Tema 1 - Escalas, condición y estabilidad

Curso de Física Computacional

M. en C. Gustavo Contreras Mayén

Facultad de Ciencias - UNAM

19 de febrero de 2018

Los números en la computadora

Las computadoras son herramientas muy poderosas, pero tienen un alcance finito.

Un problema que se presenta en el diseño de computadora es cómo representar un número arbitrario usando una cantidad finita de espacio de memoria y cómo tratar con las limitaciones que surgen de esta representación.

Los números en la computadora

Como consecuencia de que las memorias de la computadora se basan en un estado de magnetización o electrónico de un giro que apunta hacia arriba o hacia abajo, las unidades más elementales de memoria de la computadora son los dos enteros binarios (bits) `0` y `1`.

Los números en la computadora

Esto significa que la forma en que se almacenan los números en memoria, es como cadenas largas de ceros y unos, es decir, de modo binario.

Rango de representación

De esta manera: N bits almacena números enteros en el rango $[0, 2^N]$, pero debido a que el signo del número entero está representado por el primer bit (un bit cero para números positivos), el rango real disminuye a

$$[0, 2^{N-1}]$$

Otras bases de operación

La representación binaria de números a través de ceros y unos, funciona y opera bien para las computadoras, pero no para los usuarios.

Es por ello que las cadenas binarias se convierten en números octal, decimal o hexadecimal antes de que los resultados se presenten a los usuarios.

Otras bases de operación

Los números octales y hexadecimales son también oportunos porque en la conversión no se pierde precisión.

Pero no todo queda bien ya que nuestras reglas decimales de aritmética no funcionan para ellos.

Desventaja de la aritmética binaria

La conversión a números decimales hace que los números sean más fáciles de trabajar, pero a menos que el número sea una potencia de 2, el proceso conduce a una disminución en la precisión.

Longitud de palabra

Una descripción de un sistema informático particular indica normalmente la *longitud de la palabra*, siendo el número de bits utilizados para almacenar un número.

La longitud se expresa en bytes, donde

$$1 \text{ byte} \equiv 1 \text{ B} \equiv 8 \text{ bit}$$

Unidades de medida

Tanto la memoria como el almacenamiento se mide en bytes, kilobytes, megabytes, gigabytes, terabytes y petabytes (10^{15}).

No debemos de confundirnos al elegir las unidades de medida, ya que el prefijo *kilo*, no siempre equivale a 1000

$$1\,\text{k} = 1\,\text{kB} = 2^{10}\text{B} = 1024\,\text{byte}$$

Unidades de medida

La memoria de las primeras computadoras usaban palabras de 8 bits, esto implicaba que el mayor entero era $2^7 = 128$, (7 debido a 1 bit para el signo).

Overflow y underflow

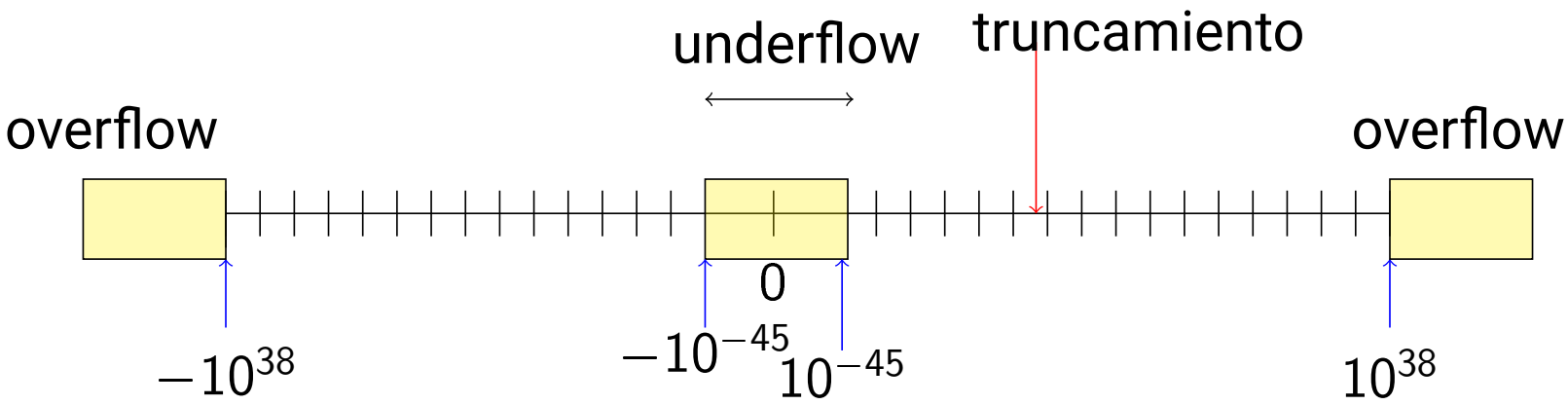
Si queríamos almacenar un número más grande, tanto el hardware como el software se diseñaban para generar un desbordamiento (overflow).

Se genera un underflow cuando queremos representar un número más pequeño del que se puede en el equipo.

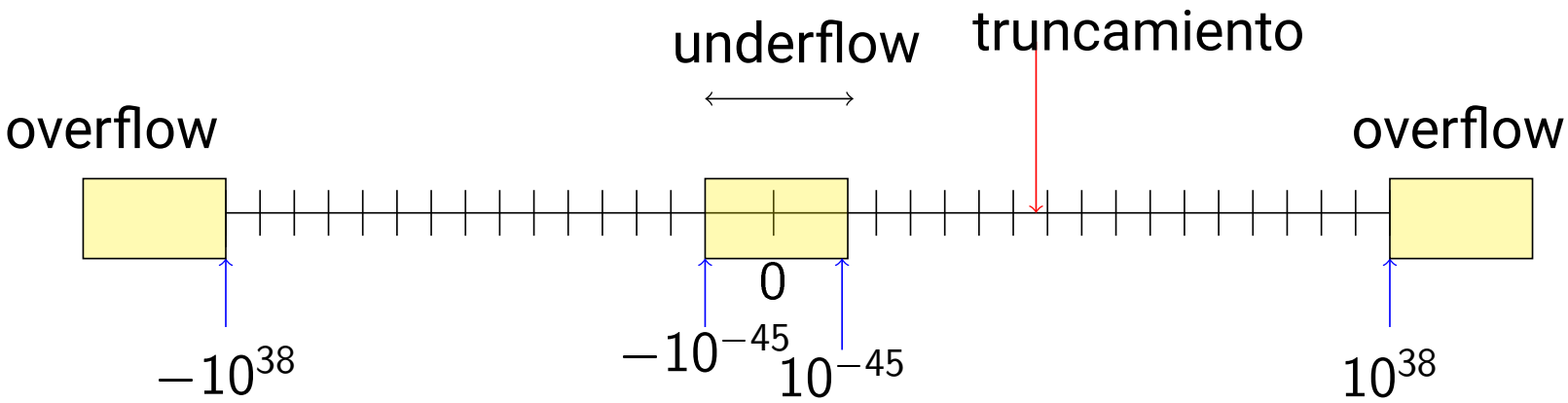
Errores de almacenamiento: overflow

Los números de máquina tienen un máximo y un mínimo.

