PRACTICA 2a

Números de punto flotante y errores de redondeo.

Sólo hay 10 tipos de personas:
las que saben binario y las que no.

Sistemas numéricos posicionales. Cotidianamente para representar los números utilizamos un *sistema posicional* de *base* 10: el *sistema decimal*. En este sistema los números son representados usando diez diferentes caracteres, llamados *dígitos decimales*, a saber, $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. La magnitud con la que un dígito a_i contribuye al valor del número depende de su posición en el número de manera tal que la representación decimal

$$(-1)^s(a_n a_{n-1} \cdots a_1 a_0 . a_{-1} a_{-2} \cdots)$$

corresponde al número

$$(-1)^s(a_n 10^n + a_{n-1}10^{n-1} + \cdots + a_1 10^1 + a_0 10^0 + \\ + a_{-1}10^{-1} + a_{-2}10^{-2} + \ldots),$$

donde s depende del signo del número ($s = 0$ si el número es positivo y $s = 1$ si es negativo). De manera análoga se puede concebir otros sistemas posicionales con una base distinta de 10. En principio, cualquier número natural $\beta \geq 2$ puede ser utilizado como base. Entonces, fijada una base, todo número real admite una *representación posicional* en la base β de la forma

$$(-1)^s(a_n \beta^n + a_{n-1}\beta^{n-1} + \cdots + a_1 \beta^1 + a_0 \beta^0 + \\ + a_{-1}\beta^{-1} + a_{-2}\beta^{-2} + \ldots),$$

donde los coeficientes a_i son los "dígitos" en el sistema con base β, esto es, enteros positivos tales que $0 \leq a_i \leq \beta - 1$. Los coeficientes $a_{i\geq 0}$ se consideran como los dígitos de la *parte entera*, en tanto que los $a_{i<0}$, son los dígitos de la *parte fraccionaria*. Si, como en el caso decimal, utilizamos un punto para separar tales partes, el número es representado en la base β como

$$(-1)^s(a_n a_{n-1} \cdots a_1 a_0 . a_{-1} a_{-2} \cdots)_\beta,$$

donde hemos utilizado el subíndice β para evitar cualquier ambigüedad con la base escogida.

☞ Aunque cualquier número natural $\beta \geq 2$ define un sistema posicional, en el ámbito computacional sólo son de interés los sistemas *decimal* ($\beta = 10$), *binario* ($\beta = 2$), *octal* ($\beta = 8$) y *hexadecimal* ($\beta = 16$). El sistema binario consta sólo de los dígitos 0 y 1, llamados *bits* (del inglés *binary digits*). Por su parte, el sistema octal usa dígitos del 0 al 7, en tanto que el sistema hexadecimal usa los dígitos del 0 al 9 y las letras A, B, C, D, E, F[1].

Ejercicio 1. Mostrar que $(13.25)_{10} = (1101.01)_2 = (15.2)_8 = (\mathrm{D}.4)_{16}$.

[1] Para sistemas con base $\beta > 10$ es usual reemplazar los dígitos $10, 11, \ldots, \beta - 1$ por las letras A, B, C, ...

Ejercicio 2. Determinar la representación binaria de los siguientes números

a) 29, b) 0.625, c) 0.1, d) 5.75.

Representación de punto flotante. Para la representación de números reales sobre un amplio rango de valores con sólo unos pocos dígitos se utiliza la *notación científica*. Así $976\,000\,000\,000\,000$ se representa como 9.76×10^{14} y 0.0000000000000976 como 9.76×10^{-14}. En esta notación el punto decimal se mueve dinámicamente a una posición conveniente y se utiliza el exponente de 10 para registrar la posición del punto decimal. En particular, todo número real no nulo puede ser escrito en forma única en la notación científica *normalizada*

$$(-1)^s\, 0.a_1 a_2 a_3 \cdots a_t a_{t+1} a_{t+2} \cdots \times 10^e,$$

siendo el dígito $a_1 \neq 0$. De manera análoga, todo número real no nulo puede representarse en forma única, respecto la base β, en la *forma de punto flotante normalizada*:

