

Este capítulo presenta una variedad de formas de programar rutinas que aproximen la función exponencial e^x y diversas maneras de probarlas, apoyándose en los principios básicos generales.

Pudiera haberse utilizado cualquier otra función, pero esta es sólo una muestra representativa de las técnicas que se pueden aplicar para resolver problemas eficientemente. El lector debe referirse a libros como [38, 140] para mayor información. Trigonométricas, hiperbólicas o logaritmos hubieran ofrecido un bufet similar.

7.1. Fundamentos de programación de funciones

Antes de abordar el caso particular de la función exponencial, es conveniente revisar, al menos superficialmente la teoría general. Aunque las funciones elementales se definen sobre el conjunto de los números reales, los únicos resultados posibles están en el itervalo $[-2^{1022}, 2^{1022}]$. Los resultados mas pequeños en magnitud deben ser mayores en valor absoluto que 2^{-1074} o ser cero[4].

Como son mucho más complejas que las operaciones elementales, la norma de 1985 no exigía que las funciones elementales dieran resultados correctamente redondeados hasta el último bit. Pero en la revisión de 2007, los avances en las investigaciones son tales que ya aparece ese requisito para algunas de estas. Una técnica generalizada es la que presenta Ziv en [192], que se comenta más adelante.

Ya son públicos algoritmos que calculan la mayoría de las funciones elementales, con algunas excepciones, por lo que todo programador debe asegurarse de trabajar

[3] La división también tiene ese tamaño, pero aún se desconoce si ocurre simultanemente con profundidad $O(\log n)$.

[4] Esto aún es cierto debido a que la nueva norma aún no entra en vigor. El rango se ampliará con la llegada de la precisión cuádruple.

con un compilador que las incluya en su biblioteca de funciones.

Los pasos típicos de la aproximación de una función elemental son los siguientes.

1. Reducción de rango, que pueden ser por etapas.
2. Aproximación por tabla y algn polinomio o función aproximante.
3. Reconstrucción, que solo revierte de manera segura la reducción.

La programación de funciones matemáticas elementales posee una basta teoría y está llena de detalles. Las siguientes secciones ofrecen una idea básica del camino que se debe seguir. La eficiencia en términos de velocidad y precisión es directamente proporcional a lo sofisticado de los algoritmos.

7.1.1. Fórmulas y propiedades

El primer recurso es buscar fórmulas, identidades o propiedades matemáticas. An sabiendo que las aproximaciones no pueden ser exactas debido a que se trabaja con $\mathbb{F}$ y no con $\mathbb{R}$, es deseable que se cumplan la mayor parte de las propiedades de las funciones elementales. Es de aquí de donde surge la batería de pruebas con que se estudia su desempeño, exactitud y precisión.

Una de las propiedades más importantes es la monotonicidad. Una función f es *monotónica* en un dominio D (que puede ser un simple intervalo) si se cumple que $x < y$ implica $f(x) < f(y)$.

Idealmente, si $x \in \mathbb{F}$ es un número de punto flotante y $f : \mathbb{R} \to \mathbb{R}$ es una función estrictamente creciente, entonces se espera que

$$f(x) < f(x + \text{ulp}(x)). \tag{7.1}$$

Por la precisión limitada de las computadoras, el $<$ de (7.1) se relaja a $\leq$. Claro, si una función es estrictamente creciente, se espera que su aproximación sea al menos creciente y no ocurra que exista $x \in \mathbb{F}$ tal que $f(x) > f(\text{nextUp}(x))$. $f : \mathbb{F} \to \mathbb{F}$

La limitante anterior es clara si se piensa en una función como $\sqrt{x}$, que mapea el intervalo $[0, 4]$ al $[0, 2]$, donde existen la mitad de elementos de $\mathbb{F}$. Imposible suponer que no existan $x, y \in \mathbb{F}$, con $x < y$ tales que $\text{fl}(\sqrt{x}) \neq \text{fl}(\sqrt{y})$.

A este respecto, Dunham publicó [53] en 1987, donde muestra las dificultades de probar la monotonicidad de funciones en general, especialmente en el caso de polinomios de grado alto. Tres años después, Dunham publicó artículo de dos páginas, bastante pesimista, [54] donde pone en duda la posibilidad de logra la cota perfecta en la evaluación de funciones elementales[5].

Otra propiedad importante para algunas funciones es la *simetría*, que implica que $f(-x) = f(x)$ o la *antisimetría*, $f(-x) = -f(x)$. La función seno es antisimétrica y coseno es simétrica.

Algunas funciones poseen otra agradable propiedad: son *periódicas*. Eso significa que $f(x) = f(x + k)$ con $k \in \mathbb{R}$ conocido como el periodo. Es el caso de algunas funciones trigonométricas, cuyo periodo es $\pi/2$.

[5] Incluso se queja de que Kahan sí garantiza .5 ulp pero sin mostrar las técnicas con que se logra.

Aunque las anteriores propiedades son generales, no debe olvidarse las propiedades particulares que cada función puede tener. Todo elemento teórico es útil para diseñar rutinas eficiente o para probarlas.

7.1.2. Reducción de rango

Este es el primero de los pasos que debe realizar un programa que aproxime una función elemental y es realmente crítico. Por ello esta sección se extiende más que las otras en esta introducción a la programación de funciones.

La *reducción de rango* consiste en aplicar propiedades y transformaciones algebraicas de tal manera que el rango del argumento se acote a un subconjunto donde se garantice una mejor aproximación. El ejemplo típico es que no se requiere programar la función seno en todo $\mathbb{R}$ pues $\operatorname{sen} 370^\circ = \operatorname{sen} 10^\circ$. De hecho, con ajustar el argumento a $[0, \pi/4]$ es suficiente.

El argumento original x debe reducirse a un nuevo valor más pequeño y. Si el *argumento reducido* y se encuentra en un intervalo $[-t, t]$ se llama *reducción simétrica*. En caso de que el intervalo sea $[0, t]$ se llama *reducción positiva*. El intervalo final es llamado *intervalo de convergencia*. Normalmente no se es tan estricto con los límites y en la práctica hay cierto valor ϵ que se pueden extender en ambos sentidos, dependiendo de la reducción, facilitando aún más las cosas.

Además del tipo de la forma del intervalo, hay dos tipos básicos de reducción de rango, la aditiva y la multiplicativa, conocidas desde hace mucho tiempo. En la *reducción aditiva* se busca un entero k tal que el argumento reducido $y = x - kC$ quede en el rango deseado y C es una constante que depende de la función. Eso es lo que se hace en el caso de la función seno, donde puede hacerse $C = \pi/2$.

En la *reducciónn multiplicativa* el argumento reducido adopta la forma $y = x/C^k$, pero ahora C se elige como la base numérica. Por ejemplo, en el caso de $\ln(x)$, se busca k tal que $y = x/2^k \in [.5, 1]$ y evaluar $\ln(x) = \ln(y) + k \ln(2)$, por lo que puede hacerse $C = \ln 2$.

Cualquiera que sea la idea que se aplique, debe entenderse que debido a las operaciones aritméticas realizadas, la equivalencia matemática no está garantizada. Basta pensar que el redondeo de la expresión $x - k\pi/2$ puede tener un error mayor a .5 ulp. En general, hacer $k = \lfloor x/C \rfloor$ y $x - kC$ es desastroso si $x \approx kC$.

Otro problema grave es que posiblemente el argumento x fuera mucho muy grande, molestia de las funciones trigonométricas en particular. Podían requerirse muchos bits adicionales de π para asegurar que la etapa de normalización no fuera perjudicial.

Ante estos problemas, los programadores de rutinas matemáticas comenzaron a desarrollar técnicas seguras de reducción de rango, algunas para casos especiales y otras orientadas a la mayoría de los casos.

El dilema de los constructores de tablas

Utilizar precisión múltiple puede resultar mucho más costoso, por lo que no es una solución adecuada, además de que no es trivial valorar cuánta precisión se necesita. El problema exacto se conoce como el *table maker's dilemma* (TMD) y es el siguiente.

Acuñado así por Kahan en 1974.

Se desea garantizar que la aproximación de $f(x)$ está correctamente redondeada, es decir, $\text{fl}(f(x)) = f(x)(1+\delta)$ con $\delta < \mu$, para todos los modos de rdondeo. La mantisa de la aproximación de $f(x)$ antes de la renormalización tendrá q bits extra en total, $q > p$, que son necesarios para redondear correctamente y obtener $\text{fl}(f(x))$ con un error de .5 ulp.

Redondear no es trivial pues pueden ocurrir los siguientes casos. Si el modo de redondeo es al más cercano, el resultado puede estar muy cerca del punto medio de los dos NPF más cercanos a $f(x)$, más de lo que la precisión intermedia permite determinar. Por ejemplo, si la mantisa de $f(x)$ es

$$\overbrace{.d_1d_2d_3...d_p1000...0}^{q \text{ bits}}b_1b_2b_3... \quad \text{o} \quad \overbrace{.d_1d_2d_3...d_p0111...1}^{q \text{ bits}}b_1b_2b_3... \tag{7.2}$$

antes de la renormalización. $b_1 = 1$ basta para cambiar el redondeo. De haber usado $q+1$ bits no habría habido duda. Ese es realmente el TMD, no saber cuántos se requieren. Los modos de redondeo direccionado no se salvan pues

$$\overbrace{.d_1d_2d_3...d_p000...0}^{q \text{ bits}}b_1b_2b_3... \quad \text{o} \quad \overbrace{.d_1d_2d_3...d_p111...1}^{q \text{ bits}}b_1b_2b_3... \tag{7.3}$$

dan problemas similares en `roundUp` y `roundDown` respectivamente, o el correspondiende redondeo a cero dependiendo del signo.

El problema persiste: ¿cuántos bits son necesarios? Si la aproximación intermedia no es suficientemente exacta, se tiene algunos de los casos anteriores de mantisa. Kahan sugirió un método que usa fracciones continuas para encontrar los peores casos de (7.2) y (7.3).

Si $x = .x_1x_2...x_p \times \beta^e$ y se escribe con mantisa entera como $x = M \times \beta^{e+1-p}$, entonces la reducción de rango aditiva equivale a encontrar un entero k y un número real $|r| < 1$ tales que

$$\frac{x}{C} = k + r \tag{7.4}$$

y $y = rC$ es la reducción de rango. La ecuación (7.4) puede reescribe como

$$\frac{\beta^{e+1-p}}{C} = \frac{k}{M} + \frac{r}{M}. \tag{7.5}$$

Si la reducción de rango y es muy pequeña en magnitud, r también debe serlo, por lo que el lado izquierdo de la ecuación (7.5) debe ser una muy buena aproximación de k/M, dándose una de las situaciones (7.2) o (7.3). Lo que se requiere para encontrar el menor valor posible de y es analizar la sucesión de la mejor aproximación racional posible a $\frac{\beta^e+1-p}{C}$, lo que se hace con fracciones contínuas.