Si se supera el máximo, se produce una condición de error conocida como desbordamiento -overflow-.



Errores de almacenamiento: underflow



Si se cae por debajo del mínimo, se produce una condición de error conocida como subflujo -underflow-.

Manejo del overflow y del underflow

En este último caso, el software y el hardware se pueden configurar para que los overflow se pongan a cero sin que se le avise al usuario.

En contraste, los overflow suelen detener la ejecución.

Alcance de 64 bits

Usando 64 bits, se permiten enteros en el rango $1 - 2^{63} \simeq 10^{19}$.

Que podría en apariencia ser una rango mucho más grande, pero no lo es cuando se compara con el rango de escalas que tenemos en el mundo real: del radio del universo al radio de un protón, tenemos aproximadamente 10^{41} órdenes de magnitud.

Tipos de notación

Los números reales se representan en computadoras en notación de punto fijo o punto flotante.

La notación de punto fijo se puede utilizar para números con un número fijo de lugares luego del punto decimal (raíz) o para números enteros.

Tipos de notación

Tiene como ventaja que se utiliza la aritmética complementaria de dos y por tanto, almacenar enteros exactamente.[1]

[1] Para ampliar el tema, revisa la guía de operaciones binarias.

Números de punto fijo

En la representación de punto fijo con N bits y con un formato de complemento de dos, un número se representa como

$$N_{\text{fijo}} = \text{signo} \times (\alpha_n \, 2^n + \alpha_{n-1} \, 2^{n-1} + \ldots + \\ + \alpha_0 \, 2^0 + \ldots + \alpha_{-m} \, 2^{-m}) \tag{1}$$

donde $n + m = N - 2$.

Números de punto fijo

Es decir, 1 bit se utiliza para almacenar el signo, con los restantes $(N-1)$ bits se usan para almacenar los α_i valores (se entienden las potencias de 2).

Los valores particulares para N, m y n son dependientes de la máquina.

Rango de los enteros

Los enteros se representan típicamente con 4 bytes (32 bits) de longitud, en el rango

$$-2147483648 \leq 4 - \text{B entero} \leq 2147483647$$

Ventaja de la representación

Una ventaja de la representación (1) es que se puede contar con todos los números de punto fijo para tener el mismo error absoluto de 2^{-m-1} (el término se deja fuera del extremo derecho de (1).

Desventaja de la representación

La correspondiente desventaja es que los números pequeños (aquellos para los cuales la primera cadena de valores de α son ceros) tienen grandes errores relativos.

Utilidad de los enteros

Dado que en el mundo real los errores relativos tienden a ser más importantes que los absolutos, los números enteros se utilizan principalmente para fines de conteo y en aplicaciones especiales.

Números de punto flotante

En la mayoría de los cálculos científicos se utilizan números de punto flotante de doble precisión (64 bits = 8 Bytes).

La representación en las computadoras de números en punto flotante de números es una versión binaria de lo que comúnmente se conoce como notación científica.

Notación científica

Por ejemplo, la velocidad de la luz en notación científica

$$c = +2.997\,924\,58 \times 10^{8}\,\mathrm{m\,s^{-1}}$$

y en notación de ingeniería

$$+0.299\,792\,458 \times 10^{9}\,\mathrm{m\,s^{-1}} \text{ ó } 0.299\,795\,498 \times 10^{9}\,\mathrm{m\,s^{-1}}$$

Notación científica

$$c = +2.997\,924\,58 \times 10^{8}\,\mathrm{m\,s^{-1}}$$

En cada uno de estos casos, el número al frente se denomina **mantisa** y contiene nueve cifras significativas.

La potencia 10 a la que se eleva se llama **exponente**, con el signo más $(+)$ incluido, como recordatorio de que estos números pueden ser negativos.

Almacenamiento

Los números de punto flotante se almacenan en la computadora como una concatenación del bit de signo, el exponente y la mantisa.

Almacenamiento

Debido a que sólo se almacena un número finito de bits, el conjunto de números de punto flotante que el equipo puede almacenar exactamente, es decir, el conjunto de **números de máquina**, es mucho menor que el conjunto de los números reales.

Valor en memoria

La relación real entre lo que se almacena en la memoria y el valor de un número de punto flotante es algo indirecta, habiendo un número de casos especiales y relaciones utilizadas a lo largo de los años.

Valor en memoria

En el pasado, cada sistema operativo de computadora y cada lenguaje de programación contenían sus propios estándares para números de punto flotante.

Valor en memoria

El que haya diferentes normas implica que el mismo programa que se ejecuta correctamente en un equipo, podría dar resultados diferentes cuando se utilice en otros equipos con configuración distinta.

Valor en memoria

A pesar de que los resultados eran sólo ligeramente diferentes, el usuario nunca podría estar seguro de si la falta de reproducibilidad de un caso de prueba se debía a la computadora particular que se está utilizando o a un error en la implementación del programa.

Estándar IEEE 754

En 1987, el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) y el American National Standards Institute (ANSI) adoptaron el estándar **IEEE 754** para la aritmética de punto flotante.

Estándar IEEE 754

Cuando se sigue el estándar, se espera que los tipos de datos primitivos tengan la precisión y los intervalos dados en la siguiente tabla:

IEEE 754 para Tipos de Datos Primitivos

Nombre	Tipo	Bits	Byte	Rango
`boolean`	Lógico	1	$\frac{1}{8}$	`True` o `False`
`char`	String	16	2	`'u0000'` $\longleftrightarrow$ `'uFFFF'`
`byte`	Integer	8	1	$-128 \longleftrightarrow +127$
`short`	Integer	16	2	$-32768 \longleftrightarrow +32767$
`int`	Integer	32	4	checar en tablas
`long`	Integer	64	8	checar en tablas
`float`	Floating	32	4	checar en tablas
`double`	Flaoting	64	8	checar en tablas

Apego al IEEE 754

Cuando las computadoras y el software se apegan a este estándar, está garantizado que un programa producirá resultados idénticos en diferentes equipos.