$$(-1)^s\, 0.a_1 a_2 a_3 \cdots a_t a_{t+1} a_{t+2} \cdots \times \beta^e,$$

donde los "dígitos" a_i respecto de la base β son enteros positivos tales que $1 \leq a_1 \leq \beta - 1$, $0 \leq a_i \leq \beta - 1$ para $i = 1, 2, \ldots$ y constituyen la parte fraccional o *mantisa* del número, en tanto que e, el cual es un número entero llamado el *exponente*, indica la posición del punto correspondiente a la base β. Si m es la fracción *decimal* correspondiente a $(0.a_1 a_2 a_3 \cdots)_\beta$ entonces el número representado corresponde al número decimal

$$(-1)^s\, m \cdot \beta^e, \qquad \text{siendo } \beta^{-1} \leq m < 1.$$

En todo dispositivo de cálculo, como una computadora o calculadora, el número de dígitos posibles para representar la mantisa es *finito*, digamos t dígitos en la base β, y el exponente puede variar sólo dentro de un rango finito $L \leq e \leq U$ (con $L < 0$ y $U > 0$). Esto implica que sólo un conjunto *finito* de números reales pueden ser representados, los cuales tienen la forma

$$(-1)^s\, 0.a_1 a_2 a_3 \cdots a_t \times \beta^e.$$

Tales números se denominan *números de punto flotante* con t dígitos (o de precisión t) en la base β y rango (L, U). Al conjunto de los mismos lo denotamos por $\mathbb{F}(\beta, t, L, U)$.

☞ El número de elementos del conjunto $\mathbb{F}$, esto es, la cantidad de números de puntos flotantes de $\mathbb{F}$, es

$$2(\beta - 1)\beta^{t-1}(U - L + 1).$$

☞ Debido a la normalización el cero *no* puede ser representado como un número de punto flotante y por lo tanto está excluido del conjunto $\mathbb{F}$.

☞ Si $x \in \mathbb{F}$ entonces su opuesto $-x \in \mathbb{F}$.

☞ El conjunto $\mathbb{F}$ está acotado tanto superior como inferiormente:

$$x_{\text{mín}} = \beta^{L-1} \leq |x| \leq x_{\text{máx}} = \beta^U(1-\beta^{-t}),$$

donde $x_{\text{mín}}$ y $x_{\text{máx}}$ son el menor y mayor número de punto flotante positivo representable, respectivamente.

☞ Se sigue de lo anterior que en la recta de los números reales hay cinco regiones excluidas para los números de $\mathbb{F}$:

- Los números negativos menores que $-x_{\text{máx}}$, región denominada *desbordamiento* (*overflow*) *negativo*.
- Los números negativos mayores que $-x_{\text{mín}}$, denominada *desbordamiento a cero* (*underflow*) *negativo*.
- El cero.
- Los números positivos menores que $x_{\text{mín}}$, denominada *desbordamiento a cero* (*underflow*) *positivo*.
- Los números positivos mayores que $x_{\text{máx}}$, denominada *desbordamiento* (*overflow*) *positivo*.

☞ Los números de punto flotante no están igualmente espaciados sobre la recta real, sino que están más próximos cerca del origen y más separados a medida que nos alejamos de él.

☞ Una cantidad de gran importancia es el denominado *epsilon de la máquina* $\epsilon_M = \beta^{1-t}$ el cual representa la distancia entre el número 1 y el número de punto flotante siguiente más próximo.

Con el fin de evitar la proliferación de diversos sistemas de puntos flotantes incompatibles entre sí a fines de la década de 1980 se desarrolló la norma o estandar IEEE754/IEC559[2] el cual es implementado en todas las computadoras actuales. Esta norma define dos formatos para la implementación de números de punto flotante en la computadora:

- *precisión simple*: $\mathbb{F}(2, 24, -125, 128)$,
- *precisión doble*: $\mathbb{F}(2, 53, -1021, 1024)$.

Ejercicio 3. Considere el conjunto de números de punto flotante $\mathbb{F}(2, 3, -1, 2)$.

a) Determinar $x_{\text{mín}}$, $x_{\text{máx}}$, ϵ_M y el número de elementos de $\mathbb{F}$.
b) Determinar los números de punto flotante positivos del conjunto $\mathbb{F}$.
c) Graficar sobre la recta real los números de puntos flotantes determinados en el punto anterior.

