

Clase 2

Análisis del Error

El concepto de error es fundamental en el cálculo numérico.

En cada problema, es muy importante realizar un seguimiento de los errores cometidos a fin de poder estimar el grado de aproximación de la solución obtenida.

Hay que estudiar los diversos tipos de error que afectan sus soluciones:

- errores que provienen del almacenamiento de los datos llamados errores de redondeo. Esto se debe a que la memoria del computador no tiene infinitos decimales, se debe usar un sistema binario que trunca los valores calculados.
- errores que provienen de los métodos o algoritmos numéricos usados. Estos provienen de la ejecución de las diferentes técnicas de los métodos numéricos.

2.1 Errores de Redondeo

Los computadores (computadoras, ordenadores) trabajan con un número **finito de dígitos**.

Almacenar números que necesitan un número infinito de dígitos no es posible.

Los números tales como $\frac{1}{3} = 0.333\dots$ o $\sqrt{2} = 1.4142136\dots$ no pueden ser almacenados ya que se necesitaría un número infinito de cifras.

El error que cometemos al almacenar un número en el ordenador se denomina error de redondeo.

Ejemplo 1:

Supongamos que nuestro computador trabaja con seis (6) cifras decimales. Cuando trabajamos con $\sqrt{2}$, tendremos un error de redondeo acotado por 0.5×10^{-6} , esto es

$$\left| \sqrt{2} - 1.414214 \right| \approx 4.4 \times 10^{-7} \leq 0.5 \times 10^{-6}$$

Antes de proseguir con el análisis del error, hagamos una breve descripción de la manera en que los computadores almacenan los valores reales con el fin de entender con más profundidad los **errores de redondeo** y los errores de provienen de las **operaciones realizadas**.

2.2 Números Binarios

La manera de almacenar los números o las aproximaciones de los mismos como se ha comentado antes, es mediante su representación binaria ya que los computadores trabajan en sistema binario.

Dado un número x , la representación binaria de x será:

$$x = \sum_{i=0}^s a_i \cdot 2^i + \sum_{i=1}^t b_i \cdot 2^{-i}$$

donde $a_i, b_i \in \{0, 1\}$

Ejemplo 2:

Veamos cómo escribir el número 10.2345 en binario

Primero escribimos su parte entera: $10 = 2 + 2^3$, y escribimos $10 = 1010_{(2)}$.

El número 1010 nos indica que 10 puede escribirse como $10 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. Para obtener los números 1010, hemos de realizar

divisiones sucesivas por 2 y considerar los restos:

$$10 = 5 \cdot 2 + 0,$$

$$5 = 2 \cdot 2 + 1,$$

$$2 = 2 \cdot 1 + 0$$

Ahora para la parte decimal debemos multiplicar sucesivamente por 2 y considerar la parte entera

$$0.2345 \cdot 2 = 0.469, \Rightarrow b_1 = 0,$$

$$(0.469 - b_1) \cdot 2 = 0.938, \Rightarrow b_2 = 0,$$

$$(0.938 - b_2) \cdot 2 = 1.876, \Rightarrow b_3 = 1,$$

$$(1.876 - b_3) \cdot 2 = 1.752, \Rightarrow b_4 = 1,$$

...

esto significa que $0.2345 = 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + \dots$ y escribimos $0.2345 = 0.0011\dots_{(2)}$, donde \dots significa que hay más cifras y que dicho número puede tener infinitas cifras en base 2.

Por tanto, el número 10.2345 en binario será de la forma $10.2345 = 1010.0011\dots_{(2)}$

Ejemplo 3:

Construir un código en Python que dado un número en base decimal lo convierta en base dos (binaria) y viceversa.

