Teoría de Errores

José Luis Ramírez

Octubre 2025

Contents

Motivación

- La gran mayoría de los modelos matemáticos que describen procesos físicos no pueden resolverse analíticamente.
- En una situación práctica, un problema matemático deriva de un fenómeno físico sobre el cual se han hecho algunas suposiciones para simplificarlo y poderlo representar matemáticamente.
- Una vez formulado el problema, deben diseñarse métodos numéricos para resolver el problema. La selección o construcción de los algoritmos apropiados cae propiamente dentro del terreno del Análisis Numérico.

Teoría de Errores y Aproximación

El análisis numérico proporciona métodos computacionales para el estudio y solución de problemas matemáticos. Debido a que muchos cálculos son realizados en computadores digitales, es conveniente la discusión para la implementación de los métodos numéricos como programas de computador.

- Desafortunadamente los resultados son afectados por el uso de la **Aritmética de Precisión Finita**.
- Esperamos tener siempre expresiones verdaderas como $2 + 2 = 4$, $3^2 = 9$, $(\sqrt{5})^2 = 5$, pero en la aritmética de precisión finita $\sqrt{5}$ no tiene un solo número fijo y finito, que lo represente.

Los resultados numéricos están influenciados por muchos tipos de errores, los cuales pueden ser catalogados a grandes rasgos en tres tipos básicos:

- **Errores inherentes** que existen en los valores de los datos de entrada, ya sean causados por incertidumbre o por la naturaleza necesariamente aproximada de la representación.
- **Errores de discretización** (llamados también de truncamiento) que surgen al reemplazar procesos límites por su resultado antes de alcanzar tal límite.
- **Errores de redondeo** que se originan al utilizar una aritmética que involucra números con un número finito de dígitos.

Errores Absolutos y Relativos.

Sea x el valor exacto de un número real y $\tilde{x}$ el valor aproximado. Contemplando todos los posibles errores, la relación entre el resultado exacto y el aproximado es:

$$x = \tilde{x} + E$$

Se define el error absoluto y se denota E_a como la diferencia $x\tilde{x}$, y se expresa siempre en valor absoluto.

$$E_a = |x - \tilde{x}|$$

Al cociente entre el error absoluto E_a y el valor real x se le denomina error relativo y se denota por E_r. Se expresa también en valor absoluto, es decir:

$$E_r = \frac{|E_a|}{|x|} = \frac{|x - \tilde{x}|}{|x|}$$

Ejemplo 1

- Calculemos los errores absolutos y relativos para el valor $x = 1234.5678$, con aproximaciones a 4 dígitos $x_1 = 1234$ y $x_2 = 1235$
  - $E_a = |x - x_1| = |1234.5678 - 1234| = 0.5678$, $E_r = \frac{|E_a|}{|x|} = \frac{0.5678}{1234.5678} \approx 5 \times 10^{-4}$.
  - $E_a = |x - x_2| = |1234.5678 - 1235| = 0.4322$, $E_r = \frac{|E_a|}{|x|} = \frac{0.4322}{1234.5678} \approx 4 \times 10^{-4}$.
- Calculemos los errores absolutos y relativos para el valor $x = 0.00004599881234$, con aproximaciones a 4 dígitos $x_1 = -0.00004599$ y $x_2 = -0.00004600$
  - $E_a = |x - x_1| = |0.00004599881234 + 0.00004599| = 8.81234 \times 10^9$, $E_r = \frac{|E_a|}{|x|} = \frac{8.81234Ö10^9}{0.00004599881234} \approx 2 \times 10^{-4}$.
  - $E_a = |x - x_2| = |0.00004599881234 + 0.00004600| = 1.18766 \times 10^9$, $E_r = \frac{|E_a|}{|x|} = \frac{1.18766 \times 10^9}{0.00004599881234} \approx 2.5819 \times 10^{-4}$.

Observaciones:

- Mirando los ejemplos anteriores, vemos que los **errores absolutos** dependen de las magnitudes de los valores x: en el primer ejemplo los errores absolutos son de orden de 10^1; en cambio, en el segundo, son del orden de 10^9.
- En cambio, los **errores relativos** no se ven afectados por dichas magnitudes. Por dicho motivo, si queremos estudiar los errores sin tener en cuenta el orden de los valores x, hay que usar los **errores relativos**. Vemos que en los dos ejemplos los errores relativos son del orden de 10^4. ya que recordemos que las aproxi9maciones son a 4 **cifras decimales significativas**.