Ejercicio 4. Determinar los valores de $x_{\text{mín}}$, $x_{\text{máx}}$, ϵ_M para la representaciones de precisión simple y doble de la norma IEEE754.

Ejercicio 5. Determinar la representación de punto flotante de simple precisión de

$$\text{a) } 5.75, \qquad \text{b) } -118.625.$$

Implementación en Fortran de los números de punto flotante de simple y doble precisión. Todos los compiladores Fortran admiten, al menos, dos *clases* para los tipos de datos reales: el primero, simple precisión y, el segundo tipo, doble precisión tal como se especifica en la norma IEEE754/IEC559. Para declarar el tipo de clase de una variable real se utiliza la siguiente sintaxis:

```
REAL(KIND=número de clase) :: variable
```

donde el *número de clase* es un número entero que identifica la clase de real a utilizar. Este número, para el compilador `gfortran`, es 4 en el caso de simple precisión y 8 para doble precisión. Si no se especifica la clase, entonces se utiliza la clase por omisión, la cual es la simple precisión.

☞ Dado que *el número de clase es dependiente del compilador* la mejor manera de especificar la clase de los tipos reales de manera que sea *independiente del compilador y procesador utilizado* consiste en seleccionar el número de clase a través de las constantes con nombres `REAL32` o `REAL64`, para simple y doble precisión, respectivamente, definidas por el *módulo intrínseco* `ISO_FORTRAN_ENV`. Esto permite asegurar la portabilidad del programa entre distintas implementaciones. Para utilizarlo, invocamos, al comienzo del programa, el módulo con la sentencia `USE` e importamos las constantes:

```
USE iso_fortran_env, ONLY: SP=>REAL32, &
                           DP=>REAL64
...
REAL(SP) :: variables
REAL(DP) :: variables
```

Aquí, por comodidad, hemos establecidos los *alias* `SP` y `DP` (por *simple precision* y *double precision*) para `REAL32` y `REAL64`, respectivamente.

☞ Constantes reales en el código son declaradas de una dada clase agregando a las mismas el guión bajo seguida del número de clase. Por ejemplo:

```
34.0        ! Real de clase por omisión
34.0_SP     ! Real de clase SP
124.5678_DP ! Real de clase DP
```

Es importante comprender que, por ejemplo, `0.1_SP` y `0.1_DP` son números de punto flotante *distintos*, siendo el último guardado internamente con un número mayor de dígitos binarios.

Una manera simple y eficiente de escribir un programa que pueda ser compilado con variables reales ya sea de una clase u otra según se requiera, consiste en utilizar las constantes con nombres `REAL32` o `REAL64`, del mencionado módulo `ISO_FORTRAN_ENV`, para definir la precisión de los tipos reales y luego en el programa invocarlo especificando el tipo de clase vía un *alias* como ser `WP` (por *working precision*, precisión de trabajo), la cual es utilizada para declarar los tipos de datos reales (variables y constantes). Entonces podemos escribir un programa que se compile ya sea con reales de simple o doble precisión escogiendo apropiadamente la sentencia que importa el módulo. Por ejemplo:

[2] IEEE = Institute of Electrical and Electronics Engineeers, IEC = International Electronical Commission.

Código 1. Uso del módulo `ISO_FORTRAN_ENV` para definir la precisión.

```
PROGRAM main
  USE iso_fortran_env, ONLY: WP => REAL32
                           ! ó WP => REAL64
  IMPLICIT NONE)
  REAL(WP) :: a

  a = 1.0_WP/3.0_WP
  WRITE (*,*) a

END PROGRAM main
```

Ejercicio 6. Fortran dispone de un conjunto de funciones intrínsecas para determinar las propiedades de la representación de punto flotante implementada en una clase de tipo real. Utilizando el siguiente programa verifique que en una computadora personal la clase asignada a los datos de tipo real son efectivamente los números de punto flotante de precisión simple y doble de la norma IEEE754.