Los detalles están disponibles en el preámbulo del programa `nearpi.c`, escrito por McDonald a partir de una versión en Basic de Kahan [101]. En su libro [140], Muller incluye los peores casos para algunas funciones elementales en diversos rangos[6], para precisión doble de IEEE.

El remedio práctico, sugerido inicialmente por Gal y Bachelis en [69] con ajustes finales de Ziv en [192], es aproximar $f(x)$ de nuevo con un valor mayor de q y repetir

[6]Extracto de [123] (o http://perso.ens-lyon.fr/jean-michel.muller/TMDworstcases.pdf, que es una versión más actualizada).

hasta asegurar el redondeo. La primera aproximación puede hacerse con las operaciones nativas del procesador, pero las siguientes requerirán de precisión múltiple. En la práctica, el redondeo correcto puede hacerse con una o dos iteraciones en la mayoría de los casos.

La importancia de determinar los peores casos para cada función elemental (el valor máximo de q) radica que eso acota el problema pues ya se tendría por adelantado la precisión máxima necesaria por función y por dominio, resolviendo el TMD. Logrado eso ya se puede diseñar una evaluación tal que el resultado final difiera del infinitamente correcto en .5 ulp como máximo.

Dentro de las funciones elementales donde los peores casos no han sido determinados están x^n y x^y, por lo que aún no existen algoritmos que alcancen la cota perfecta.

Reducciones de rango más eficientes

Para resolver losproblemas intrínsecos de la reducción de rango, se han desarrollado varias técnicas. En los años recientes, los procedimientos hacen uso de la norma de punto flotante para mejorar la eficiencia.

A la Dekker.

En el libro [38], Cody y Waite sugieren encontrar dos valores C_1 y C_2 con representación exacta en $\mathbb{F}$ tales que $C = C_1 + C_2$, con $C_1 \approx C$ usando pocos bits. De esta manera, la evaluación $x - kC$ se realiza con mayor exactitud con

$$y = (x - kC_1) - kC_2 \tag{7.6}$$

lo que simula una precisión mayor que la nativa del procesador.

En escencia, es un método rápido que sirve para reducir la cancelación de cifras. Puede extenderse a usar 3 constantes con poca complejidad temporal adicional.

Posteriormente, en 1983 Payne y Hanek publicaron en [151] un algoritmo para la reducción de rango aditiva más seguro, que se aplicó de inmediato a la libraría matemática de las computadoras VAX. En ese entonces, la norma 754 de IEEE no se había aprobado y como VAX era la compañía que más se negaba a aceptar el underflow gradual, este método de reducción no dependía de la representación de los NPF.

El trabajo fue desarrollado pensando en argumentos muy grandes en las funciones trigonométricas, por lo que $C = \pi/4$. El argumento reducido y debe quedar en $[0, C]$. EL algoritmo tiene tres características importantes:

1. En el caso de argumentos muy grandes, evita los cálculos con aquellos bits que de cualquier manera serásn descartados.
2. Utiliza bits adicionale alrededor de las raíces de la función que se evalúa, donde generalmente es crítico el reultado.
3. En los otros casos, la complejidad algoritmica es independiente del tamaño del argumento.

Siguiendo la formulación inicial, sea $r = 2\pi/2^{-i}$ e i toma valores de 0 a 4, dependiendo si se considera el periodo entero, la mitad, un cuadrante o un octante. El objetivo es determinar el entero no negativo j, un entero l y un valor h tales que cumplan con

$$0 \leq l < 2^i$$
$$0 \leq h < 1$$

y de tal forma que

$$x = r(j2^i + l + h) \tag{7.7}$$

con lo que se está separando el cociente x/r en dos partes enteras y una real. Es importante notar que l tiene i ceros y es un múltiplo de 2π, por lo que no contribuye en el caso del seno o coseno.

Con la actual norma aprobada, considerando los p de bits de mantisa, x puede escribirse en términos de un entero M como $M \times 2^{e-p+1}$, donde, en este caso, e no está recorrido como en la norma. M satisface $2^{p-1} \leq M \leq 2^p$.

Como la reducción de rango en general puede escribirse como $y = \pi/4(4x/\pi - k)$, el valor $\alpha = 4/\pi$ toma gran importancia pues la reducción de Payne y Hanek separa la mantisa de αx en tres partes, que juntas forman la cadena de p bits del formato.

La separación queda

$$\alpha x = 8Mp_1(x) + 2^w Mp_2(x) + 2^w Mp_3(x)$$

donde w depende de la precisión que se necesita, permitiendo separar de manera conveniente en tres la mantisa y $p_i(x)$ son las partes de la mantisa ya separada de x. Como $8Mp_1(x)$ es múltiplo de 8, no influye en la evalución de la función trigonométrica. Por su parte, $2^w Mp_3(x)$ puede hacerse tan pequeño como se desee.

Algunos expertos consideran que el algoritmo de Payne y Hanek es exacto pero adolece de requerir demasiadas operaciones para garantizar tal exactitud, por lo que se utiliza exclusivamente para argumetos grandes ($x > 2^{40}$).

En 1994, fue descrito por Daumas el algoritmo que se conoce como *reducción modular de rango* (RMR), pero su análisis completo se publicó en [46] un año después, por los mismos 4 autores. El método no es exclusivo para funciones trigonométricas.

RRM se basa en reescribir el argumento en punto fijo base 2, quedando

$$x = x_{n-1}x_{n-2}\ldots x_0.x_{-1}x_{-2}\ldots x_{-p}$$

que representa al número $\sum_{i=-p}^{n-1} x_i 2^i$.

Sea v un entero que cumple con $2^v < C \leq 2^{v+1}$ y se define $m_i \in [-C/2, C/2)$ tal que $(2^i - m_i)/C$ es un entero, con $i \geq v$.

El primer paso de RMR es calcular

$$r = x_{n-1}m_{n-1} + x_{n-2}m_{n-2} + \cdots + x_v m_v + x_{v-1}\ldots x_{-p}$$

que queda en el intervalo $[-\frac{C(n-v+2)}{2}, \frac{C(n-v+2)}{2}]$.

Para el segundo paso, sean r_i los dígitos binarios de r, $\epsilon > 0$ un número pequeño y s los primeros bits de r descartando los últimos $\lceil -\log(\epsilon) \rceil$.

s tiene $\left\lfloor \log(\frac{C(n-v+2)}{2}) \right\rfloor + \lceil -\log(\epsilon) \rceil$ bits, que en general se espera que sea un número pequeño.

Ahora se tienen dos opciones, dependiendo de la reducción deseada. Si k se selecciona como el entero más cercano a s/C, entonces $r - kC$ queda en $[-C/2 - \epsilon, C/2 + \epsilon]$.

Si se hace $k = \lfloor s/C \rfloor$, entonces $r - kC \in [-\epsilon, C/2 + \epsilon]$. Esto permite seleccionar reducción simétrica o positiva con facilidad.

El segundo paso puede programarse como una simple búsqueda en tabla, sin hacer ningún cálculo. El mayor costo del algoritmo es la suma de los $n - v + 1$ términos de la primera reducción. La extensión de RMR para números de punto flotante es realmente simple, simplemente reemplazando la suma de los m_i por la suma de m_{e-i}, donde e es el exponente de la representación flotante. Como antes, $m_i = 2^i \mod C$.

El método de Payne–Hanek es el adecuado para grandes argumentos, mientras que RMR lo es para pequeños ($x < 10$). En 2005 se publicó un nuevo método adecuado para argumentos de mediano tamaño en [19].

Lo complicado del método lo deja más allá de las intenciones de esta introducción a los métodos para reducción de rango. El artículo [19] presenta en su primera parte el algoritmo de Payne–Hanek, así como un estudio estadístico de la distribución de los argumentos reducidos en el intervalo de convergencia y la forma de analizar sus peores casos (el TMD de la sección anterior, que los autores definen como los valores del argumento x para los que se obtienen los menores valores del argumento reducido y, en valor absoluto.).

En la segunda parte, presentan el algoritmo, planteado para una reducción aditiva con $C = \pi/2$, pero se asegura que la extensión a otras constantes es directa. El que se separa el argumento en 8 partes I_i de 7 bits cada una. Aplicar una primera reducción con

$$\begin{aligned} S(x) =& I_0(x) \mod \pi/2 + 2^8 I_1(x) \mod \pi/2 + 2^{16} I_2(x) \mod \pi/2 + \\ &+ \cdots + 2^{56} I_7(x) \mod \pi/2 + \rho(x) \end{aligned}$$

y $\rho(x)$ es exactamente representable en precisión doble. Si $x > 2^{52}$, entonces $\rho(x) = 0$. El valor $x - S(x)$ es un múltiplo de $\pi/2$. A $S(x)$, se le aplica una segunda reducción de rango. Los cálculos indican que para alcanzar el objetivo de redondeo correcto aún en los peores casos, se requieren 153 bits.

Los valores $|2^{8i} I_i(x)| \mod \pi/2$ deben guardarse en tablas para garantizar la rápida reducción de rango. Cada valor requiere 3 valores en precisión doble para mantener los 153 bits. La reducción también queda almacenada en tres variables, por lo que el resultado final se obtiene con algoritmos suma de precisión múltiple. EL sugerido por los autores es el de Knuth en [117], donde la suma exacta de $a + b$ se almacena en dos variables $z + r$.

El argumento reducido final y se representa con dos variables, de tal forma que $y = y_h + y_l$ con una precisión de 2^{-87}, que para el rango del intervalo de convergencia es mejor que la cota perfecta.

Actualmente, una buena rutina de reducción de rango decidirá la técnica de reducción más conveniente, dependiendo del argumento x. La reducción de rango puede repetirse varias veces, hasta llegar al intervalo donde la aproximación polinomial sea tan confiable como se requiere.

7.1.3. Tipos de aproximaciones polinomiales

En 1885, Karl Weierstrass demostró[7] que toda función $f : \mathbb{R} \to \mathbb{R}$ continua en un intervalo $[a, b]$ puede ser aproximada por polinomios que pasan arbitrariamente cerca

[7] La prueba original es para funciones analíticas, es decir, definidas sobre el plano complejo $\mathbb{C}$. Existen muchas generalizaciones, dentro de ellas, una muy importante es el teorema de Stone-Weierstrass, en espacios de Banach.

de la función. Es decir, dado $\epsilon > 0$, existe un polinomio p que cumple con

$$\|f(x) - p(x)\| < \epsilon$$

para toda $x \in [a, b]$.

El teorema de Weierstrass es de gran importancia tanto teórica como práctica pues garantiza que con operaciones elementales se pueden aproximar funciones continuas de cualquier tipo.

Un corolario sencillo es que existe una infinidad de aproximaciones polinomiales, así que el problema se reduce a seleccionar la más adecuada para ser programada. Existen dos criterios básicos de selección, minimizando ya o sea el error medio o la distancia máxima.