Cifras Significativas.

Se dice que el número $\tilde{x}$ aproxima al número x con t dígitos (o cifras) significativas, si t es el número entero más grande no negativo para el cual:

$$E_r < 0.5 \times 10^{-t} \Rightarrow \frac{|x - \tilde{x}|}{|x|} < 0.5 \times 10^{-t}$$

Ejemplo 2

Sea $\tilde{x} = 3.1416$ una aproximación al valor π, y $x = 3.1415927$ una mejor aproximación.

```
import math
import string
from decimal import Decimal

def redondear(N,signif):
  """
      Dado un número real N y un número determinado de cifras significativas signif,
      nos da la aproximación de este número con signif cifras significativas.

        Parámetros:
        * N: número decimal
        * signif: número de digitos representativos

        Valor de retorno
        * número redondeado a signif número de cifras significativas

   """
  if int(N)==0:
    return float(round(Decimal(str(N)),signif))
  else:
    return float(round(Decimal(str(N)),signif-1-int(math.log(abs(N),10))))

def Err(num,dig):
    """
      Dado un número real y un número determinado de cifras significativas,
      nos da el error absoluto y relativo de dicho número.

        Parámetros:
        * num: número decimal
        * dig: número de cifras significativas

        Valor de retorno
        * Error absoluto
        * Error Relativo

    """
    err_abs=abs(num-redondear(num,dig))
    err_rel=err_abs/abs(redondear(num,dig))

    print("El error absoluto es {}".format(err_abs))
    print("El error relativo es {}".format(err_rel))

    return (err_abs, err_rel)
```

```
Err(3.1415927,5)
```

```
## El error absoluto es 7.300000000043383e-06
## El error relativo es 2.3236567354352506e-06
## (7.300000000043383e-06, 2.3236567354352506e-06)
```

$$E_a = |x - \tilde{x}| = |3.1415927 - 3.1416| = 0.0000073$$
$$E_r = \frac{|E_a|}{|x|} = \frac{0.0000073}{3.1415927} \approx 0.232 \times 10^{-5} < 0.5 \times 10^{-5}$$

Luego, $\tilde{x}$ aproxima a π con 5 cifras significativas.

Aproximación de Números.

Sea $x = a_n \dots a_0.b_1b_2 \dots \in \mathbb{R}$ (En cualquier base), para aproximar hasta el t-ésimo decimal:

- Por **truncamiento**

$$\tilde{x}_{trunc} = a_n \dots a_0.b_1b_2 \dots b_t$$

- Por **redondeo correcto**

$$\tilde{x}_{redon} = \begin{cases} a_n \dots a_0.b_1b_2 \dots b_t & \text{si } b_{t+1} < 5 \\ a_n \dots a_0.b_1b_2 \dots b_t + \beta^{-t} & \text{si } b_{t+1} \geq 5 \end{cases}$$

Forma Normalizada de Números en Punto Flotante

Un número en punto flotante en base β con precisión t es un número que puede representarse en la forma:

$$x = \sigma(0.d_1d_2 \dots d_t)_\beta \times \beta^e$$

donde:

- σ es el signo del número ($+1$ o -1),
- $d_1d_2 \dots d_t$ son los dígitos en base β (con $0 \leq d_i < \beta$ y $d_1 \neq 0$ para la forma normalizada),
- e es el exponente entero, y es tal que $L \geq e \geq U$ para ciertos enteros L y U.
- t es la precisión (número de dígitos significativos).

Una de las características de todo conjunto de punto flotante F es que es finito y tiene:

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

números diferentes (incluyendo el cero), y donde los distintos de cero están en forma normalizada.