Programa de Google Colab

Este programa se ha desarrollado usando Google Colab. Para ver el programa haz clic en el logo.



```

31.5406808853149 => 11111.10001010011010100001(2)
[3] Parte Entera: 1027
    Parte Decimal: 0
    1027 => 1000000011.(2)

[4] numDec = 31.54068088531494
    result = floatToBin(numDec)
    print(numDec, " => ", result)

Parte Entera: 31
Parte Decimal: 0.5406808853149414
31.54068088531494 => 11111.10001010011010100001(2)

Se define la función binToFloat:
Descripción: Dado numeroBin un número binario, devuelve su representación en decimal
Parámetro: numeroBin - Número Binario
  
```

Figura 2.1: Ejecución del código para transformar el número 31.54068088531494 a binario

2.2.1 Ejercicios

1. Convertir a número binario los siguientes números decimales. Compare su respuesta con el algoritmo desarrollado en Python

- | | |
|--------------|-------------|
| a) 5125.8908 | e) 1.032022 |
| b) 20.081989 | f) 2.031970 |
| c) 14.011996 | g) 7.031963 |
| d) 1.02020 | h) 1.081993 |

2.3 Representación Binaria

Veamos cómo se almacena un número real en el computador en **formato binario** usando **64 bits**.

Sea x un número real que suponemos en formato **binario**.

Escribimos x de la siguiente forma:

$$x = (-1)^s 1.f \cdot 2^{c-1023}$$

2.3.1 Signo, mantisa y exponente

○ s indica el signo del número, indicando $s = 0$ para los números positivos y $s = 1$, para los negativos.

○ $1.f$ es lo que se llama mantisa donde f es una secuencia de ceros y unos,

$$f = f_1 f_2 f_3 \dots f_n, f_i \in \{0, 1\}$$

○ $c \geq 1$ indica el exponente del número donde se le resta 1023 para poder representar números muy pequeños en valor absoluto. Escribimos $c - 1023$ en binario como $c - 1023 = e_1 e_2 e_3 \dots e_m$ con $e_i \in \{0, 1\}$

Representamos x por tres cajas: el signo s , la mantisa f y el exponente $c - 1023$ en binario

$$|s| f_1 f_2 f_3 \dots f_n |e_1 e_2 e_3 \dots e_m|$$

Observación

○ El estándar **IEEE 754**

El estándar **IEEE 754** define cómo se almacenan y representan los números en coma flotante en sistemas informáticos. Este estándar incluye diferentes formatos, como el de precisión sencilla (32 bits) y el de precisión doble (64 bits).

En el formato de precisión doble (64 bits), el exponente tiene 11 bits. Esto significa que se pueden representar $2^{11} = 2048$ valores diferentes para el exponente.

El sesgo se elige de manera que el rango de exponentes sea simétrico alrededor de cero. En este caso, el sesgo es 1023. Esto permite representar exponentes desde -1022 hasta $+1023$.

el estándar **IEEE 754** establece que el sesgo debe ser una potencia de 2 menos 1, es decir que $2^{11-1} = 1024$. Esto asegura ciertas propiedades matemáticas y facilita algunas operaciones.

Ejemplo 4:

Sea $x = 31.54068088531494$. Vamos a escribirlo en binario usando el formato anterior

Usamos el código desarrollado para convertir el número a binario, entonces tenemos lo siguiente:

$$31.54068088531494_{(10)} = 11111.10001010011010100001$$

ahora debemos correr el punto decimal cuatro posiciones a la derecha y compensar ese desplazamiento multiplicando por 2^4 (se multiplica por 2^4) porque el número ya ha sido convertido a binario, si estuviéramos trababajando en notación científica en base decimal, se multiplica por 10^4 pero este ya esta en base binaria.

$$31.54068088531494_{(10)} = +1.111110001010011010100001 \cdot 2^4$$

así $s = 0$, $f = 111110001010011010100001$ y

$$\begin{aligned} c &= 100_{(2)} + 1023 \\ &= 100_{(2)} + 1111111111_{(2)} \\ &= 10000000011_{(2)} \end{aligned}$$

entonces escribimos

$$31.54068088531494_{(10)} = |0|111110001010011010100001|10000000011|$$

Ejercicio 1:

Escribir un programa en **Python** que dado un número decimal genere su representación binaria en el formato de 64 bits

Programa de Google Colab

Este programa se ha desarrollado usando Google Colab. Para ver el programa haz clic en el logo.