Código 2. Funciones intrínsecas para las propiedades de la representación de punto flotante.

```
PROGRAM machar
  USE iso_fortran_env, ONLY: WP => REAL32
                           ! ó  WP => REAL64
  IMPLICIT NONE
  INTEGER :: i
  REAL(WP) :: x

  WRITE(*,*) ' base   = ', RADIX(i)
  WRITE(*,*) ' t      = ', DIGITS(x)
  WRITE(*,*) ' L      = ', MINEXPONENT(x)
  WRITE(*,*) ' U      = ', MAXEXPONENT(x)
  WRITE(*,*) ' x_max  = ', HUGE(x)
  WRITE(*,*) ' x_min  = ', TINY(x)
  WRITE(*,*) ' eps_M  = ', EPSILON(x)

END PROGRAM machar
```

¿Cuándo utilizar doble precisión?

La respuesta corta es *siempre*. Para las aplicaciones científicas la precisión de los resultados es generalmente crucial, con lo cual debe utilizarse la mejor representación de punto flotante disponible en la computadora. Nótese además que aún cuando los números de punto flotante de doble precisión utilizan el doble de dígitos binarios que los de simple precisión, en las computadoras personales (PC) el procesador matemático realiza internamente los cálculos con 80 bits independientemente de la precisión de los datos a ser procesados. Por lo tanto, la diferencia de velocidad entre cálculos en doble y simple precisión en una PC es ínfima.

Redondeo de un número real a su representación de punto flotante. El hecho de que sólo el subconjunto $\mathbb{F}$ de los números reales es representable en una computadora implica que dado cualquier número real x, para ser representado, debe ser aproximado por un número de punto flotante de tal conjunto, al que denotaremos por $fl(x)$. La manera usual de proceder consiste en aplicar el *redondeo simétrico* a t dígitos a la mantisa de la representación de punto flotante normalizada (infinita) de x. Esto es, a partir de

$$x = (-1)^s 0.a_1 a_2 \dots a_t a_{t+1} a_{t+2} \cdots \times \beta^e,$$

si el exponente e está dentro del rango $-L \leq e \leq U$, obtenemos $fl(x)$ como

$$fl(x) = (-1)^s 0.a_1 a_2 \dots \widetilde{a}_t \times \beta^e,$$

donde

$$\widetilde{a}_t = \begin{cases} a_t & \text{si } a_{t+1} < \beta/2 \\ a_t + 1 & \text{si } a_{t+1} \geq \beta/2. \end{cases}$$

El error que resulta de reemplazar un número real por su forma de punto flotante se denomina *error de redondeo*. Una estimación del mismo está dado en el siguiente resultado: *Todo número real x dentro del rango de los números de punto flotante puede ser representado con un error relativo que no excede la unidad de redondeo* $\mathbf{u}$:

$$\frac{|x - fl(x)|}{|x|} \leq \mathbf{u} = \frac{1}{2}\,\epsilon_M.$$

Se sigue de este resultado que existe un número real δ, que depende de x, tal que

$$fl(x) = x(1 + \delta), \qquad \text{siendo } |\delta| \leq \mathbf{u}.$$

Ejercicio 7. Determinar la representación de punto flotante decimal de 5 dígitos de π.

Ejercicio 8. Mostrar que en la representación de precisión simple de la norma IEEE754 el número de dígitos *decimales* significativos es alrededor de 7, mientras que para la precisión doble es alrededor de 16.

Aritmética de punto flotante. Además de dar una representación inexacta de los números, la aritmética realizada en la computadora no es exacta. Aún si x e y son números de punto flotante, el resultado de una operación aritmética sobre ellos no necesariamente es un número de punto flotante. En consecuencia, debe definirse una aritmética en $\mathbb{F}$ que sea lo más semejante posible a la aritmética de los números reales.