Ejemplo 3

Sea el conjunto de punto flotante F con parámetros $\beta = 2$(Binario), $t = 3$, $L = -2$, $U = 2$. Tal conjunto F tiene

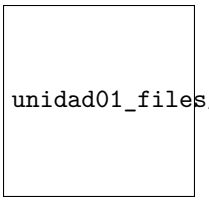
$$2(2 - 1)2^{3-1}(2 - (-2) + 1) + 1 = 41$$

números diferentes (incluyendo el cero). La lista completa de números en F es:

```python
import numpy as np
import matplotlib.pyplot as plt
def generar_punto_flotante(beta, t, L, U):
    numeros = set()
    for signo in [1, -1]:
        for exponente in range(L, U + 1):
            for d1 in range(1, beta):
                for d2 in range(beta):
                    for d3 in range(beta):
                        numero = signo * (d1 * beta**(-1) + d2 * beta**(-2) + d3 * beta**(-3)) * (beta
                        numeros.add(round(numero, 10))  # Redondear para evitar problemas de precisión
    numeros.add(0)  # Incluir el cero
    return sorted(numeros)

# Parámetros del sistema de punto flotante
beta = 2  # Base (binaria)
t = 3     # Dígitos de la mantisa
L = -2    # Exponente mínimo
U = 2     # Exponente máximo
punto_flotante = generar_punto_flotante(beta, t, L, U)
print("Números en el conjunto de punto flotante F:{}".format(punto_flotante))
```

```
## Números en el conjunto de punto flotante F:[-3.5, -3.0, -2.5, -2.0, -1.75, -1.5, -1.25, -1.0, -0.875
```

```python
# --- Visualización de la Recta Real y los Puntos ---
plt.figure(figsize=(12, 2)) # Ajusta el tamaño de la figura (ancho, alto)

# Dibuja la recta real (una línea horizontal)
plt.axhline(0, color='gray', linestyle='-', linewidth=1)

# Dibuja los puntos generados sobre la recta 'o' para los marcadores, 'b' para el color azul, markersiz
plt.plot(punto_flotante, np.zeros_like(punto_flotante), 'ob', markersize=5)
plt.plot(0, 0, 'go', markersize=7, label='Cero') # 'go' = green circle

# Etiquetas y Título
plt.title(f'Representación de Números de Punto Flotante (beta={beta}, t={t}, L={L}, U={U})')
plt.xlabel('Recta Real')

# Ajustes de los ejes para que los puntos sean más visibles
min_val = min(punto_flotante) - 0.5
max_val = max(punto_flotante) + 0.5
x_limits = plt.xlim(min_val, max_val) # Extiende ligeramente los límites del eje X
yticks = plt.yticks([]) # Oculta las marcas del eje Y, ya que solo es una recta horizontal

# Muestra una leyenda si es necesario
plt.legend()
plt.grid(True, linestyle=':', alpha=0.7)
plt.show()
```



- Hay un rango limitado para representar cantidades
  - Hay números grandes positivos y negativos que no pueden ser representados (overflow)
  - No pueden representarse números muy pequeños (underflow)
- Hay sólo un número finito de cantidades que puede ser representado dentro de un rango
  - El grado de precisión es limitado
  - Para aquellos que no pueden ser representados exactamente, la aproximación real se puede lograr: truncando o redondeando.
- El intervalo entre números aumenta tanto como los números crecen en magnitud
  - El error cuantificable más grande ocurrirá para aquellos valores que caigan justo debajo del límite superior de la primera serie de intervalos igualmente espaciados.

Vamos a ver cómo se almacena un número real en el ordenador en **formato binario** usando **64 bits**.

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

Escribimos x de la forma siguiente:

$$x = (1)^s 1.f \times 2^{c1023},$$

donde

- s indica el signo del número, indicado $s = 0$ para los números positivos y $s = 1$, para los negativos.
- $1.f$ es lo que se llama la mantisa donde f es una secuencia de ceros y unos, es decir,

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

- $c \geq 1$ indica el exponente del número donde se le resta 1023 para poder representar números muy pequeños en valor absoluto. Escribimos $c1023$ en binario como $c1023 = e_1 e_2 \dots e_m$, con $e_i \in \{0, 1\}$

Representamos x por tres cajas: el signo s, la mantisa f y el exponente $c1023$ en binario:

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

Ejemplo 4:

Representar el número $x = 31.53173828125$ en formato de punto flotante de 64 bits.

1. Determinar el signo s : Como $x > 0,$ entonces $s = 0$.
2. Convertir x a binario:
   Parte entera: $31_{10} = 11111_2$.
   Parte fraccionaria: $0.53173828125_{10} = 0.10001000001_2$.
   Por lo tanto, x en binario es aproximadamente 11111.10001000001_2.
3. Normalizar el número:
   $11111.10001000001_2. = 1.1111110001000001_2 \times 2^4$.
4. Determinar la mantisa f :
   $f = 1111110001000001$
5. Determinar el exponente $c - 1023$:
   $c = 4 + 1023 = 1027$.
   Convertir 1027 a binario: $1027_{10} = 10000000011_2$.
6. Representar el número en formato de 64 bits:
   $|0|1111110001000001|10000000011|$.