Apego al IEEE 754

En realidad hay una serie de componentes en el estándar IEEE, y diferentes fabricantes de computadoras o procesadores pueden apegarse a sólo algunos de ellos.

Además, `python` puede no seguir a todos a medida que se desarrolla, pero probablemente con el tiempo lo haga.

Ńotación de punto flotante

Normalmente un número de punto flotante x se almacena como

$$x_f = (-1)^s \times 1.f \times 2^{e-\text{sesgo}} \tag{2}$$

esto es, con partes por separado para:

- El signo s.

Ńotación de punto flotante

Normalmente un número de punto flotante x se almacena como

$$x_f = (-1)^s \times 1.f \times 2^{e-\text{sesgo}} \tag{2}$$

esto es, con partes por separado para:

- El signo s.
- La parte fraccional de la mantisa f.

Ńotación de punto flotante

Normalmente un número de punto flotante x se almacena como

$$x_f = (-1)^s \times 1.f \times 2^{e-\text{sesgo}} \tag{2}$$

esto es, con partes por separado para:

- El signo s.
- La parte fraccional de la mantisa f.
- El valor del exponente e.

Almacenamiento en punto flotante

Todas las partes se almacenan en forma binaria y ocupan segmentos adyacentes de una sola palabra de 32 bits o dos palabras adyacentes de 32 bits para palabras dobles.

El signo s se almacena como un solo bit, con $s = 0$ ó 1 para un signo positivo o negativo.

Se usan ocho bits para almacenar el exponente e, lo que significa que e puede estar en el rango $0 \leq e \leq 255$.

Los extremos, $e = 0$ y $e = 255$, son casos especiales.

Casos especiales del estándar IIEEE

Nombre del número	Valores de s, e y f	Valor
Normal	$0 < e < 255$	$(-1)^s \times 2^{e-127} \times 1.f$
Subnormal	$e = 0, f \neq 0$	$(-1)^s \times 2^{e-126} \times 0.f$
Signo cero (± 0)	$e = 0, f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0, e = 255, f = 0$	**+INF**
$-\infty$	$s = 1, e = 255, f = 0$	**-INF**
Not a number	$s = u, e = 255, f \neq 0$	**NaN**

Números normales

Los números normales tienen un valor entre $0 < e < 255$, y con ellos la convención es suponer que el primer bit de la mantisa es un 1.

De modo que sólo se almacena la parte fraccional f después del punto binario.

Valores $\pm$ INF y NaN

Nótese que los valores $\pm$ INF y NaN no son números en el sentido matemático, es decir, son objetos que pueden ser manipulados o utilizados en cálculos para tomar límites

Valores $\pm$ INF y NaN

Nótese que los valores $\pm$ INF y NaN no son números en el sentido matemático, es decir, son objetos que pueden ser manipulados o utilizados en cálculos para tomar límites

Más bien, son señales para la computadora y para el usuario de que algo ha ido mal y que el cálculo probablemente debería detenerse hasta resolver las cosas.

Los valores ± 0

En contraste, el valor -0 se puede utilizar en un cálculo sin problemas.

Algunos lenguajes pueden establecer variables no asignadas como -0 como una pista de que aún no se han asignado, pero no es lo más conveniente.

Precisión relativa en punto flotante

Debido a que la incertidumbre (error) está presente sólo en la mantisa y no en el exponente, las representaciones IEEE aseguran que todos los números normales de punto flotante tengan la misma precisión relativa.

El bit fantasma

Debido a que el primer bit se supone que es 1, no tiene que ser almacenado, y los diseñadores de computadoras sólo necesitan recordar que hay un *bit fantasma* allí para obtener un poco más de precisión.

El primer bit

Durante el procesamiento de números en un cálculo, el primer bit de un resultado intermedio puede llegar a ser cero, pero éste se cambia antes de que se almacene el número final.

Para repetir, en los casos normales, la mantisa real ($1.f$ en notación binaria) contiene un 1 implícito que precede al punto binario.

El número de sesgo

Con el fin de garantizar que el exponente e almacenado sea siempre positivo, se agrega un número fijo llamado sesgo al exponente real p, antes de que se almacene como exponente e.

El exponente real, que puede ser negativo, es

$$p = e - \text{sesgo} \tag{3}$$

Formatos básicos del IEEE

Hay dos formatos básicos para el IEEE de punto flotante, de precisión simple y de precisión doble.

Precisión simple

La precisión simple representa a los números de punto flotante de precisión simple (sencilla).

Ocupan 32 bits en total: con 1 bit para el signo, 8 bits para el exponente y 23 bits para la mantisa fraccional (lo que da una precisión de 24 bits cuando el bit fantasma es incluido)

Precisión doble

Es una abreviatura para los números de punto flotante de precisión doble.

Ocupan 64 bits en total: con 1 bit para el signo, 10 bits para el exponente y 53 bits para la mantisa fraccional (para precisión de 54 bits).

Precisión doble

Esto significa que los exponentes y mantisas para la precisión doble no son simplemente el doble de los floating.

Representación de 32 bits

Para ver el esquema en la práctica consideremos la representación de 32 bits:

	s	e		f	
Bit	31	30	23	22	0

Representación de 32 bits

Para ver el esquema en la práctica consideremos la representación de 32 bits:

	s	e		f	
Bit	31	30	23	22	0

El bit de signo s está en la posición 31, el exponente con sesgo, está en los bits $30 - 23$, y la mantisa fraccional f, en los bits $22 - 0$.

Exponente en 32 bits

Ya que se usan 8 bits para almacenar el exponente e, y $2^8 = 256$, el exponente tiene un rango

$$0 \leq e \leq 255$$

Exponente en 32 bits

Ya que se usan 8 bits para almacenar el exponente e, y $2^8 = 256$, el exponente tiene un rango

$$0 \leq e \leq 255$$

Con el sesgo = 127_{10}, el exponente completo queda como

$$p = e_{10} - 127$$

Rango del exponente

Para la precisión sencilla el exponente tiene el rango

$$-126 \leq p \leq 127$$

La mantisa f se almacena en los 23 bits con posición $22 - 0$.

Para los números normales, esto es, para aquellos con $0 < e < 255$, f es la parte fraccional de la mantisa

Número en 32 bits

Por tanto, el número normal de punto flotante representado en 32 bits es

$$(-1)^s \times 1.f \times 2^{e-127}$$

Números subnormales

Los números subnormales tienen $e = 0$, $f \neq 0$; para estos números, f es la mantisa completa.