Si $\circ$ denota una operación (suma, resta, multiplicación o división) entre dos números reales x e y, la correspondiente operación de punto flotante denotada por $\circledcirc$ es definida como:

$$x \circledcirc y = fl(fl(x) \circ fl(y))$$

Esta aritmética consiste en efectuar la operación exacta en las representaciones de punto flotante de x e y

y luego convertir el resultado a su representación de punto flotante[3]. En particular, si x e y son *números de punto flotante*, se sigue que existe un número real δ tal que

$$x \circledcirc y = (x \circ y)(1+\delta), \qquad \text{siendo } |\delta| \leq \mathbf{u}.$$

Ejercicio 9. Utilizando aritmética de siete dígitos decimales efectuar los siguientes cálculos.

a) Con $a = 1234.567$, $b = 45.67844$, $c = 0.0004$,

$$(a+b)+c, \qquad a+(b+c).$$

b) Con $a = 1234.567$, $b = 1.234567$, $c = 3.333333$,

$$(a+b)\cdot c, \qquad a\cdot c + b\cdot c.$$

Al comparar los resultados, ¿qué puede concluirse?

> ✏ **Propiedades de la aritmética de punto flotante**
>
> No todas las propiedades de las operaciones aritméticas con números reales se preservan en la aritmética con números de punto flotante. En particular, aunque la adición y multiplicación de números de punto flotantes es conmutativa, no necesariamente es asociativa ni distributiva.

Ejercicio 10. Interprete el resultado del siguiente programa en virtud de la representación de punto flotante de los datos reales.

Código 3. Test de igualdad entre datos reales.

```
PROGRAM test_igualdad
  USE iso_fortran_env, ONLY: WP =>  REAL64
  IF ( 19.08_WP + 2.01_wp == 21.09_WP ) THEN
     WRITE(*,*) '19.08+2.01 = 21.09'
  ELSE
     WRITE(*,*) '19.08+2.01 es distinto a 21.09'
  ENDIF
END PROGRAM test_igualdad
```

> ✏ **Números de punto flotante en tests de igualdad**
>
> La comparación `x == y` entre dos datos de tipo numérico testea la igualdad estricta de las representaciones de punto flotante de los mismos. Debido a la naturaleza inexacta de la representación y los errores de redondeo involucrados en los cálculos que llevaron al valor de los datos, tal comparación puede dar un valor lógico falso aún cuando los datos que representan sean matemáticamente iguales. Por este motivo la igualdad estricta debe ser reemplazada por un test de igualdad dentro de cierta tolerancia: `abs(x-y) <= tol` donde el valor de `tol` depende del problema considerado.

[3] La implementación efectiva de estas operaciones en una computadora no procede exactamente de esta forma pero el resultado final se comporta como hemos indicado.

Números especiales. La condición de normalización sobre la mantisa de los números de punto flotante impide la representación del cero, por lo tanto debe disponerse de una representación separada del mismo. Por otra parte, en la aritmética de punto flotante pueden presentarse las tres siguientes condiciones excepcionales: *i)* una operación puede conducir a un resultado fuera del rango representable (ya sea porque $|x| > x_{\text{máx}}$ –*overflow*– o porque $|x| < x_{\text{mín}}$ –*underflow*–), *ii)* el cálculo puede ser una operación matemática indefinida (tal como la división por cero) o *iii)* ser ilegal (como la división $0/0$). Antes de la implementación de la norma IEEE754, frente a tales situaciones excepcionales, las computadoras abortaban el cálculo y detenían el programa. Por el contrario, la norma IEEE754 define una aritmética *cerrada* en $\mathbb{F}$ introduciendo ciertos números especiales. De esta manera, con la implementación de la norma IEEE754 en las computadoras actuales, cuando un cálculo intermedio conduce a una de las situaciones excepcionales el resultado es asignado al número especial apropiado y los cálculos continúan (*aritmética de no detención*).

Ceros. En la norma IEEE754 el cero es representado por un número de punto flotante con una mantisa nula y exponente $e = L-1$, pero, como ninguna condición es impuesta sobre el signo, existen dos ceros: $+0$ y -0 (con la salvedad de que en una comparación se consideran iguales en vez de $-0 < +0$). Un cero con signo es útil en determinadas situaciones, pero en la mayoría de las aplicaciones el signo del cero es invisible.