Minimizar el error medio significa encontrar la aproximación p tal que la distancia $d(f, p) = \|f(x) - p(x)\|$ sea mínima en el dominio de interés $[a, b]$. Una forma común de definirla es

Es una aproximación de mínimos cuadrados.

$$d(f, p) = \sqrt{\int_a^b (f(x) - p(x))^2 \, dx}$$

pero no es la única manera. Polinomios importantes de esta familia son los de Chebyshev, Laguerre, Jacobi o Legendere.

La otra forma es *minimizar el error máximo*. En este caso, se define la distancia como

También conocida como aproximación *minimax*.

$$d(f, p) = \max_{a \le x \le b} \|f(x) - p(x)\| \tag{7.8}$$

y el polinomio más conocido es el de Taylor. El error lo acota el residuo, que es el último término de la ecuación (1.4) de la página 16.

Es más complicada esta técnica, por lo que se recurre no solo a polinomios. De las más conocidas son las aproximaciones racionales de Padé, que son cocientes de polinomios que se obtienen a partir del de Taylor. La aproximación de Padé se denota como $P_{m,n}$ y consiste en un polinomio de grado m sobre uno de grado n que aproximan a la función tan bien como el polinomio de Taylor de grado $m+n$. Más detalles pueden consultarse en textos como [72, 116].

El método estándar para encontrar la mejor aproximación en espacios de Chebyshev (de polinomios) es el algoritmo que Evgeny Y. Remez publicó en 1934, utilizado por paquetes como Matlab, Maple o Mathematica. El polinomio encontrado con el algoritmo de Remez se llama polinomio de Chebyshev.

Detalles de implementación pueden consultarse en el artículo de Fraser [64]. La idea básica es interpolar la función en un conjunto de puntos iniciales (los nodos de Chebyshev) e irlos ajustando aplicando el teorema de alternancia de Chebyshev hasta lograr el mínimo error con el criterio 7.8. Detalles adicionales sobre la selección de los puntos iniciales está en el artículo de Dunham [52].

El método es tan importante que se enuentra fácilmente en textos de deducación superior como [25, 116]. Comunmente está acompañado de la teoría de subespacios de Haar.

La manera de encontrar la mejor función aproximante depende de cuál de las dos estrategias se seleccione.

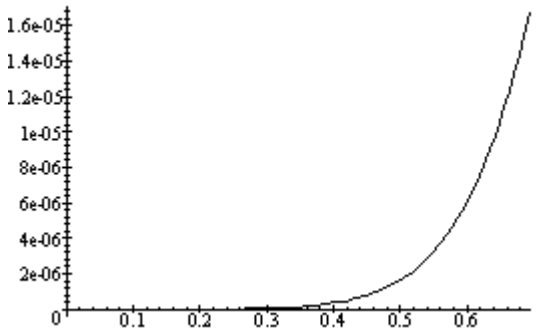


Figura 7.1: Diferencia entre la función exponencial y el polinomio de Taylor p de grado 6 en $[0, \ln(2)]$. Se grafica la diferencia $|e^x - p(x)|$. Es común que el mayor error esté en un extremo del intervalo.

7.1.4. Apoximación por tablas

Si se dispone de memoria infinita, se podrían evaluar por anticipado las funciones de interés en cada NPF y sólo consultar cuando se necesiten. Aún cuando la memoria es barata, no lo es en ese grado, por lo que este esquema es impráctico.

Sin embargo, tener ciertas evaluaciones exactas en el intervalo de convergencia es indispensable. Tales evaluaciones definen subintervalos no necesariamente del mismo tamaño. Con esas aproximaciones exactas se pueden construir polinomios aproximantes que se ajusten a cada subintervalo, almacenando en una tabla los coeficiones que correspondan. Esta técnica se ha usado desde finales de los aos 60.

Con esos polinomios la aproximación alcanza la convergencia en una cantidad fija de iteraciones. Los coeficientes se seleccionan dependiendo del subintervalo al que pertenezca el argumento reducido y. Por ejemplo, si e^x se aproxima en el intervalo $[0, \ln(2)]$ con un polinomio de Taylor truncado de grado 6 centrado en cero

$$p(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720}$$

la diferencia $|e^x - p(x)|$ es $O(10^{-5})$ y su gráfica se presenta en la figura 7.1. Es evidente que el error no está distribuido uniformemente, sino en un solo extremo del intervalo en este caso.

Entre más subintervalos se tengan se requerirá de polinomios de menor grado, lo que hace más veloz los procesos al tiempo de necesitar mayor cantidad de memoria. Un buen diseño se basa en el equilibrio entre complejidad temporal y espacial.

Los puntos que delimitan los subintervalos y la evaluación exacta de la función en los mismos forman la tabla a la que se refiere el título de esta sección. Hay tres formas básicas de diseãrlas.

1. Las *tablas normales* son aquellas en las que el intervalo de convergencia se subdivide en partes iguales, típicamente una potencia de dos. Por ejemplo, si se

separa en $2^5 = 32$ partes, entonces los primeros 5 bits del argumento reducido servirán como el índice en la tabla de 32 valores. Son sencillas de programar, con la desventaja de requerir una precisión mayor que la nativa para conseguir una aproximación con un error máximo de .5 ulp. El mejor representante es el algoritmo de Tang [176, 177].

2. Las *tablas exactas* no dividen el intervalo de convergencia en partes iguales, ya que los puntos de división son frecuentemente números reales de forma $iC/2^k$, con i y k enteros. Como normalmente C es trascendente, las subdivisiones deben redondearse, por lo que ya acarrean un error. La idea de las tablas exactas es subdividir el intervalo de tal manera que cada punto de división x_i se aun número de $\mathbb{F}$. Se espera que las subdivisiones estén distribuidos lo mas uniformemente posible. Este método no requiere el uso de precisión extra como en el caso de las tablas normales.

   El representante típico es el algoritmo de Gal, descrito en [69] junto con todos los detalles de las tablas exactas (pasando por una reducción de rango similar a las presentadas y la aproximación minimax), evitando el uso de precisión extra, pero calculando el ulp(x) correctamente redondeado. En dicho artículo, Gal y Bachelis construyen todo sobre la norma de punto flotante de IEEE, consiguiendo que el 99.7 % de los resultados esté correctamente redondeado[8].

   La idea de Gal fue extendida con éxito por Ziv en [192], logrando el 100 % de los resultados correctamente redondeados. Se utiizaron dos reducciones de rango y precisión extra cuando se detectaba algún caso de redondeo problemático.

3. Las tablas múltiples se componen de tablas en varios niveles. La idea es usar el argumento reducido para determinar el subintervalo de la primera tabla. Una vez determinado, el argumento se usa de nuevo para seleccionar el siguiente subintervalo. Puede haber tantas tablas como se necesiten. En la práctica estas tablas requieren implementarse en hardware, por lo que no se describirá ningún método.

No se presentan más detalles pues se utilizarán con la función exponencial.

7.1.5. Pruebas de exactitud

Las pruebas de eficiencia son más complicadas que el objeto de la prueba. El principal problema radica en el marco de referencia: ¿contra qué se compara? Si se compara contra las rutinas existentes, no se está suponiendo que sean eficientes, solo son el punto de comparación, por lo que la respuesta será relativa.

Medir la velocidad puede ser mas sencillo pues las funciones que miden el tiempo son fáciles de utilizar. Medir la exactitud requiere conocer los resultados correctos para medir la exactitud de las funciones programadas. Cómo obtener esos resultados correctos para comparar es el problema.

Una manera es basarse en las propiedades de las funciones programadas y ver que las identidades se cumplen, salvo los errores de redondeo de las operaciones que implican. La selección de tales propiedades no se sencilla.

Lo mejor es construir una tabla de valores correctamente redondeados utilizando precisión múltiple y luego comparar los resultados de las rutinas programadas. Es la manera más sencilla de verificar la exactitud.

[8] A partir de una muestra de 300 000 evaluaciones.

Los valores especiales deben asegurse, por lo que la evaluación correcta en ∞, $-\infty$, 0, ceros y asíntotas es importante. Además, toda prueba debiera incluir valores dentro de los peores casos del TMD (ver sección 7.1.2).

La manera más ampliamente utilizada para evaluar la exactitud es comparar las rutinas contra resultados calculados utilizando mayor precisión, como en [177]. La mayor complicación es que el programa de evaluación generalmente no es portable, además de que no puede aplicarse cuando la rutina que se prueba está en la máxima precisión posible. Claro, siempre es posible simular la precisión adicional con técnicas como las de Dekker.

En [37], Cody sugiere cómo usar polinomios de Taylor para evaluar la exactitud de rutinas y evitar el uso de precisión extra.

* * *

Como segunda parte de este capítulo se presentan una diversidad de maneras de programar la función exponencial. Las primeras versiones son tan sencillas como inocentes, pero la idea es ilustrar las distintas técnicas de programación posibles, poniendo en evidencia aquellas que resultan deficientes.

Aunque hay métodos muy sofisticados que alcanzan la cota perfecta de $fl(e^x) = e^x(1+\delta)$ con $|\delta| \leq \mu$, estos no se muestran con todos los detalles. Si el lector desea estudiarlos, se sugiere ir a las referencias que aparecen al final del capítulo.

7.2. Propiedades básicas de e^x

La función exponencial es bastante conocida y se sabe de gran cantidad de propiedades. Aparece constantemente en identidades de todo tipo. Para efectos de cómputo interesan las propiedades que involucran pocas operaciones.

En resumen, se puede decir que la función $\exp(x) = e^x$ es real[9], trascendente, definida positiva, monotónica y estrictamente creciente. Además cumple con propiedades útiles (algunas) al momento de probar si una rutina que la aproxime trabaja bien, como son las siguientes.

$$e^0 = 1 \tag{7.9}$$

$$e^1 = \lim_{n\to\infty} (1+\frac{1}{n})^n \tag{7.10}$$

$$= 2.71828182845904523536\,0287471352\ldots$$

$$e^{a+b} = e^a e^b \tag{7.11}$$

$$e^x < e^y \qquad \text{siempre que} \qquad x < y \tag{7.12}$$

$$\tag{7.13}$$

Otra propiedad importante es que su derivada es igual a sí misma, es decir, la recta tangente en el punto x tiene pendiente e^x.

Numéricamente, si se considera la precisión doble de IEEE, no puede calcularse e^x para todo NPF pues se causa overflow o underflo. La tabla 7.1 muestra los valores que debe regresar la función exponencial en diversos intervalos.

[9] La definición para variable compleja $e^z = e^{a+ib} = e^a(\cos b + i \operatorname{sen} b)$ no se analizará.