Como sólo tenemos 64 bits para representar el número, la cantidad de bits usados para su representación no puede superar 64

En caso que los superase, tendremos que considerar una aproximación del mismo.

En el ejemplo se usado el número siguiente de bits:

- signo: 1 bit,
- mantisa: 15 bits,
- exponente: 11 bits,

en total, 27 bits, por tanto, sí sería posible su representación exacta y no haría falta considerar una aproximación del mismo.

Ejemplo 5:

Hagamos la conversión contraria.

Imaginemos que nos dan el número siguiente:

$$|1|101101110011|1111|.$$

Vamos a ver a qué número x corresponde.

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

La mantisa será:

$$1.f = 1 + \frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^8} + \frac{1}{2^{11}} + \frac{1}{2^{12}} = 1.7155762.$$

El exponente c vale $1111_2 1023 = 1 + 2 + 2^2 + 2^3 1023 = 1008$.

El número será:

$$x = 1.7155762 \times 2^{1008} \approx 6.25424 \times 10^{304}.$$

- La combinación aritmética usual $+, , \times, \div$ de dos números de punto flotante no siempre produce un número de punto flotante.

- Supongamos que $fl(x), fl(y) \in F$. Veamos, como ejemplo, que la suma usual $fl(x) + fl(y)$ no necesariamente será un número en F.
  - Sea el conjunto F dado en el ejemplo: $fl(x) = 5/32 \in F$, $fl(y) = 48/32 \in F$, sin embargo $fl(x) + fl(y) = 5/32 + 48/32 = 53/32 \notin F$.
- Las operaciones aritméticas que realiza un computador no corresponden de forma exacta con las operaciones usuales. El estudio de lo que ocurre realmente es dificil de realiza y en todo caso depende de la máquina que se esté utilizand.o
- Denotando por $\oplus, \ominus, \otimes, \oslash$ las operaciones de suma, resta, multiplicación y división de la máquina. Se definen estas operaciones por:

$$
\begin{aligned}
x \oplus y &= fl(fl(x) + fl(y)) \\
x \ominus y &= fl(fl(x) - fl(y)) \\
x \otimes y &= fl(fl(x) \times fl(y)) \\
x \oslash y &= fl(fl(x)/fl(y))
\end{aligned}
$$

Unidad de Precisión o Redondeo

- En la representación en punto flotante con n dígitos en bas β y exponente e, el error relativo en la representación de un número real x, $x \neq 0$ es estimado por:

$$\left| \frac{x - fl(x)}{x} \right| \leq \mu$$

- donde μ se conoce como la unidad de precisión (o redondeo) de la máquina. El valor μ es una característica de la máquina, su sistema operativo y el modo de calcular (simple o doble precisión).
- Otra definición de $\mu \approx \epsilon$ (Epsilon de la máquina): ϵ es el número más pequeño positivo de la forma $\epsilon = 2^{-k}$ tal que:

$$1.0 + \epsilon \neq 1.0 \quad \text{(en la máquina)}$$

```
# Definición de una función para calcular el épsilon de la máquina
def calcular_epsilon_maquina():
    """
    Calcula el épsilon de la máquina (el número más pequeño tal que 1 + eps > 1).
    """
    epsilon = 1.0

    # Mientras 1.0 + epsilon sea reconocido como igual a 1.0,
    # dividimos epsilon a la mitad.
    while 1.0 + epsilon > 1.0:
        epsilon /= 2.0

    # La última división hace que 1.0 + epsilon ya no sea > 1.0.
    # Por lo tanto, el verdadero épsilon es el valor anterior (el doble).
    return epsilon * 2.0

# Ejecutar la función y mostrar el resultado
epsilon_calculado = calcular_epsilon_maquina()

print("---")
```

```
## ---
```

```
print(f"Épsilon de la máquina (calculado): {epsilon_calculado}")
```

```
## Épsilon de la máquina (calculado): 2.220446049250313e-16
```

```
print(f"En notación científica: {epsilon_calculado:.2e}")
```

```
## En notación científica: 2.22e-16
```

```
print("---")
```

```
## ---
```

```
# Comprobación de la definición:
# 1 + épsilon
print(f"Comprobación (1 + Épsilon): {1.0 + epsilon_calculado}")
```

```
## Comprobación (1 + Épsilon): 1.0000000000000002
```

```
# 1 + épsilon / 2 (debería ser igual a 1)
print(f"Comprobación (1 + Épsilon / 2): {1.0 + epsilon_calculado / 2.0}")
```

```
## Comprobación (1 + Épsilon / 2): 1.0
```