Infinitos. Cuando un cálculo produce un desbordamiento (*overflow*) positivo el resultado es asignado al número especial denominado *infinito positivo*, codificado como `+Infinity`[4]. De la misma manera, el cálculo de un desbordamiento negativo es asignado al número especial *infinito negativo*, codificado como `-Infinity`. Los infinitos permiten considerar también el caso excepcional de la división de un número no nulo por cero: el resultado es asignado al infinito del signo apropiado. Los infinitos son representados en el estandar por los números de punto flotante con mantisa nula y exponente $e = U+1$ con el correspondiente signo.

Números denormalizados. Tradicionalmente si una operación producía un valor de magnitud menor que $x_{\text{mín}}$ (desbordamiento a cero, o *underflow*), el resultado era asignado a cero. Ahora bien, la distancia entre cero y $x_{\text{mín}} = \beta^{L-1}$ (el menor número de punto flotante positivo representable) es mucho mayor que la distancia entre este número y el siguiente por lo que la asignación a cero de una condición de *underflow* produce errores de redondeo excepcionalmente grandes. Para cubrir esta distancia y reducir así el efecto de desbordamiento a cero a un nivel comparable con el redondeo de los números de pun-

[4] Nótese que "infinito" no significa necesariamente que el resultado sea realmente ∞, sino que significa "demasiado grande para representar".

to flotante se implementa el *desbordamiento a cero gradual* (*gradual underflow*) introduciendo los *números de punto flotante denormalizados*. Los números denormalizados son obtenidos removiendo en la representación de punto flotante la condición de que a_1 sea no nulo *solo* para los números que corresponden al mínimo exponente $e = L$. De esta manera la unicidad de la representación es mantenida y ahora es posible disponer de números de punto flotante en el intervalo $(-\beta^{L-1}, \beta^{L-1})$. La magnitud del más pequeño de estos números denormalizados es igual a β^{L-t}. De este modo, cuando el resultado de una operación tiene magnitud menor que $x_{\text{mín}}$ el mismo es asignado al correspondiente número de punto flotante denormalizado más próximo. En el estandar, los números denormalizados son representados como números de punto flotante con mantisa no nula y exponente $e = L - 1$.

NaN. Operaciones matemáticamente ilegales, como $0/0$ ó $\sqrt{x}$ para $x < 0$, son asignadas al número especial denominado *Not a Number* (no es un número), codificado como `NaN`. En el estandar un `NaN` es representado por un número de punto flotante con mantisa no nula y exponente $e = U + 1$ (puesto que la mantisa no está especificada no existe un único `NaN`, sino un conjunto finito de ellos los cuales pueden utilizarse para especificar situaciones de excepción particulares).

Las operaciones aritméticas que involucran a los números especiales están definidas de manera de obtener resultados razonables, tales como

$$
\begin{gathered}
(\pm\texttt{Infinity}) + (+1) = \pm\texttt{Infinity} \\
(\pm\texttt{Infinity}) \cdot (-1) = \mp\texttt{Infinity} \\
(\pm\texttt{Infinity}) + (\pm\texttt{Infinity}) = \pm\texttt{Infinity} \\
(\pm\texttt{Infinity}) + (\mp\texttt{Infinity}) = \texttt{NaN} \\
1/(\pm 0) = \pm\texttt{Infinity} \quad 1/(\pm\texttt{Infinity}) = \pm 0 \\
0/0 = \texttt{NaN} \\
(\pm\texttt{Infinity})/(\pm\texttt{Infinity}) = \texttt{NaN} \\
0 \cdot (\pm\texttt{Infinity}) = \texttt{NaN}
\end{gathered}
$$

Por otra parte un `NaN` se propaga agresivamente a través de las operaciones aritméticas: *en cualquier operación donde un* `NaN` *participe como operando el resultado será un* `NaN`.

Ejercicio 11. Compilar y ejecutar el siguiente programa para mostrar la acción de los números especiales.