Valor de x	Resultado de e^x
$709.7827 < x$	$+\infty$
$-708.396 \leq x \leq 709.7827$	Número normal
$-745.1332 \leq x < -708.396$	Número subnormal
$x < -745.1332$	0
$-2^{-54} \leq x \leq 2^{-53}$	1
NaN	NaN

Tabla 7.1: Los rangos válidos para evaluar la función e^x en $\mathbb{F}$, considerando precisión doble. Los valores son meras aproximaciones.

Por ser una función tan importante, desde el inicio de la computación se han programado aproximaciones cada vez mejores. Desde polinomios de Taylor, fracciones contínuas y todo tipo de esquemas.

7.3. Aproximación de e^x con series de potencias

Lo más sencillo es generar el polinomio de Taylor con

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \cdots \tag{7.14}$$

y evaluarlo en los puntos donde el usuario indique. Aunque en los libros de cálculo se diga que aumentando la cantidad de términos se reduce el error y se mejora la aproximación lejos de $x = 0$, esta no es una buena idea.

Tan solo para garantizar un error máximo de .5 ulp (un error absoluto máximo de 2^{-53}) en el intervalo $[-1, 1]$, se requieren 18 términos de (7.14), pues con 17 se tiene un error acotado por $1.68 \times 10^{-16} \approx 2^{52.4}$, es decir, casi .8 ulp. Sin embargo, con esos 18 términos, el error en $x = 2$ aumenta a 4.53×10^{12}.

Lo anterior significa que una serie de potencias u otra aproximación que se emplee, no obtendrá buenos resultados a menos que se acompañe de una adecuada reducción de rango. Una versión bastante deficiente se muestra en el código 7.1.

El código 7.1 es el cálculo directo de la fórmula (7.14) en la forma como típicamente lo hace un programador novato. Va calculando cada término como el cociente de la potencia y la actualización del factorial anterior (líneas 9 y 10) y lo agrega a la suma parcial `r` (línea 15). Se detiene al cumplirse una de tres condiciones:

1. El siguiente término ya no modifica la suma parcial (línea 11).
2. El siguiente término es menor que cierta tolerancia aceptable (línea 13).
3. Se alcanza el máximo de iteraciones. `MaxTerm` debe definirse previamente.

Pese a que en algebraicamente los los cálculos, esta versión adolece de varias cosas, las más notables son las siguientes.

1. Lo más crítico es el uso de la función `pow`, incluida en las bibliotecas de funciones para calcular la complicada función x^y y no una potencia entera.

Los programadores principiantes abusan de esta función constantemente.

Listing 7.1: Versión deficiente de la función exponencial.

```
1  double exp_v0(double x)
2  {
3      int i;
4      double r;     /* Resultado final */
5      double Fact=1.0, Term;

7      r = 1;
8      for ( i=1 ; i<MaxTerm ; i++ ) {
9          Fact *= (double)i;
10         Term = pow(x,i)/Fact;
11         if ( r+Term==r )
12             break;
13         if ( fabs(Term) < Tol )
14             break;
15         r += Term;
16     }
17     return r;
18 }
```

2. El cálculo por separado de x^n y $n!$ tiene riesgo de overflow aún cuando el cociente tenga representación en $\mathbb{F}$. Es más seguro actualizar el término anterior como

$$\frac{x^{n+1}}{(n+1)!} = \frac{x^n}{n!}\frac{x+1}{n+1}$$

3. La tolerancia `Tol` está dada en valor absoluto, lo que no es adecuado fuera del intervalo $[-1, 1]$.

4. En el caso de que el argumento x sea positivo, entonces el número de condición para la suma de la serie es (véase (5.7) en la página 111)

$$\text{cond}(1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots) = 1.$$

   Pero si $x < 0$ la cosa cambia radicalmente pues cada término con exponente impar será negativo y la cancelación catastrófica impide garantizar un error relativo bajo. En este caso lo mejor es valerse de la propiedad $a^{-x} = 1/a^x$.

5. Debieran verificarse los límites de overflow y underflow de la tabla 7.1.

Con las observaciones anteriores, el código debe modificarse. Una mejor versión, que aún adolece de la falta de reducción de rango y otros detalles se muestra en el código 7.2. La mejora cuando $x < 0$ es substancial, lo que es evidente a partir de experimentos sencillos.

Pocas mejoras adicionales son posibles. Puede utilizarse suma compensada en vez de la línea 17 o algún algoritmo más sofisticado de suma, pero no se trata de inexactitud, sino de imprecisión, por lo que se deben buscar mejores soluciones.

Una vez determinado el intervalo de convergencia, se puede dejar fija la cantidad de términos necesarios para alcanzar la precisión necesaria. Entre más pequeño el intervalo menos términos se requerirán.

Listing 7.2: Versión un poco más eficiente de la función exponencial.

```
1  double exp_v2(double x)
2  {
3      int i;
4      double r;
5      double Term=1.0;

7      if ( x<0 )
8          return 1/exp_v1(-x);

10     if ( x>709 )
11         return Huge*Huge;       /* overflow */
12     if ( x<-745 )
13         return 1/(Huge*Huge); /* underflow */
14     if ( fabs(x)<1e-16 )
15         return 1;

17     r = 1;
18     for ( i=1 ; i<MaxTerm ; i++ ) {
19         Term *= x/(double)i;
20         if ( r+Term == r )
21             break;
22         if ( fabs(Term) < Tol )
23             break;
24         r += Term;
25     }
26     return r;
27 }
```

Listing 7.3: Separando la función exponencial en sus partes fraccionaria y entera.

```
1  double exp_v3(double x)
2  {
3      double n, f;

5      /* modf se incluye en la biblioteca de todos
6         los compiladores que respetan el C ANSI */
7      f = modf(x, &n);

9      return Pot(M_E, n)*exp_v2(f);
10 }
```

Las comprobaciones iniciales (overflow, underflow y e^{ε}, líneas 7–15) no se realizarán en los siguientes códigos con el propósito de ahorrar líneas, pero debe suponerse que siempre son necesarias.

7.4. Reducciones básicas de rango

La reducción de rango para la función exponencial son del tipo multiplicativo, ya que no se trata de una función periódica. Sea $x \in \mathbb{R}$ el argumento de la función.

7.4.1. Reducciones sencillas

Esto no es realmente una reducción de rango, sino una técnica *divide y vencerás*.

Una primera idea es separarlo en dos problemas más sencillos: sus partes entera y fraccional. Si x tiene la forma $n.f$ con n entero y $|f| < 1$, entonces

$$e^x = e^{n+f} = e^n \times e^f \tag{7.15}$$

y ahora el problema se separa en dos partes: elevar e a una potencia entera[10] y calcular e^f en el intervalo de convergencia $(-1, 1)$, donde puede utilizarse la función `exp_v2` con cierta confianza.

La mejor manera de separar x en sus partes entera y fraccionaria es utilizar la función `modf` especificada en la norma. Su prototipo especifica que recibe el valor flotante `x`, la dirección del entero `n` y regresa la parte fraccionaria `f`. La nueva versión de función exponencial se muestra en el código 7.3, donde la constante `M_E` se define como $\text{fl}(e^1)$ en `math.h`.

Lo único que falta es una función que calcule x^n de manera más eficiente que `pow`. La idea detrás del código 7.4 es debida a Kahan. Si bien no es óptima, al menos sí lo es asintóticamente. La idea es separar n como la suma de potencias de 2.

Todo entero puede escribirse como la suma de potencias de 2, por lo que si

$$n = 2^{p_1} + 2^{p_2} + \cdots + 2^{p_n}$$

[10] e^n parece un problema sencillo, pero e es un número trascendente y por tanto $\text{fl}(e)$ es inexacto.

Listing 7.4: Función para calcular x^n.

```
1  double Pot(double x, int n)
2  {
3      double r;

5      if ( n==0 )
6          return 1;
7      if ( n<0 )
8          return 1.0/Pot(x, -n);

10     while ( !(n&1) ) {
11         n>>=1;
12         x *= x;
13     }
14     r = x;
15     for (;;) {
16         n>>=1;
17         if ( n==0 )
18             break;
19         x *= x;
20         if ( n&1 )
21             r *= x;
22     }
23     return r;
24 }
```

entonces

$$x^n = x^{2^{p_1}} x^{2^{p_2}} \cdots x^{2^{p_n}}$$

que es lo que hace el código 7.4. Por ejemplo, $7^{11} = 7^8 7^2 7^1$ ya que $11 = 8 + 2 + 1$.

La función `exp_v3` tiene errores de redondeo intrínsecos. La separación con `modf` es exacta, pero el uso de `exp_2` no lo es, tampoco la potencia mediante `Pot` y el último redondeo no controlado es el producto de la línea 9 del código 7.3.

Lo anterior implica que hay que sumar los errores por lo que son varios ulp incorrectos. Sin embargo no deja de ser una versión que depende de la buena aproximación de `exp_v2` en $[-1, 1]$.

Una reducción de rango sencilla es $y = x \ln 2$, pero resulta tímida pues el factor de reducción es apenas .6, por lo que el intervalo de convergencia no cambia gran cosa. Lo que se hace es calcular

$$e^x = 2^y$$

y se cambia el problema a encontrar una buena aproximación de 2^y. La forma de programarla podría ser un simple `return pow(2,x*M_LOG2E);`. Claro, puede aplicarse de nuevo la separación de y en sus partes entera y_n y fraccionaria y_f como en `exp_v3`, pero no deja de ser una idea poco efectiva.

Más tentador resulta calcular

$$e^x = 2^{y_n} 2^{y_f} \tag{7.16}$$

aproximando 2^{y_f} con un polinomio de Taylor de orden fijo evaluado con el método de Horner y usando `fma` de ser posible.

Listing 7.5: La función exponencial con la técnica de Hull. Se han suprimido las comprobaciones de los límites.

```
1  double exp_Hull(double x)
2  {
3      double m, f, k;
4      int e;

6      m = frexp(x, &e); /* Extrae el exponente */
7      k = ldexp(1.0,e); /* Calcula 1*2^e       */
8      y = x/k;

10     return Pot(exp_v2(y), k);
11 }
```

Debe ser claro que trabajar con 2 en vez de e tiene sus ventajas en el caso de los formatos con $\beta = 2$, lo que será más obvio en la sección 7.5. Para los formatos decimales puede usarse el equivalente en base 10, es decir, transformar $e^x = 10^y$ con las operaciones correspondientes.

Esta sí es una reducción de rango, de tipo multiplicativo, con $C = 2$.

Otra idea es determinar el entero k tal que $y = x/k$ puede calcularse con exactitud y $|x/k| < 1$, basta que k sea una potencia de 2 para no perder exactitud. Entonces se puede aproximar e^y en el intervalo de convergencia con el polinomio (7.14) truncado. La reconstrucción se hace con

$$e^x = (e^y)^k \tag{7.17}$$

y es idea de Hull y Abrham[11], presentada en [90] y la califican de "razonablemente eficiente". El valor de k puede calcularse como la menor potencia de 2 mayor que x.