Por lo que el número subnormal representado en 32 bits es

$$(-1)^s \times 0.f \times 2^{e-126} \tag{4}$$

Representación de la mantisa

Los 23 bits $m_{22} - m_0$ que se utilizan para almacenar la mantisa de números normales, corresponde a la notación

$$\text{Mantisa } = 1.f = 1 + m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \ldots + \\ + m_0 \times 2^{-23} \tag{5}$$

Representación de la mantisa

Los 23 bits $m_{22} - m_0$ que se utilizan para almacenar la mantisa de números normales, corresponde a la notación

$$\text{Mantisa } = 1.f = 1 + m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \ldots + \\ + m_0 \times 2^{-23} \tag{5}$$

Para los números subnormales, se usa $0.f$

Ejemplo

Veamos un ejemplo: el mayor número positivo de punto flotante de precisión simple para una máquina de 32 bits, tiene el valor máximo para $e = 254$ y el valor máximo para f:

$$\begin{aligned} X_{max} &= 01111111011111111111111111111111 \\ &= (0)(11111110)(11111111111111111111111) \end{aligned} \tag{6}$$

donde se han agrupado los bits.

Se tienen entonces los siguiente valores

$$
\begin{aligned}
s &= 0 \\
e &= 11111110 = 254 \\
p &= e - 127 = 127 \\
f &= 1.11111111111111111111111 = 1 + 0.5 + 0.25 + . \\
&\rightarrow (-1)^s \times 1.f \times 2^{p=e-127} \simeq 2 \times 2^{127} \simeq 3.4 \times 10^{38}
\end{aligned}
\tag{7}
$$

Ejercicio a cuenta

Ahora te toca desarrollar las cuentas para evaluar el número positivo de punto flotante de precisión simple más pequeño subnormal, con $e = 0$, y con una mantisa:

$$0\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 001$$

Solución

Acomodamos los términos para un manejo más sencillo

$$s = 0$$

$$e = 0$$

$$p = e - 126 = -126$$

$$f = 0.0000\ 0000\ 0000\ 0000\ 0000\ 001 = 2^{-23}$$

$$\rightarrow (-1)^s \times 0.f \times 2^{p=e-126} = 2^{-149} \simeq 1.4 \times 10^{-45}$$

Resumen de la precisión simple

Podemos dejar como resumen que los números de precisión simple, tienen 6 ó 7 decimales de significancia y las magnitudes están en el rango

$$1.4 \times 10^{-45} \leq \text{ precision simple } \leq 3.4 \times 10^{38}$$

Representación de la precisión doble

Los números de precisión doble se almacenan en dos palabras de 32 bits, para un total de 64 bits (8) bytes.

Representación de la precisión doble

Los números de precisión doble se almacenan en dos palabras de 32 bits, para un total de 64 bits (8) bytes.

	s	e		f		f cont	
Bit	63	62	52	51	32	31	0

Representación de la precisión doble

El signo ocupa 1 bit, el exponente e ocupa 11 bits, la mantisa fraccional 52 bits.

Los valores se almacenan de manera contigua, en donde parte de la mantisa f se almacena en un palabra de 32 bits.

Valor del sesgo

Para los números de precisión doble, el valor del seso es mayor que en los de precisión simple

$$\text{sesgo } = 1111111111_2 = 1023_{10}$$

por lo que el exponente real es $p = e - 1023$.

Rangos en precisión doble

Los números en precisión doble tienen aproximadamente 16 decimales de precisión (1 parte en 2^{52}), y las magnitudes están en el rango

$$4.9 \times 10^{-324} \leq \text{ precision doble } \leq 1.8 \times 10^{308}$$

Hablando de errores

Para entender el por qué los errores deben ser motivo de preocupación, imaginemos un programa con el siguiente flujo lógico

$$\text{Inicio} \rightarrow U_1 \rightarrow U_2 \rightarrow \ldots \rightarrow U_n \rightarrow \text{Fin} \tag{8}$$

Hablando de errores

$$\text{Inicio} \rightarrow U_1 \rightarrow U_2 \rightarrow \ldots \rightarrow U_n \rightarrow \text{Fin}$$

Donde cada unidad U puede ser una declaración de una tarea o un paso.

Si cada unidad tiene probabilidad p de ser correcta, entonces la probabilidad conjunta P de que todo el programa sea correcto es $P = p^n$.

Hablando de errores

Digamos que tenemos un programa de tamaño mediano con $n = 1000$ pasos y que la probabilidad de que cada paso sea correcto es casi uno, $p \simeq 0.9993$.

Hablando de errores

Esto significa que terminas con $P \simeq \frac{1}{2}$, es decir, la respuesta final es tan probablemente correcta como equivocada.

Hablando de errores

El problema es que como científicos, queremos un resultado correcto o al menos en el que la incertidumbre sea pequeña y de tamaño conocido.

Fuentes de errores computacionales

Existen al menos cuatro fuentes de errores que se pueden presentar en los cálculos computacionales:

Fuentes de errores computacionales

Existen al menos cuatro fuentes de errores que se pueden presentar en los cálculos computacionales:

1. Un modelo equivocado.

Fuentes de errores computacionales

Existen al menos cuatro fuentes de errores que se pueden presentar en los cálculos computacionales:

1. Un modelo equivocado.
2. Errores aleatorios.

Fuentes de errores computacionales

Existen al menos cuatro fuentes de errores que se pueden presentar en los cálculos computacionales:

1. Un modelo equivocado.
2. Errores aleatorios.
3. Errores por aproximación

Fuentes de errores computacionales

Existen al menos cuatro fuentes de errores que se pueden presentar en los cálculos computacionales:

1. Un modelo equivocado.
2. Errores aleatorios.
3. Errores por aproximación
4. Errores por redondeo.

Un modelo equivocado

Se presentan cuando nuestra propuesta de modelo no es el pertinente, siendo quizá la parte más crítica ya que antes de continuar, debemos de repasar la parte de la física para resolver debidamente nuestro problema

Un modelo equivocado

También se consideran los errores tipográficos, introducir valores que no corresponden, ejecutar el programa incorrecto, usar el archivo de datos equivocado, etc.

Sugerencia:

Un modelo equivocado

También se consideran los errores tipográficos, introducir valores que no corresponden, ejecutar el programa incorrecto, usar el archivo de datos equivocado, etc.

Sugerencia: Si el número de errores comienza a crecer, es momento de ir a casa o tomar un descanso.

Errores aleatorios

Se presenta una imprecisión debida por acontecimientos tales como: fluctuaciones en la electrónica, incidencia de rayos cósmicos, o alguien que se tropieza con un enchufe.