2.3.2 Ejercicios

1. Representar en formato de 64 bits los siguientes números, comparar los resultados con el algoritmo desarrollado en **Python**.
 - a) 1.235711131719
 - b) 2.468101214161
 - c) 3.691215182124
 - d) 2.481632641282
 - e) 9.765432123456

2.4 Rango de Valores

En general, para representar un número en 64 bits en el **IEEE 754** se guardan:

- ☐ 1 bit para su signo,
- ☐ 52 bits para su mantisa y
- ☐ 11 bits para su exponente

Rango de Valores para la Mantisa

Como solo tenemos 52 bits para la mantisa, esto significa que solo podemos representar números en base 10 con 16 cifras decimales como máximo ya que

$$2^{52} = 4.50359962737 \times 10^{15}$$

¿Por qué 16 cifras significativas?

El número $4.50359962737 \times 10^{15}$ se puede representar como $0.450359962737 \times 10^{16}$.

Rango de Valores para el Exponente

El exponente estaría entre los valores -1022 y 1023 . ¿Por qué?. Dado que el exponente tiene 11 bits el número máximo que puede tomar el exponente es $2^{11} = 2048$. Según el estándar IEEE 754 los valores del exponente deben representar simétricamente que van desde -1022 hasta 1023 . Ahora como 11111111111 es un número positivo se le debe restar 1023 para poder simetrizar el exponente.

Ejemplo 5:

Realizar la conversión de representación binaria a número decimal

$$|1|101101110011|1111|$$

El signo es negativo, por tanto $x < 0$.

La mantisa será:

$$\begin{aligned} 1.f &= 1 + \frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^8} + \frac{1}{2^{11}} + \frac{1}{2^{12}} \\ &= 1.71557617188 \end{aligned}$$

El exponente c vale $1111_{(2)} - 1023 = 1 + 2 + 2^2 + 2^3 - 1023 = -1008$ el número entonces es

$$-1.71557617188 \times 2^{-1008} = -6.25423760148 \times 10^{-304}$$

2.4.1 Número más pequeño

Con la representación anterior, el número más pequeño x_{min} en valor absoluto que se podría representar sería:

$$|0|000 \dots 0_{52}|1|$$

cuyo valor es:

$$\begin{aligned} x_{min} &= 1 \cdot 2^{1-1023} = 2^{-1022} \\ &= 2.2250738585 \times 10^{-308} \end{aligned}$$

¿De dónde sale el 1?, del formato de la mantisa que se escribe como $1.f$

2.4.2 Número más grande

Con la representación binaria de 64 bits el número más grande $x_{máx}$ que se puede representar es

$$|0|111 \dots 1_{52}|1111111111|$$

cuyo valor es:

$$\begin{aligned} x_{máx} &= \left(1 + \sum_{i=1}^{52} \frac{1}{2^i}\right) \cdot 2^{2047-1023} \\ &= 1.9999999999999998 \cdot 2^{1024} \\ &= 3.595386269724631416290548475 \times 10^{308} \end{aligned}$$