La idea se ejemplifica en el código 7.5, que aún puede optimizarse más si se considera que la división entre k (línea 8) sólo requiere modificar el campo del exponente en la codificación flotante. Las funciones `frexp` y `ldexp` pertenecen a la norma (véase la tabla 4.3). La primera extrae el exponente e y la mantisa m del formato flotante de x y la segunda procede a la inversa, calculando $x = m \times 2^e$.

Incluso se puede aumentar el exponente a $e+1$ y reducir el intervalo de convergencia a la mitad, con lo que la función `exp_v2` utilizará menos términos.

Otra posibilidad es la de Brown y Feldman publicada en [22][12], donde el argumento reducido y queda en el intervalo $[0, \ln 2] \approx [0, .7]$. Entre más pequeño sea el intervalo de convergencia se gana en exactitud. Lo que se hace es calcular $q = \lfloor x/\ln 2 \rfloor$ y luego $y = x - q \times \ln 2$. La reconstrucción es

$$e^x = 2^q \times e^y. \tag{7.18}$$

Como el exponente de e^x es $q + 1$, puede verificarse que no ocurra overflow antes de calcular (7.18) comparando con el exponente máximo e_{max}. En ese caso, se regresa ∞. El valor de 2^q debe calcularse con la función `ldexp`.

[11] Se presenta sólo la idea, pues la versión original utiliza base 10 en una computadora VAX.

[12] En este artículo realmente proponen parámetros numéricos como exponente y fracción independientes del lenguaje y algunos términos genéricos específicos para Fortran 77. Algunos de estos ya los comentaban desde antes Forsythe, Moler e incluso Wilkinson. La versión de función exponencial es sólo un ejemplo de uso de la propuesta.

7.4.2. Reducciones más sofisticadas

En este caso se trata no sólo de reducciones de rango que consiguen un intervalo de convergencia muy pequeño, sino que se apoyan con tablas precalculadas. Los 3 métodos siguientes se publicaron con sólo dos años de diferencia. Debido a la gran cantidad de detalles que acompañan a los algoritmos, solo se presentan de manera general.

En [176], publicado en 1989, Tang reduce x al intervalo $[-\frac{\ln 2}{64}, \frac{\ln 2}{64}] \approx [-.02, .02]$. La idea es encontrar enteros m y j y valores r_1 y r_2 tales que

$$x = (32m + j)(\frac{\ln 2}{32} + (r_1 + r_2))$$

donde $|r_1 + r_2| \leq \frac{ln2}{64}$. Se aproxima $z = e^{r_1+r_2}$ mediante un polinomio minimax y se reconstruye

$$e^x = 2^m(2^{j/32} + 2^{j/32}z).$$

Tang también especifica cómo manejar todas las posibilidades de argumento x, considerando las posibles excepciones y explica cómo optimizar el uso de las tablas exactas.

En 1991, Gal y Bachelis publicaron [69] donde presentan los excelentes resultados del centro científico israelí de IBM. Como el anterior, se basa en el uso de tablas exactas con 64 bits de mantisa de precisión y un polinomio minimax de orden 4.

Primero aplican una técnica de reducción de rango similar a (7.16) y construyen el valor $z = y - i/512 - \epsilon_i$, con $-177 \leq i \leq 177$, por lo que ahora

$$e^x = 2^{y_n} \times f_i \times e^z.$$

y el último factor se aproxima con un polinomio minimax en el intervalo $[-2^{-10}, 2^{-10}] \approx [-.001, .001]$. Las f_i se almacenan en una tabla. Como el polinomio tiene un coeficiente 1, la aproximación es realmente $2^{y_n} \times (f_i + f_i \times pz)$.

Para garantizar la exactitud de la reducción $y = x - y_n \times \ln 2$ se representa $\ln 2 = L_0 + L_1$ como la suma de dos números, donde el sumando mayor L_0 tiene sus últimos 10 bits en cero, para garantizar que el producto $L_0 \times n$ sea exacto para toda $-1024 \leq n \leq 1024$. El cálculo de y queda

A la Dekker.

$$\begin{aligned} n =& INT(x/\ln 2) \\ y =& x - n \times L_0 \end{aligned}$$

y la corrección $d = n \times L_1$ se mantiene a parte. Con lo anterior, la reducción de rango queda

$$e^x = 2^{y_n} \times (f_i + f_i \times (p(z) - d - d \times p(z))).$$

El método de Gal consigue el último bit correctamente redondeado en el 99.8 % del intervalo $[-170, 170]$.

En ese mismo año, Abraham Ziv publicó una novedosa técnica que logra calcular $\text{fl}(e^x) = e^x(1 + \delta)$ con $\delta \leq \mu$. Utiliza dos etapas, esperando no necesitar de la segunda, que es en la que sucesivamente se aumenta la precisión iterativamente hasta lograr un redondeo seguro. La primera etapa debe ser lo más rápida posible.

En la primera etapa utiliza las reducciones de rango sucesivas

$$\begin{aligned} y_1 &= x - n \ln 2 \\ y_2 &= y_1 - u_i^1 \\ y_3 &= y_2 - u_j^2 \end{aligned}$$

donde el entero n se selecciona de tal forma que $-\frac{1}{2}\ln 2 < y_1 < \frac{1}{2}\ln 2$. Usa un total de 6 tablas. La evaluación queda

$$e^x = z + z \times (e^{x^3} - 1)$$

donde $z = 2^n \times y_1' \times y_2'$ y las y son valores de las tablas. La función $e^{x^3} - 1$ se aproxima con un polinomio en un subintervalo de $[-\frac{1}{2}\ln 2, \frac{1}{2}\ln 2]$. Esto concluye la primera etapa.

Si la precisión no es suficiente para redondear con confianza, se pasa a la segunda etapa, que utiliza precisión extra hasta conseguir el redondeo correcto. Normalmente esta etapa se apoya en polinomios de Taylor. El peor caso de la función exponencial tiene 104 dígitos 1 juntos, por lo que se sabe que el proceso tiene fin.

Para ello se requiere disponer de rutinas aritméticas de precisión múltiple donde se puede controlar la precisión.

La técnica de Ziv puede aplicarse cuando son conocidos los peores casoso del TMD para una función en el intervalo de interés.

7.5. Uso básico de tablas

La tabla con valores precalculados para la función depende directamente de la reducción de rango seleccionada pues es del intervalo de convergencia de donde deben seleccionarse, en cuanto a cantidad, precisión y distribución.

Por ejemplo, si se utiliza la reducción $e^x = 2^y$ con $y = x \log e$ y se separa $y = y_n + y_f$ en sus partes entera y fraccionarin, 2^{y_n} no tiene problemas de cálculo al ser potencia entera de dos (se usa `ldexp`).

Para calcular 2^{y_f} se puede usar una tabla que contenga los valores $2^{i/32}$ con i desde 0 hasta 31. Entonces $i = \lfloor 32 \times f \rfloor$ es el índice en la tabla al valor $2^{i/32}$ que corresponde a y_f. Se separa el cálculo como

$$2^{y_f} = 2^{i/32} 2^{y - i/32}$$

y el primer factor está precalculado en la tabla. Consiste de las potencias desde $2^{.00000}$, $2^{.00001}$, hasta $2^{.11111}$, considerando las potencias en base dos. Por ello la forma de calcular i, pues multiplicar por 32 desplaza el punto 5 posiciones a la derecha. Al truncar la parte fraccionaria se obtiene el índice de la tabla.

Esta idea se ejemplifica en el código 7.6. Se han utilizado decimales extra en la tabla para permitir que el compilador haga la conversión a binario con eficiencia. Debe estarse seguro de que el compilador no comete errores al hacer la conversión. Si se desea mayor seguridad, es preferible calcular las representaciones exactas.

Listing 7.6: La función exponencial usando una tabla uniformemente distribuida.

```
1  double exp_v8(double x)
2  {
3      int i;
4      double n, f;
5      static const double Tbl[] = { 1, /* = 2^(-0/32) */
6       1.0218971486541166782, 1.0442737824274138403,
7       1.0671404006768236618, 1.0905077326652576592,
8       1.1143867425958925363, 1.1387886347566916537,
9       1.1637248587775775138, 1.1892071150027210667,
10      1.2152473599804688781, 1.2418578120734840486,
11      1.2690509571917332226, 1.2968395546510096659,
12      1.3252366431597412946, 1.3542555469368927283,
13      1.3839098819638319549, 1.4142135623730950488,
14      1.4451808069770466200, 1.4768261459394993114,
15      1.5092644275934227398, 1.5422108254079408236,
16      1.5759808451078864865, 1.6104903319492543082,
17      1.6457554781539648445, 1.6817928305074290861,
18      1.7187192981224779156, 1.7562521603732994831,
19      1.7947090750031071864, 1.8340080864093424635,
20      1.8741676341102999013, 1.9152065613971472939,
21      1.9571441241754002690  /* = 2^(-31/32) */};

23      if ( x<0 )
24          return 1/exp_v8(-x);

26      f = modf(x*M_LOG2E, &n);
27      i = floor(f*32);
28      f -= i/32.0;

30      return ( ldexp(1.0,(int)n) * Tbl[i] ) * p(f);
31  }
```

La última línea del código 7.6 usa al polinomio p definido como

$$p(f) = 1 + f\ln 2 + \frac{f^2}{2}\ln^2 2 + \frac{f^3}{6}\ln^3 2 + \frac{f^4}{24}\ln^4 2 + \\ + \frac{f^5}{120}\ln^5 2 + \frac{f^6}{720}\ln^6 2 + \frac{f^7}{5040}\ln^7 2 \tag{7.19}$$

que aproxima la función 2^f en el intervalo $[-.03125, .03125]$ con un error máximo de 1.5×10^{-18}.

La constante `M_LOG2E` se define en `math.h`. Si esa tabla de valores se utilizará con otra función elemental, puede dejarse como global en vez de ser un dato local. La razón de declarar `static` al arreglo es mantenerlo siempre en memoria, lo que se puede cambiar dependiendo de las restricciones.

El polinomio (7.19) no es la única posibilidad, también puede utilizarse una aproximación como la de Padé $P_{4,4}$ definida como

$$P_{4,4}(x) = \frac{x^4 + 20x^3 + 180x^2 + 840x + 1680}{x^4 - 20x^3 + 180x^2 - 840x + 1680} \tag{7.20}$$

que equivale a un polinomio de Taylor de grado 8, con un error en el intervalo $[-2^{-5}, 2^{-5}] = [-.03125, .03125]$ de 1.2×10^{-21}.

$P_{3,3}$ se evalúa más rápido que $P_{4,4}$ y con mayor estabilidad.