Éstos errores pueden ser raros, pero no tenemos ningún control sobre ellos y su probabilidad aumenta conforme transcurre el tiempo.

Errores por aproximación

La imprecisión surge de la simplificación de las matemáticas para que un problema pueda ser resuelto en la computadora.

Incluyen la sustitución de series infinitas por sumas finitas, intervalos infinitesimales por finitos, y funciones variables por constantes.

Errores por aproximación

Por ejemplo

$$\sin(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}\, x^{2n-1}}{(2n-1)!} \quad \text{(exacta)} \tag{9}$$

donde $\varepsilon(x, N)$ es el error por la aproximación, en este caso ε corresponde a los términos desde $N+1$ hasta ∞.

Errores por aproximación

Por ejemplo

$$\sin(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}\, x^{2n-1}}{(2n-1)!} \quad \text{(exacta)}$$

$$\simeq \sum_{n=1}^{N} \frac{(-1)^{n-1}\, x^{2n-1}}{(2n-1)!} \quad \text{(algoritmo)} \tag{9}$$

Errores por aproximación

Por ejemplo

$$\sin(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}\, x^{2n-1}}{(2n-1)!} \quad \text{(exacta)}$$

$$\simeq \sum_{n=1}^{N} \frac{(-1)^{n-1}\, x^{2n-1}}{(2n-1)!} \quad \text{(algoritmo)} \tag{9}$$

$$= \sin(x) + \varepsilon(x, N)$$

donde $\varepsilon(x, N)$ es el error por la aproximación, en este caso ε corresponde a los términos desde $N + 1$ hasta ∞.

Errores por aproximación

Dado que el error por la aproximación se genera en el algoritmo que usamos para aproximar la matemática, también se le conoce como error del algoritmo.

Errores por aproximación

Para una buena y razonable aproximación, el error debido por la aproximación debería de reducirse cuando el valor de N se incrementa, y debería de anularse en el límite $N \to \infty$.

Errores por aproximación

Para el ejemplo (9), como la escala de N se fija por el valor de x, un pequeño error de aproximación requiere que $N \geqslant x$.

Por lo que si x y N son valores cercanos entre sí, el error de aproximación será grande.

Errores por redondeo

La imprecisión se genera por el número finito de dígitos utilizados para almacenar números de punto flotante.

Estos “errores” son análogos a la incertidumbre en la medición de una cantidad física encontrada en un laboratorio de física.

Errores por redondeo

El error general de redondeo se acumula a medida que el equipo maneja más números, es decir, a medida que aumenta el número de pasos en un cálculo, y puede hacer que algunos algoritmos se vuelvan inestables con un rápido aumento del error.

Errores por redondeo

En algunos casos, el error por redondeo puede convertirse en el componente principal en la respuesta, lo que lleva a lo que los expertos informáticos llaman **basura**.

Errores por redondeo

Por ejemplo, si la computadora mantiene cuatro decimales, entonces almacenará $\frac{1}{3}$ como 0.3333 y $\frac{2}{3} = 0.6667$, donde el equipo ha “redondeado” el último dígito en $\frac{2}{3}$.

Errores por redondeo

Por consiguiente, si hacemos en la computadora un cálculo tan simple como $2\frac{1}{3} - \frac{2}{3}$, produce

$$2\left(\frac{1}{3}\right) - \frac{2}{3} = 0.6666 - 0.6667 = -0.0001 \neq 0 \quad (10)$$

Errores por redondeo

Por consiguiente, si hacemos en la computadora un cálculo tan simple como $2\frac{1}{3} - \frac{2}{3}$, produce

$$2\left(\frac{1}{3}\right) - \frac{2}{3} = 0.6666 - 0.6667 = -0.0001 \neq 0 \quad (10)$$

Aunque el resultado es pequeño, no es 0, y si se repete este tipo millones de veces este cálculo, la respuesta final puede que ni siquiera sea pequeña (la basura genera basura).

Definiciones

Sea x^* el valor exacto de una cierta cantidad y x la aproximación a esa cantidad.

Por definición, el valor absoluto asociado a x es

$$\Delta = \left| x^* - x \right| \tag{11}$$

Definiciones

En general, los cálculos numéricos operan con aproximaciones tanto a las cantidades como a sus errores absolutos.

En el caso ideal, en el que, además de la aproximación x, también se conocería el error absoluto exacto, el número exacto sería exactamente expresable como

$$x^* = x \pm \Delta^* \tag{12}$$

Definiciones

Sin embargo, en general, solo está disponible una estimación del error absoluto y, en consecuencia, solo un estimado del valor exacto puede determinarse:

$$x - \Delta \leq x^* \leq x + \Delta \tag{13}$$

Definiciones

Para que la estimación x sea confiable, es importante que Δ no subestime el error verdadero, es decir, $\Delta \geq \Delta^*$, en este caso, se denomina *limitación de error absoluto*.

Más definiciones

El error relativo δ^* es la aproximación de x a x^*, y es igual al cociente del error absoluto con el número exacto:

$$\delta^* = \frac{\Delta^*}{|x^*|} \qquad x^* \neq 0 \tag{14}$$

Más definiciones

Re-emplazando Δ^* en la ecuación(12) de la ecuación (14), tendremos la expresión para un número exacto:

$$x^* \simeq x \left(1 \pm \delta^*\right) \tag{15}$$

Más definiciones

Basándose en el error relativo de la aproximación $\delta \geq \delta^*$, se expresa el valor exacto como:

$$x^* \simeq x \left(1 \pm \delta\right) \tag{16}$$

Errores en las operaciones elementales

Revisaremos la forma en que varias operaciones elementales propagan los errores relativos de sus operandos.

En relación con los cálculos numéricos, las consideraciones desarrolladas a continuación se refieren específicamente a los errores de redondeo y truncamiento.

Error en la suma

Sea la suma de dos números próximos, x_1 y x_2, que tienen el mismo signo.

La suma evidentemente conserva el signo y acumula las magnitudes de los operandos:

$$x = x_1 + x_2 \tag{17}$$

Error en la suma

Tenemos que el error total es

$$\Delta x = \Delta x_1 + \Delta x_2$$

Error en la suma

Tenemos que el error total es

$$\Delta x = \Delta x_1 + \Delta x_2$$

pero, dado que los errores individuales pueden, en función de sus signos, compensarse mutuamente y acumularse.