Código 4. Números especiales.

```
PROGRAM excepciones
 IMPLICIT NONE
 INTEGER :: i
 REAL :: x,x_max,x_min,x_min_den,cero
 x_max = HUGE(x)
 x_min = TINY(x)
 x_min_den = REAL(RADIX(i))**(MINEXPONENT(x) &
 - DIGITS(x))
 cero = 0.0
 WRITE(*,*) 'Desbordamiento =', 2.0*x_max
 WRITE(*,*) 'Desbordamiento gradual a cero =',&
 x_min/2.0
 WRITE(*,*) 'Menor numero denormalizado =', &
            x_min_den
 WRITE(*,*) 'Desbordamiento a cero =', &
            x_min_den/2.0
 WRITE(*,*) 'Division por 0 =', 1.0/cero
 WRITE(*,*) '0/0 =', cero/cero
 WRITE(*,*) 'NaN + 1 =', cero/cero + 1.0
END PROGRAM excepciones
```

Ejercicio 12. Determinar los números de punto flotante denormalizados positivos asociados al conjunto $\mathbb{F}(2, 3, -1, 2)$.

> **Anulando la aritmética de no detención de la norma IEEE74.**
>
> La filosofía detrás de la aritmética de no detención de la norma IEEE754 es que el sistema de punto flotante extendido simplifica la programación en algunos casos, en particular cuando los cálculos involucran puntos singulares. Sin embargo, muchos usuarios la encuentran confusa y prefieren que los cálculos sean abortados con un apropiado mensaje de error. Para anular el comportamiento del estandar en las situaciones excepcionales se puede utilizar la opción `-ffpe-trap=invalid,zero,overflow, underflow,denormal` en la compilación del programa con el compilador `gfortran`.

Propagación del error de redondeo. Un punto de mucha importancia para el análisis numérico es analizar la forma en que un error de redondeo introducido en algún punto de un cálculo se *propaga* en los cálculos posteriores. Los siguientes ejercicios ilustran los problemas que pueden ocurrir al trabajar con una representación finita de los números reales.

Ejercicio 13. Considere la suma de cuatro números positivos:

$$y = x_1 + x_2 + x_3 + x_4.$$

a) Mostrar que si los x_i son números de punto flotante, en una aritmética de t dígitos, el error de redondeo en y está acotado por

$$|\Delta y| \leq (3x_1 + 3x_2 + 2x_3 + x_4)\,\mathbf{u}.$$

b) Mostrar que el error de redondeo se minimiza reacomodando los números a sumar de manera que los más pequeños sean los que se sumen primero.

Ejercicio 14. Implementar un programa Fortran para evaluar la suma (en precisión simple)

$$\sum_{n=1}^{10\,000\,000} 1/n,$$

primero en el orden usual y luego en el orden opuesto. Explique las diferencias obtenidas e indique cual es el resultado más preciso.

Ejercicio 15. Supóngase que x e y son números positivos correctamente redondeados a t dígitos. Mostrar que la magnitud del error relativo de redondeo en $z = x - y$ está acotada por

$$\left|\frac{\Delta z}{z}\right| \leq \frac{|x| + |y|}{|x - y|}\,\mathbf{u} + \mathbf{u}.$$

Mostrar, entonces, que si x e y son aproximadamente iguales, los errores de redondeo de x e y pueden propagarse de manera tal que el error relativo en z puede ser grande aunque el error absoluto sea pequeño (*fenómeno de cancelación de dígitos significativos*).

El siguiente ejercicio muestra que la pérdida de precisión en la resta de dos números aproximadamente iguales puede tener un efecto drástico en expresiones que contengan dicha sustracción.

Ejercicio 16. La fórmula cuadrática nos dice que las raíces de $ax^2 + bx + c = 0$ son

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

Si $b^2 \gg 4ac$ entonces, cuando $b > 0$ el cálculo de x_1 involucra en el numerador la sustracción de dos números casi iguales, mientras que si $b < 0$ esta situación ocurre para el cálculo de x_2. "Racionalizando el numerador" se obtienen las siguientes fórmulas alternativas que no sufren de este problema:

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

siendo la primera adecuada cuando $b > 0$ y la segunda cuando $b < 0$.

Utilizar la fórmula usual y la "racionalizada" para calcular las raíces de

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

utilizando aritmética a cuatro dígitos. Interprete sus resultados.

Ejercicio 17. Considere el polinomio

$$p(x) = x^3 - 6x^2 + 3x - 0.149,$$

a) Evalue el polinomio en $x = 4.71$ utilizando aritmética decimal de tres dígitos. Estime el error relativo cometido.
b) Repita el punto anterior pero con el polinomio escrito en su forma *anidada*

$$p(x) = ((x - 6)x + 3)x - 0.149.$$

c) ¿A qué se debe el decrecimiento del error en el segundo caso?