La aproximación con $P_{3,3}$ se queda corta pues $|P_{3,3}(x) - e^x| > 2\mu$, por lo que no es útil para doble precisión. Si se insiste en usar $P_{3,3}$, habría que usar una tabla con 64 valores, ya que en el intervalo reducido $[-2^{-6}, 2^{-6}] = [-.015625, .015625]$ sí se logra un error máximo de 2.2×10^{-18}.

Cuando las aproximaciones de Padé tienen el mismo exponente en el denominador y numerador, los coeficientes se diferencian solo por el signo de algunos de ellos, por lo que se puede hacer la evaluación de una manera muy óptima.

* * *

A partir de los anteriores ejemplos de aproximación puede verse que hay un compromiso permanente entre la reducción de rango, el cálculo previo de constantes, el tamaño de la tabla y el grado del polinomio o función aproximante seleccionada.

7.6. Probando la rutina

Como ya se ha dicho, probar que una rutina es correcta cuesta más que programarla.

Otra posibilidad sugerida por Cody es el uso de series de Taylor.

Bibliografía

[1] Aberth, O., and Schaefer, M. J. Precise computation using range arithmetic, via c++. *ACM Transactions on Mathematical Software 18*, 4 (1992), 481–491.

[2] Aberth, O., and Schaefer, M. J. Precise matrix eigenvalues using range arithmetic. *SIAM Journal on Matrix Analysis and Applications 14*, 1 (1993), 235–241.

[3] Adams, D. A. A stopping criterion for polynomial root finding. *Communications of the ACM 10*, 10 (octubre 1967), 655–658.

[4] Alefeld, G. E. On the convergence of Halley's method. *American Mathematical Monthly 88*, 7 (septiembre 1981), 530–536.

[5] Alefeld, G. E., Potra, F. A., and Shi, Y. Algorithm 748: Enclosing zeros of continuous functions. *ACM Transactions on Mathematical Software 21*, 3 (septiembre 1995), 327–344.

[6] Allison, C. Where did all my decimals go? *J. Comput. Small Coll. 21*, 3 (febrero 2006), 47–59.

[7] Alt, H. Comparison of arithmetic functions with respect to boolean circuit depth. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1984), ACM Press, pp. 466–470.

[8] Alt, H. Comparing the combinational complexities of arithmetic functions. *Journal of the ACM 35*, 2 (abril 1988), 447–460.

[9] Amat, S., Busquier, S., and Gutiérrez, J. M. Geometric constructions of iterative functions to solve nonlinear equations. *Journal of Computational and Applied Mathematics 157*, 1 (2003), 197–205.

[10] Anderson, I. J. A distillation algorithm for floating point summation. *SIAM Journal on Scientific Computing 20*, 5 (1999), 1797–1806.

[11] Ashenhurst, R. L. Function evaluation in unnormalized arithmetic. *Journal of the ACM 11*, 2 (1964), 168–187.

[12] Ashenhurst, R. L., and Metropolis, N. Unnormalized floating point arithmetic. *Journal of the ACM 6*, 3 (julio 1959), 415–428.

[13] Bailey, D. H. A fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software 21*, 4 (1995), 379–387.

[14] Barrett, G. Formal methods applied to a floating-point number system. *IEEE Trans. Softw. Eng. 15*, 5 (mayo 1989), 611–621.

[15] Black, C. M., Burton, R., and Miller, T. H. The need for an industry standard of accuracy for elementary-function programs. *ACM Transactions on Mathematical Software 10*, 4 (diciembre 1984), 361–366.

[16] Blue, J. L. A portable fortran program to find the euclidean norm of a vector. *ACM Transactions on Mathematical Software 4*, 1 (marzo 1978), 15–23.

[17] Borwein, P., and Erdelyi, T. *Polynomials and Polynomial Inequalities*, vol. 161 of *Graduate Texts in Mathematics*. Springer Verlag, 1995. ISBN: 978-0-387-94509-5.

[18] Brent, R. P. An algorithm with guaranteed convergence for finding a zero of a function. *The Computer Journal 14*, 4 (1971), 422–425.

[19] Brisebarre, N., Defour, D., and Revol, N. A new range-reduction algorithm. *IEEE Trans. Comput. 54*, 3 (marzo 2005), 331–339. Member-Peter Kornerup and Senior Member-Jean-Michel Muller.

[20] Brown, J. H., III, J. W. C., Larrowe, B., and McReynolds, J. R. Prevention of propagation of machine errors in long problems. *Journal of the ACM 3*, 4 (octubre 1956), 348–354.

[21] Brown, W. S. A simple but realistic model of floating-point computation. *ACM Transactions on Mathematical Software 7*, 4 (diciembre 1981), 445–480.

[22] Brown, W. S., and Feldman, S. I. Environment parameters and basic functions for floating-point computation. *ACM Transactions on Mathematical Software 6*, 4 (diciembre 1980), 510–523.

[23] Brown, W. S., and Richman, P. L. The choice of base. *Communications of the ACM 12*, 10 (octubre 1969), 560–561.

[24] Buchholz, W. Fingers or fists? (the choice of decimal or binary representation). *Communications of the ACM 2*, 12 (diciembre 1959), 3–11.

[25] Burden, R. L., and Faires, J. D. *Análisis Numérico*, 7 ed. International Thomson, México, 1999.

[26] Burger, R. G., and Dybvig, R. K. Printing floating-point numbers quickly and accurately. In *SIGPLAN Conference on Programming Language Design and Implementation* (1996), pp. 108–116.

[27] Burkill, J. C., Ed. *Functions of Intervals* (1924), vol. 2-22, London Mathematical Society, Proceedings of the London Mathematical Society.

[28] Bus, J., and Dekker, T. Two efficient algorithms with guaranteed convergence. *ACM Transactions on Mathematical Software 1*, 4 (diciembre 1975), 330–345.

[29] Chapra, S. C., and Canale, R. P. *Métodos Numéricos para Ingenieros*, 4 ed. WCB/McGraw-Hill, Boston, MA, USA, 2003.

[30] Clenshaw, C. W., and Olver, F. W. J. Beyond floating point. *Journal of the ACM 31*, 2 (abril 1984), 319–328.

[31] Clinger, W. D. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1990), ACM Press, pp. 92–101.

[32] Cody, W. J. The FUNPACK package of special function subroutines. *ACM Transactions on Mathematical Software 1*, 1 (marzo 1975), 13–25.

[33] Cody, W. J. Software basics for computational mathematics. *SIGNUM Newsl. 15*, 2 (junio 1980), 18–29.

[34] Cody, W. J. Algorithm 665: Machar: a subroutine to dynamically determined machine parameters. *ACM Transactions on Mathematical Software 14*, 4 (diciembre 1988), 303–311.

[35] Cody, W. J. Algorithm 714; CELEFUNT: a portable test package for complex elementary functions. *ACM Transactions on Mathematical Software 19*, 1 (marzo 1993), 1–21.

[36] Cody, W. J., Coonen, J. T., Gay, D. M., Hanson, K., Hough, D., Kahan, W., Karpinski, R., Palmer, J., Ris, F. N., and Stevenson, D. A proposed radix- and word-length-independent standard for floating-point arithmetic. *SIGNUM Newsl. 20*, 1 (1985), 37–51.

[37] Cody, W. J., and Stoltz, L. The use of Taylor series to test accuracy of function programs. *ACM Transactions on Mathematical Software 17*, 1 (marzo 1991), 55–63.

[38] Cody, W. J., and Waite, W. *Software Manual for the Elementary Functions*. Prentice-Hall series in computational mathematics. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1980.

[39] Committee, M. S. Standard for floating-point arithmetic. Borrador, 2007.

[40] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 2 ed. MIT Press (McGraw Hill), Cambridge MA, 2001.

[41] Cornea-Hasegan, M. Proving IEEE correctness of iterative floating-point square root, divide, and remainder algorithms. Tech. rep., Intel Corporation, 1998. Intel Technology Journal, Q2.

[42] Cornea-Hasegan, M. IA-64 floating-point operations and the IEEE standard for binary floating-point arithmetic. Tech. rep., Intel Corporation, 1999. Intel Technology Journal, Q4.

[43] Corporate IFIP Working Group 2, N. S. Comments on version 3.1 of draft iso/iec 10967: 1991 language compatible arithmetic. *SIGNUM Newsl. 27*, 1 (1992), 2–3.

[44] Cowlishaw, M. F. Densely packed decimal encoding. *IEE Computers and Digital Techniques 149*, 3 (mayo 2002), 102–104.

[45] Cowlishaw, M. F. Decimal floating-point: Algorism for computers. In *ARITH '03: Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)* (Washington, DC, USA, 2003), IEEE Computer Society, p. 104. ISBN:0-7695-1894-X.

[46] Daumas, M., Mazenc, C., Merrheim, X., and Muller, J.-M. Modular range reduction. *Journal of Universal Computer Science 1*, 3 (marzo 1995), 162–175.

[47] Defour, D., Hanrot, G., Lefevre, V., Muller, J.-M., Revol, N., and Zimmermann, P. Proposal for a standardization of mathematical function implementation in floating-point arithmetic. *Numerical Algorithms 37* (2004), 367–375.

[48] Dekker, T. J. A floating-point technique for extending the available precision. *Numerische Mathematik 18*, 3 (junio 1971), 224–242.

[49] Demmel, J., and Hida, Y. Accurate floating point summation. Tech. rep., University of California, Berkeley, CA 94720, mayo 2002. UCB//CSD-02-1180.

[50] Demmel, J., and Hida, Y. Accurate and efficient floating point summation. *SIAM Journal on Scientific Computing 25* (2003), 1214–1248.

[51] Demmel, J., and Hida, Y. Fast and accurate floating point summation with application to computational geometry. *Numerical Algorithms 37* (2004), 101–112.

[52] Dunham, C. B. Choice of basis for chebyshev approximation. *ACM Transactions on Mathematical Software 8*, 1 (marzo 1982), 21–25.

[53] Dunham, C. B. Provably monotone approximations. *SIGNUM Newsl. 22*, 2 (abril 1987), 6–11.

[54] Dunham, C. B. Feasibility of "perfect function evaluation. *SIGNUM Newsl. 25*, 4 (junio 1990), 25–26.

[55] Eggers, T. W., Leonard, J. S., and Payne, M. H. Handling of floating point exceptions. In *Proceedings of the SIGNUM Conference on the Programming Environment for Development of Numerical Software* (New York, NY, USA, 1979), ACM Press, pp. 100–108.

[56] Einarsson, B., Ed. *Accuracy and Reliability in Scientific Computing (Software, Environments, Tools) (Software, Environments, Tools)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, enero 2005.

[57] Eisinberg, A., and Fedele, G. Accurate floating-point summation: a new approach. *Applied Mathematics and Computation 189* (2007), 410–424.

[58] Fateman, R. J. High-level language implications of the proposed IEEE floating-point standard. *ACM Trans. Program. Lang. Syst. 4*, 2 (abril 1982), 239–257.