Error en la suma

La suma de los errores absolutos de los operandos proporciona en realidad un límite superior para el error absoluto total de la suma:

$$\Delta \leq \Delta_1 + \Delta_2 \tag{18}$$

Error relativo en la suma

El error relativo de la suma de dos números proximos del mismo signo no supera el error relativo más grande de los sumandos:

$$\delta \leq \text{máx}\left(\delta_1, \delta_2\right) \tag{19}$$

Error en la diferencia

Ahora consideremos la diferencia de dos números próximos con *el mismo signo*:

$$x = x_1 - x_2 \tag{20}$$

Error en la diferencia

Ahora consideremos la diferencia de dos números próximos con *el mismo signo*:

$$x = x_1 - x_2 \tag{20}$$

Debería ser evidente que las ecuaciones (17) y (20) cubren todas las posibles situaciones algebraicas.

Error en la diferencia

Al igual que en el caso de la suma, el caso más desfavorable es aquél en el que los errores absolutos se potencian entre sí, y, por lo tanto, la suma de los errores absolutos es, de nuevo, el límite superior para el error absoluto de la diferencia:

$$\Delta \leq \Delta_1 + \Delta_2$$

Error en la diferencia

Maximizando el error absoluto total en la definición de error relativo (δ), tenemos que

$$\delta \leq \frac{\Delta_1 + \Delta_2}{|x_1 - x_2|} = \frac{|x_1|\,\delta_1 + |x_2|\,\delta_2}{|x_1 - x_2|} \tag{21}$$

Error en la diferencia

Si los sumandos están cerca de su valor, la diferencia absoluta $|x_1 - x_2|$ es pequeño e, incluso para pequeños errores relativos individuales, δ_1 y δ_2, el error relativo de la diferencia, δ, puede volverse significativo.

Error en el producto

El error relativo de un producto de dos números próximos $x = x_1\, x_2$, no excede a la suma del error relativo de los factores δ_1 y δ_2:

$$\delta = \delta_1 + \delta_2 \tag{22}$$

Error en el cociente

Consideremos ahora el cociente

$$x = \frac{x_1}{x_2}$$

de dos números próximos no nulos.

Error en el cociente

Consideremos ahora el cociente

$$x = \frac{x_1}{x_2}$$

de dos números próximos no nulos.

Teniendo en cuenta, una vez más, que x_1 y x_2 se ven afectados por pequeños errores absolutos $\Delta_1 = |\Delta x_1|$ y $\Delta_2 = |\Delta x_2|$:

Error en el cociente

Obtenemos la relación límite:

$$\delta = \delta_1 + \delta_2 \tag{23}$$

que establece que el error relativo del cociente no excede los errores relativos acumulados del dividendo y divisor.

Funciones de Bessel

La acumulación de errores por redondeo a menudo limita la capacidad de un programa para calcular con precisión.

Funciones de Bessel

La acumulación de errores por redondeo a menudo limita la capacidad de un programa para calcular con precisión.

Revisaremos como ejercicio, la manera de calcular las funciones esféricas de Bessel $j_\ell(x)$ y de Neumann $n_\ell(x)$.

Funciones de Bessel

Estas funciones son respectivamente, las soluciones regulares/irregulares (no singulares / singulares en el origen) de la ecuación diferencial

$$x^2\, f''(x) + 2\, x\, f'(x) + \left[x^2 - \ell(\ell+1)\right] f(x) = 0 \quad (24)$$

Funciones de Bessel y esféricas de Bessel

Las funciones esféricas de Bessel se relacionan con las funciones de Bessel de primera clase por

$$j_\ell(x) = \sqrt{\frac{\pi}{2\,x}}\, J_{n+\frac{1}{2}}(x)$$

Funciones de Bessel y esféricas de Bessel

Se encuentran en diversos problemas de la física, como la expansión de una onda plana en ondas esféricas parciales

$$e^{i\,\mathbf{k}\cdot\mathbf{r}} = \sum_{\ell=0}^{\infty} i^{\ell}\,(2\,\ell+1)\,j_{\ell}(k\,r)\,P_{\ell}(\cos\theta) \qquad (25)$$

Funciones esféricas de Bessel de orden n

A continuación se muestra una gráfica con las funciones esféricas de Bessel de orden $n = 0, \ldots, 5$:

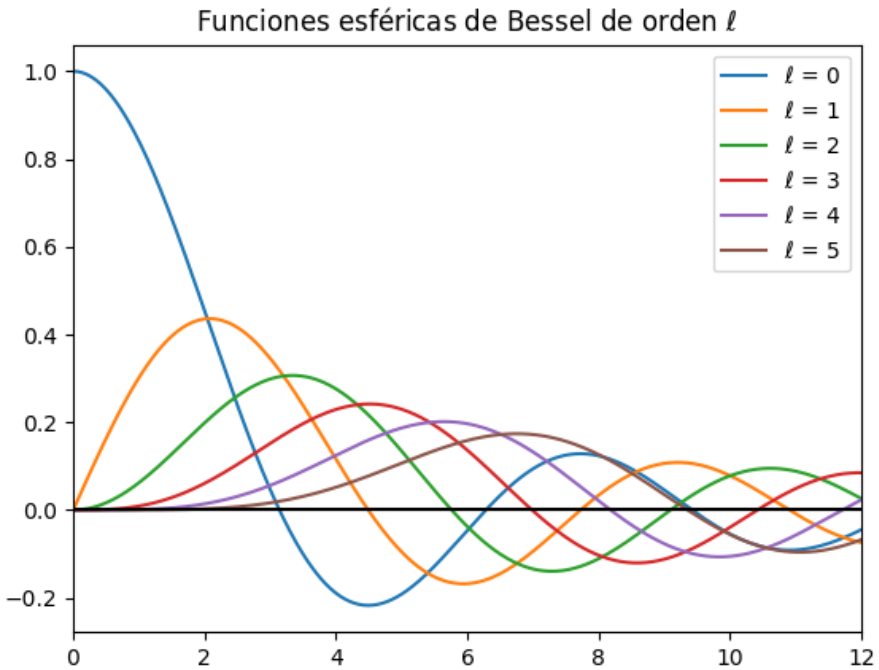


Figura: La gráfica se generó usando la librería de funciones especiales de `python`.

Valores para $\ell = 0, 1$

Los dos primero valores para ℓ son

$$j_0(x) = +\frac{\sin x}{x}, \qquad j_1(x) = +\frac{\sin x}{x^2} - \frac{\cos x}{x} \tag{26}$$

$$n_0(x) = -\frac{\cos x}{x}, \qquad n_1(x) = -\frac{\cos x}{x^2} - \frac{\sin x}{x} \tag{27}$$

Relaciones de recursión

La manera clásica de calcular las funciones de Bessel de orden j: $j_\ell(x)$ es sumar su serie de potencias para valores pequeños de $\frac{x}{\ell}$ y sumando su expansión asintótica para valores de x grandes.