✏ **Lista de recomendaciones para minimizar el impacto del error de redondeo.**

- Evitar testear la igualdad entre dos números, en su lugar testear la magnitud de la diferencia dentro de cierta tolerancia.
- Cuando se van a sumar y/o restar números, considerar siempre los números más pequeños primero.
- De ser posible evitar la sustracción de dos números aproximadamente iguales. Una expresión que contenga dicha sustracción puede a menudo ser reescrita para evitarla.
- Minimizar el número de operaciones aritméticas.

Inestabilidad numérica Los ejercicios anteriores ilustran un resultado importante: *algoritmos matemáticamente equivalentes no necesariamente son numéricamente equivalentes*. Por *equivalencia matemática* de dos algoritmos queremos decir que los algoritmos dan los mismos resultados para los mismos datos de entrada suponiendo que los cálculos son realizados sin errores de redondeo. En tal caso un algoritmo puede ser derivado uno del otro con ayuda de las reglas del álgebra e identidades matemáticas. Dos algoritmos se dicen *numéricamente equivalentes* si sus resultados respectivos, usando los mismos datos de entrada, no difieren más de lo que los datos de salida exactos del problema cambiarían si los datos de entrada fueran modificados en unas pocas unidades de la unidad de redondeo **u**. Más aún, los ejemplos muestran que los errores de redondeo en los cálculos involucrados en un algoritmo pueden producir pérdida de precisión en los resultados e incluso destruir completamente el resultado exacto (fenómeno denominado *inestabilidad numérica*). A la luz de una aproximación del análisis del error, conocida como *análisis inverso del error*, se puede mostrar que los resultados que un algoritmo produce bajo la influencia de los errores de redondeo, son el resultado exacto de un problema del mismo tipo en el cual los datos de entrada están perturbados por cantidades de cierta magnitud. De esta forma transferimos el problema de estimar los efectos del redondeo *durante* los cálculos de un algoritmo, al problema de estimar los efectos de perturbar los datos de entrada. Esto permite, entonces, establecer la siguiente definición:

Un algoritmo se dice *numéricamente estable* si pequeños cambios en los datos iniciales, de magnitudes (relativas) del orden de la unidad de redondeo **u**, producen en correspondencia pequeños cambios en los resultados finales. De lo contrario, se dice que el algoritmo es *numéricamente inestable*.

Nótese que algunos algoritmos son estables para cierto grupo de datos iniciales pero no para todos. Además algunos problemas son numéricamente inestables *independientemente* de la elección del algoritmo. En este caso se dice que el problema está *mal condicionado* o que ésta *mal planteado*.

Ejercicio 18. Considere el problema de calcular las integrales

$$I_n = \int_0^1 x^n \exp(x-1)\,\mathrm{d}x, \quad \text{para } n = 1, 2, \ldots$$

Es claro de la definición que $I_1 > I_2 > \ldots > I_{n-1} > I_n > \ldots > 0$.

a) La integración por partes proporciona el siguiente procedimiento recursivo para la evaluación (¡verificarlo!):

$$\begin{cases} I_1 = e^{-1}, \\ I_n = 1 - nI_{n-1} \quad n \geq 2. \end{cases}$$

Usando aritmética de simple precisión calcular los primeros $n = 20$ valores de I_n. ¿Tienen sentido los resultados obtenidos? Analice la estabilidad numérica del proceso iterativo suponiendo que se comienza con un valor perturbado $\hat{I}_1 = I_1 + \delta$ de I_1.

b) Considere ahora la fórmula recursiva escrita en forma inversa, junto con la (cruda) aproximación $I_{20} = 0$:

$$\begin{cases} I_{20} = 0, \\ I_{n-1} = \dfrac{1 - I_n}{n} \quad n = 20, 19, \ldots, 2. \end{cases}$$

Estime los primeros $n = 20$ valores de I_n con dicho algoritmo. ¿Tienen sentido los resultados obtenidos? Analice la estabilidad numérica del algoritmo.