[59] Feigenbaum, L. Taylor and the method of increments. *Arch. Hist. Exact Sci. 34* (1985), 1–140.

[60] Fishman, G. *Monte Carlo: Concepts, algorithms, and applications*, 4 ed. Springer Series in Operations Research. Springer, 2003.

[61] Ford, B. Parameterization of the environment for transportable numerical software. *ACM Transactions on Mathematical Software 4*, 2 (junio 1978), 100–103.

[62] Ford, J. A. Improved algorithms of Illinois-type for the numerical solution of nonlinear equations. Technical report csm-257, University of Essex, 1995.

[63] Fousse, L., Hanrot, G., 'e, V. L., Plíssie, P., and Zimmermann, P. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software 33*, 2 (junio 2007), 1.

[64] Fraser, W. A survey of methods of computing minimax and near-minimax polynomial approximations for functions of a single independent variable. *Journal of the ACM 12*, 3 (julio 1965), 295–314.

[65] Frontini, M., and Sormani, E. Modified Newton's method with third-order convergence and multiple roots. *Journal of Computational and Applied Mathematics 156*, 2 (julio 2003), 345–354.

[66] Frontini, M., and Sormani, E. Some variant of Newton's method with third-order convergence. *Applied Mathematics Letters 140*, 2-3 (agosto 2003), 419–426.

[67] G. Alefeld, J. H. *Introduction to Interval Computations*. Academic Press, New York, 1974.

[68] G. E. Forsythe, M. A. M. Y. C. B. M. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, NJ, 1977.

[69] Gal, S., and Bachelis, B. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software 17*, 1 (marzo 1991), 26–45.

[70] Gay, D. M. Correctly rounded binary-decimal and decimal-binary conversions. Tech. Rep. 90-10, AT&T Bell Laboratories, Murray Hill, NJ, USA, noviembre 1990.

[71] Gentleman, W. M., and Marovich, S. B. More on algorithms that reveal properties of floating point arithmetic units. *Communications of the ACM 17*, 5 (mayo 1974), 276–277.

[72] Gerald, C. F., and Wheatley, P. O. *Análisis numérico con aplicaciones*. Pearson Educación, 2000.

[73] Gibb, A. Algorithm 61: procedures for range arithmetic. *Communications of the ACM 4*, 7 (1961), 319–320.

[74] Glaser, A., Liu, X., and Rokhlin, V. A fast algorithm for the calculation of the roots of special functions. *SIAM Journal on Scientific Computing 29*, 4 (2007), 1420–1438.

[75] Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys 23*, 1 (Marzo 1991), 5–48.

[76] Goldberg, D. The design of floating-point data types. *ACM Lett. Program. Lang. Syst. 1*, 2 (junio 1992), 138–151.

[77] Goldberg, I. B. 27 bits are not enough for 8-digit accuracy. *Communications of the ACM 10*, 2 (febrero 1967), 105–106.

[78] Goldstein, M. Significance arithmetic on a digital computer. *Communications of the ACM 6*, 3 (1963), 111–117.

[79] Gregory, R. T., and Raney, J. L. Floating-point arithmetic with 84-bit numbers. *Communications of the ACM 7*, 1 (enero 1964), 10–13.

[80] Gutiérrez, J. M., and Hernández, M. A. Newton's method under weak Kantorovich conditions. *Journal of Numerical Analysis 20* (2000), 521–532.

[81] Gutiérrez, J. M., and Hernández, M. A. An acceleration of newton's method: Super-halley method. *Applied Mathematics and Computation 117*, 2-3 (enero 2001), 223–239.

[82] Guy L. Steele Jr., J. L. W. How to print floating-point numbers accurately. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1990), ACM Press, pp. 112–126.

[83] Hatton, L. Embedded system paranoia: a tool for testing embedded system arithmetic. http://www.leshatton.org/Documents/esp.pdf, febrero 2004.

[84] Hauser, J. R. Handling floating-point exceptions in numeric programs. *ACM Trans. Program. Lang. Syst. 18*, 2 (Marzo 1996), 139–174.

[85] Henrici, P. A subroutine for computations with rational numbers. *Journal of the ACM 3*, 1 (enero 1956), 6–9.

[86] Hickey, T., Ju, Q., and Emden, M. H. V. Interval arithmetic: From principles to implementation. *Journal of the ACM 48*, 5 (2001), 1038–1068.

[87] Higham, N. J. The accuraccy of floating point summation. *SIAM Journal on Scientific Computing 14*, 4 (1993), 783–799.

[88] Higham, N. J. *Accuracy and Stability of Numerical Algorithms*, 1 ed. ed. SIAM, 1996.

[89] Householder, A. S. Generation of errors in digital computation. *Bulletin of American Mathematical Society 60* (1954), 234–247.

[90] Hull, T. E., and Abrham, A. Variable precision exponential function. *ACM Transactions on Mathematical Software 12*, 2 (junio 1986), 79–91.

[91] Hull, T. E., Cohen, M. S., Sawshuk, J. T. M., and Wortman, D. B. Exception handling in scientific computing. *ACM Transactions on Mathematical Software 14*, 3 (septiembre 1988), 201–217.

[92] Hull, T. E., Fairgrieve, T. F., and Tang, P. T. P. Implementing the complex arcsine and arccosine functions using exception handling. *ACM Transactions on Mathematical Software 23*, 3 (septiembre 1997), 299–335.

[93] Ikebe, Y. Note on triple-precision floating-point arithmetic with 132-bit numbers. *Communications of the ACM 8*, 3 (marzo 1965), 175–177.

[94] Jacobi, C., Weber, K., Paruthi, V., and Baumgartner, J. Automatic formal verification of fused-multiply-add fpus. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe* (Washington, DC, USA, 2005), vol. 2, IEEE Computer Society, pp. 1298–1303.

[95] John W. Carr, I. Error analysis in floating point arithmetic. *Communications of the ACM 2*, 5 (mayo 1959), 10–15.

[96] Jones, C. B. A significance rule for multiple-precision arithmetic. *ACM Transactions on Mathematical Software 10*, 1 (1984), 97–107.

[97] K. Hillesland, A. L., Ed. *GPU floating-point paranoia* (2004), vol. p. C-8 of *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors*.

[98] Kahan, W. Further remarks on reducing truncation errors. *Communications of the ACM 8*, 1 (enero 1963), 40.

[99] Kahan, W. In memoriam: Hirondo kuki: Apr. 25, 1925 - dec. 28, 1971. *SIGNUM Newsl. 7*, 1 (abril 1972), 8–10.

[100] Kahan, W. Why we need a floating-point arithmetic standard. Tech. rep., Univ. of California at Berkeley, Univ. of California at Berkeley, Marzo 5 1981.

[101] Kahan, W. Minimizing $q \times m - n$. Comentario inicial del programa nearpi.c, en http://www.cs.berkeley.edu/ wkahan/testpi/nearpi.c, 1983.

[102] Kahan, W. Branch cuts for complex elementary functions. Notas de curso, 1987.

[103] Kahan, W. Analysis and refutation of the LCAS. *SIGNUM Newsl. 26*, 3 (1991), 2–15.

[104] Kahan, W. The improbability of probabilistic error analyses for numerical computations. Tech. rep., Univ. of Calif. Berkeley, 1011 Evans Hall, Feb. 1996. UCB Statistics Colloquium.

[105] Kahan, W. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. Tech. rep., University of California at Berkeley, Octubre 1 1997.

[106] Kahan, W. Matlabs loss is nobodys gain. Tech. rep., Univ. of Calif. Berkeley, julio 2002. Creado en 1998.

[107] Kahan, W. An ordinary differential equation in interval-arithmetic. Tech. rep., Univ. of Calif. Berkeley, marzo 2002. Notas de curso.

[108] Kahan, W. Lecture notes on real root-finding. Lecture notes, University of California at Berkeley, junio 2004.

[109] Kahan, W. A demonstration of presubstitution. Lecture notes, Univ. of Calif. Berkeley, julio 2005.

[110] Kahan, W., Coonen, J., Palmer, J., Pittman, T., and Stevenson, D. A proposed standard for binary floating point arthmetic. *SIGNUM Newsl. 14*, si-2 (1979), 4–12.

[111] Kahan, W., and Darcy, J. How Java's floating-point hurts everyone everywhere. Pgina web, marzo 1998.

[112] Kaivola, R., and Narasimhan, N. Formal verification of the Pentium 4 floating-point multiplier. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe* (Washington, DC, USA, 2002), IEEE Computer Society, p. 20.

[113] Kaneko, T., and Liu, B. On local roundoff errors in floating-point arithmetic. *Journal of the ACM 20*, 3 (1973), 391–398.

[114] Kearfott, R. B., Dawande, M., Du, K., and Hu, C. Algorithm 737: Intlib—a portable Fortran 77 interval standard-function library. *ACM Transactions on Mathematical Software 20*, 4 (diciembre 1994), 447–459.

[115] Kern, C., and Greenstreet, M. R. Formal verification in hardware design: a survey. *ACM Trans. Des. Autom. Electron. Syst. 4*, 2 (abril 1999), 123–193.

[116] Kincaid, D., and Cheney, W. *Análisis Numérico*. Addison Wesley Iberoamericana, 1994.

[117] Knuth, D. E. *The art of computer programming: seminumerical algorithms*, 3 ed., vol. 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, noviembre 1997.

[118] Koren, I. *Computer Arithmetic Algorithms*, 2 ed. A K Peters, Ltd., 2002. Primera edición de Prentice-Hall.

[119] Kuki, H., and Cody, W. J. A statistical study of the accuracy of floating point number systems. *Communications of the ACM 16*, 4 (abril 1973), 223–230.

[120] Lagrange, S., Delanoue, N., and Jaulin, L. On sufficient conditions of the injectivity: Development of a numerical test algorithm via interval analysis. *Reliable computing 13*, 5 (octubre 2007), 409–421. publicado en internet.

[121] Lake, G. T. Hardware conversion of decimal and binary numbers. *Communications of the ACM 5*, 9 (septiembre 1962), 468–469.

[122] Le, D. An efficient derivative-free method for solving nonlinear equations. *ACM Transactions on Mathematical Software 11*, 3 (1985), 250–262.

[123] Lefèvre, V., and Muller, J.-M. Worst cases for correct rounding of the elementary functions in double precision. In *Proceedings of the 15th Symposium on Computer Arithmetic* (Vail, Colorado, 2001), N. Burgess and L. Ciminiera, Eds., pp. 111–118.

[124] Lerch, M., Tischler, G., Gudenberg, J. W. V., Hofschuster, W., and Krämer, W. FILIB++, a fast interval library supporting containment computations. *ACM Transactions on Mathematical Software 32*, 2 (junio 2006), 299–324.