Relaciones de recurrencia

El enfoque que usaremos, se basa en *las relaciones de recurrencia*:

$$j_{\ell+1}(x) = \frac{2\,\ell+1}{x}\, j_\ell(x) - j_{\ell-1}(x) \quad \text{hacia arriba} \tag{28}$$

$$j_{\ell-1}(x) = \frac{2\,\ell+1}{x}\, j_\ell(x) - j_{\ell+1}(x) \quad \text{hacia abajo} \tag{29}$$

Relaciones de recurrencia

Las ecuaciones (28) y (29) son la misma relación:

1. Una escrita para la recurrencia hacia adelante que parte de valores pequeños a valores grandes de ℓ.

Relaciones de recurrencia

Las ecuaciones (28) y (29) son la misma relación:

1. Una escrita para la recurrencia hacia adelante que parte de valores pequeños a valores grandes de ℓ.
2. La otra relación de recurrencia es descendente para valores de ℓ pequeños.

Relaciones de recurrencia

Las ecuaciones (28) y (29) son la misma relación:

1. Una escrita para la recurrencia hacia adelante que parte de valores pequeños a valores grandes de ℓ.
2. La otra relación de recurrencia es descendente para valores de ℓ pequeños.

Relaciones de recurrencia

Las ecuaciones (28) y (29) son la misma relación:

1. Una escrita para la recurrencia hacia adelante que parte de valores pequeños a valores grandes de ℓ.
2. La otra relación de recurrencia es descendente para valores de ℓ pequeños.

Con unas cuantas sumas y multiplicaciones, las relaciones de recurrencia permiten el cálculo rápido y sencillo de todo el conjunto de valores de j_ℓ para x fijo y todo ℓ.

Relaciones de recurrencia

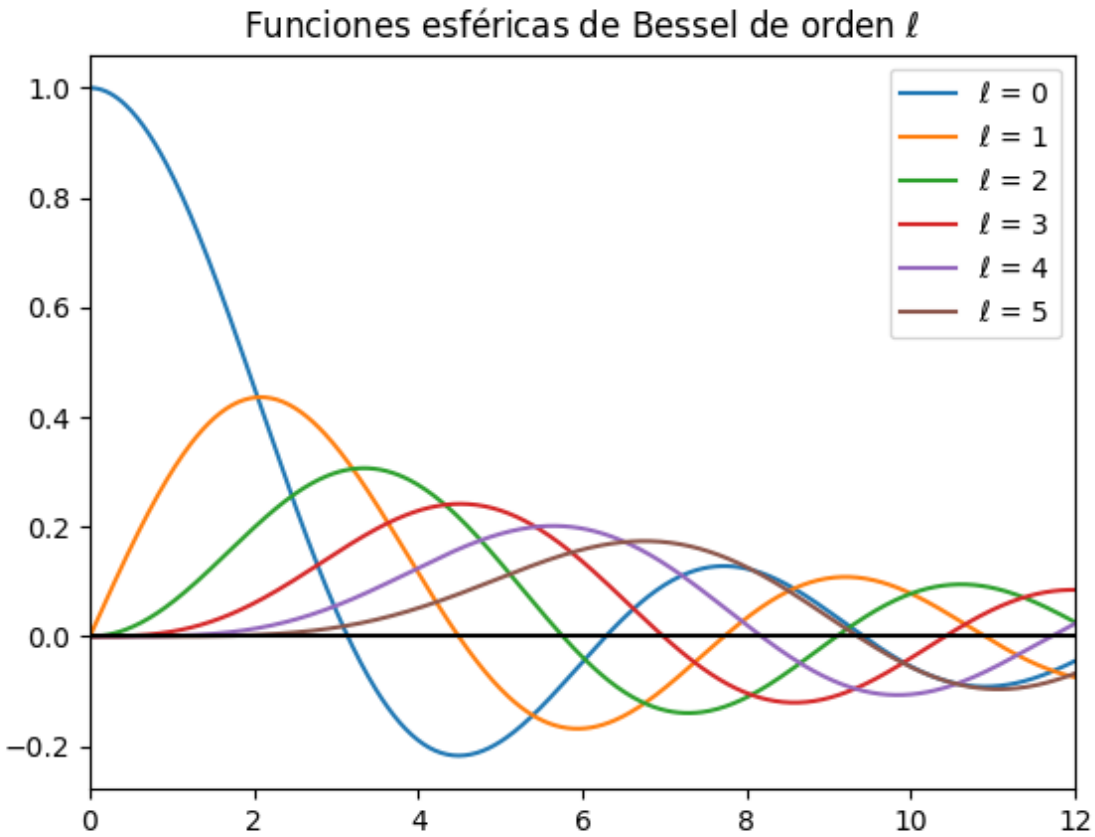
Para la relación de recurrencia hacia adelante de ℓ para un x fijo, comenzamos con las formas conocidas para j_0 y j_1 (26) y usamos (28).

Relaciones de recurrencia

Como veremos, esta recurrencia hacia adelante parece funcionar al principio, pero luego falla.

La razón de la falla se puede ver en las gráficas de $j_\ell(x)$ y $n_\ell(x)$ como función de x.

Relaciones de recurrencia



Relaciones de recurrencia

Si iniciamos con $x \simeq 2$ y $\ell = 0$, veremos que a medida que la recurrencia hacia adelante para j_ℓ con valores mayores de ℓ con (28), esencialmente tomamos la diferencia de dos funciones “grandes” para producir un valor “pequeño” para j_ℓ.

Relaciones de recurrencia

Este proceso sufre de cancelación sustractiva y siempre reduce la precisión.

A medida que continuamos con la recurrencia, tomamos la diferencia de dos funciones pequeñas con errores grandes y producimos una función más pequeña con un error aún mayor. Después de cierto tiempo, nos quedamos con sólo el error de redondeo (basura).

Estimando el error

Para ser más específicos, llamemos $j_\ell^{(c)}$ al valor numérico que calculamos como nua aproximación para $j_\ell(x)$.

Por lo que si comenzamos con un valor neto j_ℓ, después de un corto tiempo la falta de precisión de la computadora se mezcla efectivamente en un poco de n_ℓ:

$$j_\ell^{(c)} = j_\ell(x) + \varepsilon\, n_\ell(x) \tag{30}$$

Estimando el error

Esto no lo podemos evitar, porque tanto j_ℓ como n_ℓ satisfacen la misma ecuación diferencial y, por esa razón, la misma relación de recurrencia.