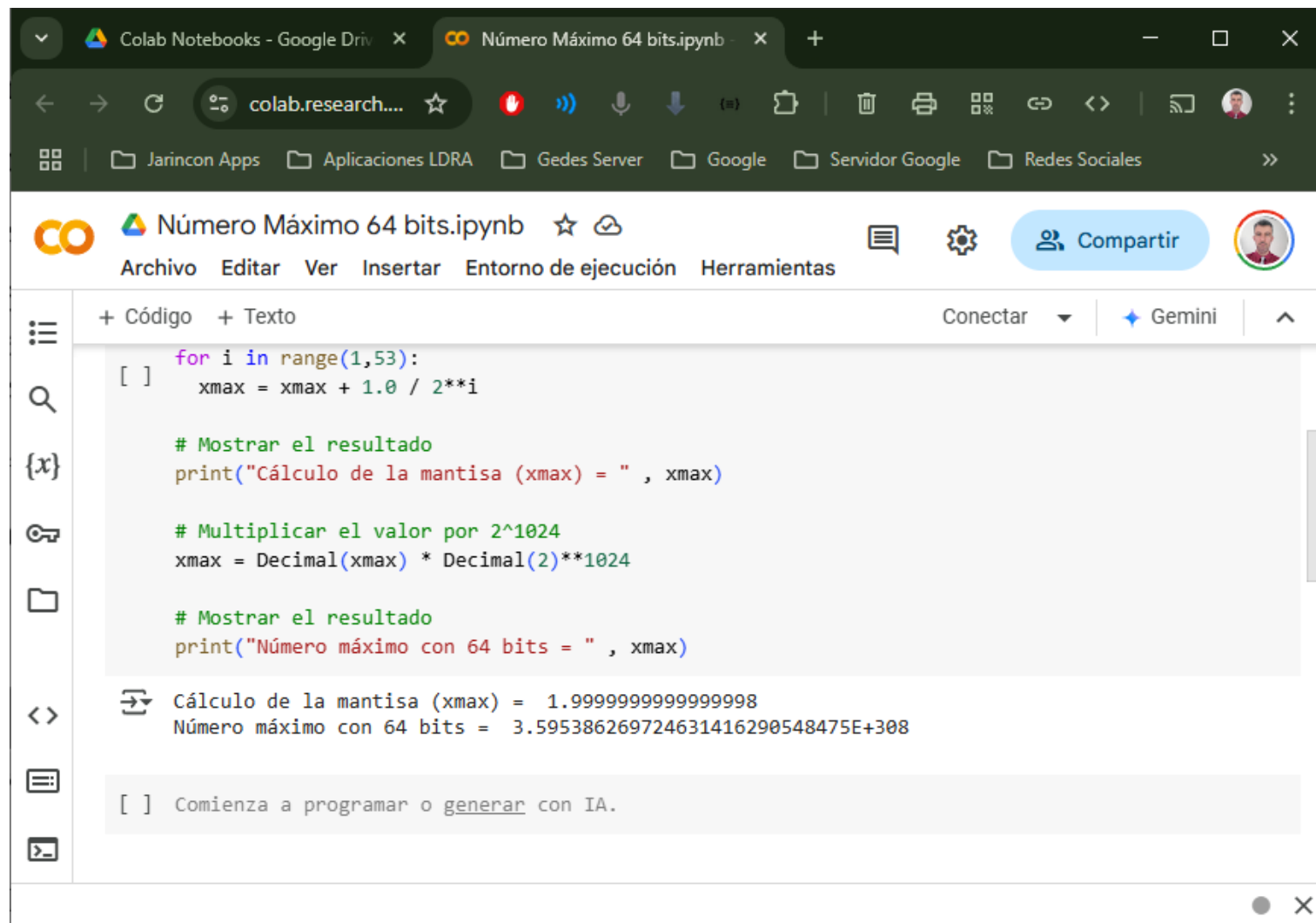
Ejercicio 2:

Construir un programa en **Python** que permita calcular el número más grande que se puede representar con 64 bits

Programa de Google Colab

Este programa se ha desarrollado usando Google Colab. Para ver el programa haz clic en el logo.





```
+ Código + Texto
[ ] for i in range(1,53):
    xmax = xmax + 1.0 / 2**i

# Mostrar el resultado
print("Cálculo de la mantisa (xmax) = " , xmax)

# Multiplicar el valor por 2^1024
xmax = Decimal(xmax) * Decimal(2)**1024

# Mostrar el resultado
print("Número máximo con 64 bits = " , xmax)

Cálculo de la mantisa (xmax) = 1.9999999999999998
Número máximo con 64 bits = 3.595386269724631416290548475E+308

[ ] Comienza a programar o generar con IA.
```

Figura 2.2: Ejecución del código, calculando el número más grande que se puede representar con 64 bits