[125] Li, X. S., Demmel, J. W., Bailey, D. H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S. Y., Kapur, A., Martin, M. C., Thompson, B. J., Tung, T., and Yoo, D. J. Design, implementation and testing of extended and mixed precision blas. *ACM Transactions on Mathematical Software 28*, 2 (junio 2002), 152–205.

[126] Linnainmaa, S. Software for doubled-precision floating-point computations. *ACM Transactions on Mathematical Software 7*, 3 (septiembre 1981), 272–283.

[127] Linz, P. Accurate floating-point summation. *Communications of the ACM 13*, 6 (junio 1970), 361–362.

[128] Loan, G. H. G. Y. C. F. V. *Matrix Computations*, segunda edición ed. Johns Hopkins Univ. Press, 1989.

[129] Lynch, W. C. On a wired-in binary-to-decimal conversion scheme. *Communications of the ACM 5*, 3 (marzo 1962), 159.

[130] Malcolm, M. A. On accurate floating-point summation. *Communications of the ACM 14*, 11 (noviembre 1971), 731–736.

[131] Malcolm, M. A. Algorithms to reveal properties of floating-point arithmetic. *Communications of the ACM 15*, 11 (noviembre 1972), 949–951.

[132] Matula, D. W. In-and-out conversions. *Communications of the ACM 11*, 1 (enero 1968), 47–50.

[133] Matula, D. W., and McFearin, L. D. A formal model and efficient traversal algorithm for generating testbenches for verification of ieee standard floating point division. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe* (3001 Leuven, Belgium, Belgium, 2006), European Design and Automation Association, pp. 1134–1138.

[134] McNamee, J. M. A comparison of methods for accurate summation. *ACM SIGSAM Bulletin 38*, 1 (marzo 2004), 1–1.

[135] Mikov, A. I. Largescale addition of machine real numbers: accuracy estimates. *Theor. Comput. Sci. 162* (1996), 151–170.

[136] Miller, W. Software for roundoff analysis. *ACM Transactions on Mathematical Software 1*, 2 (junio 1975), 108–128.

[137] Miller, W., and Spooner, D. Software for roundoff analysis, ii. *ACM Transactions on Mathematical Software 4*, 4 (diciembre 1978), 369–387.

[138] Moler, C., and Morrison, D. Replacing square roots by pythagorean sums. *Journal or Research and Development 28* (1983), 577–582.

[139] Müller, D. E. A method for solving algebraic equations using an automatic computer. *Math. Tables Aids Comput. 10* (1956), 208–215.

[140] Muller, J.-M. *Elementary functions*, 2 ed. Birkhauser, 2006.

[141] Neher, M. Complex standard functions and their implementation in the CoStly library. *ACM Transactions on Mathematical Software 33*, 1 (marzo 2007), 2.

[142] Neta, B. A sixth-order family of methods for nonlinear equations. *International Journal of Computer Mathematics 7*, 2 (1979), 157–161.

[143] Nievergelt, Y. Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Transactions on Mathematical Software 29*, 1 (marzo 2003), 27–48.

[144] Ogita, T., Rump, S., and Oishin, S. Accurate sum and dot product. *SIAM Journal on Scientific Computing 26*, 6 (2005), 1955–1988.

[145] O'Leary, J., Zhao, X., Gerth, R., , and Seger, C.-J. H. Formally verifying IEEE compliance of floating point hardware. *Intel Technology Journal 3*, 1 (1999), 10.

[146] Overton, M. *Numerical Computing with IEEE Floating Point Arithmetic.* SIAM, 2001.

[147] Overton, M. *Cómputo numérico con aritmética de punto flotante IEEE.* No. 19 in Aportaciones matemáticas. SIAM - SMM, 2002. Traducción de Alejandro Casares.

[148] Paul, G., and Wilson, M. W. Should the elementary function library be incorporated into computer instruction sets? *ACM Transactions on Mathematical Software 2*, 2 (junio 1976), 132–142.

[149] Payne, M., and Bhandarkar, D. VAX floating point: A solid foundation for numerical computation. *ACM SIGARCH Computer Architecture News 8*, 4 (junio 1980), 22–33.

[150] Payne, M., and Strecker, W. Draft proposal for a binary normalized floating point standard. *SIGNUM Newsl. 14*, 2 (octubre 1979), 24–28.

[151] Payne, M. H., and Hanek, R. N. Radian reduction for trigonometric functions. *SIGNUM Newsl. 18*, 1 (enero 1983), 19–24.

[152] Pichat, M. Correction d'une somme en arithmetique a virgule flottante. *Numerische Mathematik 19* (1972), 400–406.

[153] Plagianakos, V. P., Nousis, N. K., and Vrahatis, M. N. Locating and computing in parallel all the simple roots of special functions using pvm. *Journal of Computational and Applied Mathematics 133*, 1-2 (2002), 545–554.

[154] Popova, E. D. On a formally correct implementation of IEEE computer arithmetic. *Journal of Universal Computer Science 1*, 7 (julio 1995), 560–569.

[155] Potra, F. A. The Kantorovich theorem and interior point methods. *Math. Program. 102*, 1 (febrero 2005), 47–70.

[156] Potra, F. A., and Wright, S. J. Interior-point methods. *Journal of Computational and Applied Mathematics 124*, 1-2 (2000), 281–302.

[157] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 1992.

[158] Priest, D. M. *On Properties of Floating-Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California, Berkeley, CA, 1992.

[159] Priest, D. M. Efficient scaling for complex division. *ACM Transactions on Mathematical Software 30*, 4 (diciembre 2004), 389–401.

[160] Redish, and Ward. Environment enquiries. *SIGNUM Newsl. 6*, 1 (enero 1971), 10–15.

[161] Reid, J. Functions for manipulating floating-point numbers. *SIGNUM Newsl. 14*, 4 (diciembre 1979), 11–13.

[162] Robertazzi, T. G., and Schwartz, S. C. Best ordering for floating-point addition. *ACM Transactions on Mathematical Software 14*, 1 (marzo 1988), 101–110.

[163] Rump, S. M., Ogita, T., and Oishi, S. Acurate floating-point summation. Tech. rep., Hamburg University of Technology, noviembre 2005.

[164] Rump, S. M., Ogita, T., and Oishi, S. Accurate floating-point summation part i: faithful rounding. Enviado para publicación en SISC, 2005, revisado en abril de 2007, 2007.

[165] Rump, S. M., Ogita, T., and Oishi, S. Accurate floating-point summation part ii: sign, k-fold faithful, and rounding to nearest. Enviado para publicación en SISC, 2005, revisado en abril de 2007, 2007.

[166] Russinoff, D. M. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Form. Methods Syst. Des. 14*, 1 (enero 1999), 75–125.

[167] Sarafyan, D. A new method of computation of square roots without using division. *Communications of the ACM 2*, 11 (noviembre 1959), 23–24.

[168] Scavo, T. R., and Thoo, J. B. On the geometry of Halley's method. *The American Mathematical Monthly 102*, 5 (mayo 1995), 417–426.

[169] Sharma, J. R., and Goyal, R. K. Fourth-order derivative-free methods for solving non-linear equations. *International Journal of Computer Mathematics 83*, 1 (enero 2006), 101–106.

[170] Siegel, S. F., Mironova, A., Avrunin, G. S., and Clarke, L. A. Using model checking with symbolic execution to verify parallel numerical programs. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis* (New York, NY, USA, 2006), ACM Press, pp. 157–168.

[171] Smith, D. M. Algorithm 786: Multiple-precision complex arithmetic and functions. *ACM Transactions on Mathematical Software 24*, 4 (diciembre 1998), 359–367.

[172] Smith, R. L. Algorithm 116: Complex division. *Communications of the ACM 5*, 8 (agosto 1962), 435.

[173] Stewart, G. W. A note on complex division. *ACM Transactions on Mathematical Software 11*, 3 (septiembre 1985), 238–241.

[174] Stewart, G. W. *Errors in variables for numerical analysts*. Recent Advances in Total Least Squares Techniques and Errors-in-Variables Modeling. SIAM, 1997.

[175] Sylvain Collange, J'eremie Detrey, F. d. D. Floating point or LNS: Choosing the right arithmetic on an application basis. *Proceedings of the 9th EUROMICRO Conference on Digital System Design 1* (2006), 10–16.

[176] Tang, P.-T. P. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software 15*, 2 (junio 1989), 144–157.

[177] Tang, P.-T. P. Accurate and efficient testing of the exponential and logarithm functions. *ACM Transactions on Mathematical Software 16*, 3 (septiembre 1990), 185–200.

[178] Tien Chi-Chen, I. T. H. Storage-efficient representation of decimal data. *Communications of the ACM 18*, 1 (enero 1975), 49–52.

[179] Traub, J. F. *Iterative Methods for the Solution of Equations*. Prentice-Hall, Englewood Cliffs, N. J., 1964.

[180] Verdonk, B., Cuyt, A., and Verschaeren, D. A precision- and range-independent tool for testing floating-point arithmetic i: basic operations, square root, and remainder. *ACM Transactions on Mathematical Software 27*, 1 (marzo 2001), 92–118.

[181] Verdonk, B., Cuyt, A., and Verschaeren, D. A precision- and range-independent tool for testing floating-point arithmetic ii: conversions. *ACM Transactions on Mathematical Software 27*, 1 (marzo 2001), 119–140.

[182] Verschaeren, D., Cuyt, A., and Verdonk, B. On the need for predictable floating-point arithmetic in the programming languages fortran 90 and c/c++. *SIGPLAN Not. 32*, 3 (marzo 1997), 57–64.

[183] von Neumann, and Goldstine. Numerical inverting of matrices of high order. *Bulletin of American Mathematical Society 53* (1947), 1021–1099.

[184] von Neumann, and Goldstine. Numerical inverting of matrices of high order ii. *Proceedings of American Mathematical Society 2* (1951), 188–202.

[185] Wadey, W. G. Floating-point arithmetics. *Journal of the ACM 7*, 2 (abril 1960), 129–139. Remington Rand UNIVAC.

[186] Weerakoon, S., and Femando, T. A variant of newton's method with accelerated third-order convergence. *Applied Mathematics Letters 13*, 8 (noviembre 2000), 87–93.

[187] Wilkinson, J. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1964.

[188] Wilkinson, J. H. *The algebraic eigenvalue problem*. Oxford University Press, Inc., New York, NY, USA, 1988.

[189] Wolfe, J. M. Reducing truncation errors by programming. *Communications of the ACM 7*, 6 (junio 1964), 355–356.

[190] Zhengda, H. A note on the Kantorovich theorem for Newton iteration. *Journal of Computational and Applied Mathematics 47*, 2 (1993), 211–217.

[191] Zielke, G., and Drygalla, V. Genaue lösung linearer gleichungssysteme. *GAMM Mitt. Ges. Angew. Math. Mech. 26* (2003), 7–107.

[192] Ziv, A. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software 17*, 3 (septiembre 1991), 410–423.

Índice alfabético