La mezcla de n_ℓ se convierte en un problema cuando el valor numérico de $n_\ell(x)$ es mucho mayor que el de $j_\ell(x)$ porque incluso una cantidad minúscula de un número muy grande puede ser grande.

Estimando el error

Por el contrario, si usamos la relación de recurrencia hacia adelante (28) para generar la función esférica de Neumann n_ℓ, no habrá ningún problema porque estamos combinando funciones pequeñas para producir más grandes, ya que es un proceso que no contiene cancelación sustractiva.

Solución al problema

La solución simple a este problema es usar (29) para la recursión descendente de los valores de j_ℓ que comienzan en un valor grande $\ell = L$.

Esto evita la cancelación sustractiva tomando valores pequeños de $j_{\ell+1}(x)$ y $j_\ell(x)$ y generando por la suma un valor mayor en $j_{\ell-1}(x)$.

Solución al problema

Mientras que el error todavía puede mantenerse como una función de Neumann, la magnitud real del error disminuirá rápidamente conforme la recurrencia hacia atrás use valores pequeños de ℓ.

Solución al problema

De hecho, si empezamos iterando hacia atrás con valores arbitrarios para $j_{L+1}^{(x)}$ y $j_L^{(c)}$, después de un corto tiempo llegaremos a la correcta relación de ℓ para este valor de x.

Solución al problema

Aunque el valor numérico de j_0^c así obtenido no será correcto porque depende de los valores arbitrarios asumidos para j_{L+1} y $j_L^{(c)}$, los valores relativos serán precisos.

Solución al problema

Los valores absolutos se fijan a partir del valor conocido (26), $j_0(x) = \sin x/x$.

Debido a que la relación de recurrencia es una relación lineal entre los valores de j_ℓ, sólo necesitamos normalizar todos los valores calculados mediante

Solución al problema

$$j_\ell^{\text{normalizada}} = j_\ell^{(c)}(x) \times \frac{j_0^{\text{analitica}}}{j_0^{(c)}} \tag{31}$$

En consecuencia, después de haber terminado la recurrencia hacia abajo, se obtendrá la respuesta final normalizando todos los valores de $j_\ell^{(c)}$ basados en el valor conocido para j_0.

Propuesta de código

Veamos la manera de implementar un código con `python`para calcular el valor de la función esférica de Bessel con la recurrencia hacia atrás.

Propuesta de código I

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import scipy.special as spl
4
5  Xmax = 40.
6  Xmin = 0.25
7  paso = 0.1
8  orden = 10
9  inicio = 50
10
11 def abajo (x, n, m):
12       j = np.zeros( (inicio + 2), float)
13       j[m+1] = j[m] = 1.
14
15       for k in range(m, 0, -1):
```

Propuesta de código II

```
16            j[k-1] = ((2.*k + 1.)/x)*j[k] - j[k+
    1]
17
18        escala = (np.sin(x)/x)/j[0]
19
20        return j[n] * escala
21
22 valoresy = []
23
24 valoresx = []
25
26 for x in np.arange(Xmin, Xmax, paso):
27     valoresy.append(abajo(x, orden, inicio))
28     valoresx.append(x)
29
30 plt.plot(valoresx, valoresy, color='r')
```

Propuesta de código III

```
31 plt.axhline(y=0, ls='dashed', color = 'k')
32 plt.xlim([0.25, 40])
33 plt.show()
```

Función aproximada

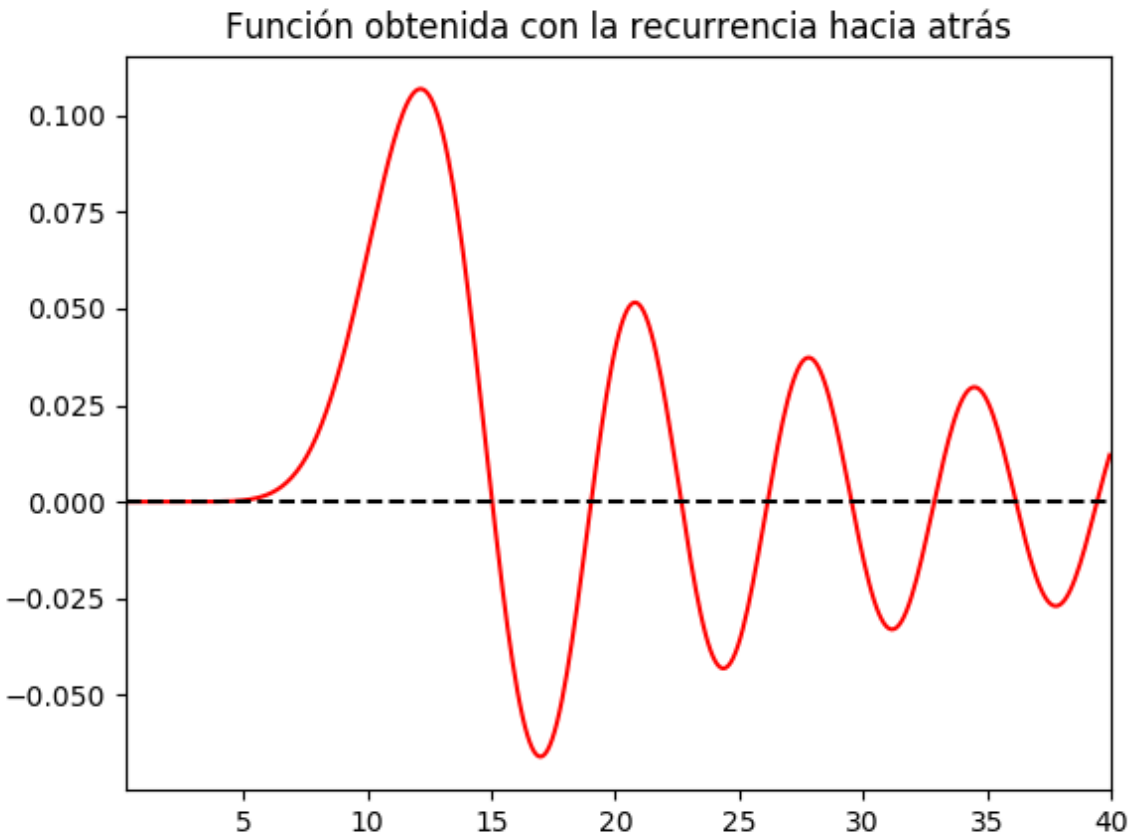


Figura: Gráfica obtenida con el algoritmo propuesto