Condicionamiento y Estabillidad

- Diremos que un proceso numérico, o una operación, es inestable cuando pequeños errores en los datos de entrada, o errores de redondeo en alguna de las etapas el proceso, producen errores grandes en los datos de salida.
- Diremos que un proceso numérico, es estable cuando no es inestable.
- Un mismo algoritmo puede ser estable para algunos datos iniciales e inestable para otros. Entonces se dice que el algoritmo es condicionalmente estable.

Ejemplo 6

Nos planteamos calcular el valor $x_n = \frac{1}{3^n}$, donde $x_0 = 1$, $x_1 = \frac{1}{3}$, $x_2 = \frac{1}{9}$, y así sucesivamente.

Existe una fórmula recursiva para calcular x_n

$$x_{n+1} = Ax_n + \left(\frac{1-3A}{9}\right) x_{n-1}$$

donde A es una constante fija.

- $\therefore$ Si elegimos $x_1 = \frac{1}{3} \approx 0.3333 = \tilde{x}_1$
- $\tilde{x}_1 = x_1 + E_1$, donde E_1 es el error de redondeo en la representación de x_1.
- Calculamos x_2 usando la fórmula recursiva:
  - $\tilde{x}_2 = A\tilde{x}_1 + \left(\frac{1-3A}{9}\right) x_0$
  - $= Ax_1 + \left(\frac{1-3A}{9}\right) x_0 + AE_1$
  - $= x_2 + AE_1$
- Así, el error en el cálculo de x_2 es $E_2 = AE_1$.
- De forma similar, el error en el cálculo de x_3 es $E_3 = AE_2 = A^2E_1$.
- En general, el error en el cálculo de x_n es $E_n = A^{n-1}E_1$.
- Si $|A| > 1$, el error crece exponencialmente con n y el proceso es inestable.
- Si $|A| < 1$, el error decrece con n y el proceso es estable.

```
import pandas as pd

pd.set_option('display.float_format', '{:.15f}'.format)

def calcular_xn_tabla(A, n, x0=1.0, x1=1/3):
```

```
    """
    Calcula x_n usando la fórmula recursiva y devuelve todos los valores en una lista.
    """
    # Lista para almacenar los resultados: [índice, valor]
    resultados = []

    # Valores iniciales
    xn_minus_2 = x0
    xn_minus_1 = x1

    # Almacenar x0 y x1
    resultados.append({'n': 0, 'x_n': x0})
    resultados.append({'n': 1, 'x_n': x1})

    # Calcular y almacenar desde n=2 hasta el valor deseado
    for i in range(2, n + 1):
        # Fórmula recursiva: x_n = A * x_{n-1} + ((1 - 3 * A) / 9) * x_{n-2}
        xn = A * xn_minus_1 + ((1 - 3 * A) / 9) * xn_minus_2

        # Almacenar el resultado de la iteración actual
        resultados.append({'n': i, 'x_n': xn})

        # Actualizar variables para la próxima iteración
        xn_minus_2 = xn_minus_1
        xn_minus_1 = xn

    return resultados

# Parámetros
A = 4.0  # Coeficiente
n = 20   # Número máximo de iteraciones

# 1. Generar los datos
datos = calcular_xn_tabla(A, n)

# 2. Crear el DataFrame de pandas
df = pd.DataFrame(datos)

# 3. Imprimir la tabla
# La simple visualización del DataFrame de pandas genera una tabla bien formateada en R Markdown.
print(f"Resultados de la recurrencia con A = {A:.15f}:")
```

```
## Resultados de la recurrencia con A = 4.000000000000000:
```

```
df
```

```
##      n               x_n
## 0    0  1.000000000000000
## 1    1  0.333333333333333
## 2    2  0.111111111111111
## 3    3  0.037037037037036
## 4    4  0.012345679012343
## 5    5  0.004115226337439
## 6    6  0.001371742112447
## 7    7  0.000457247370695
```

```
## 8    8  0.000152415789788
## 9    9  0.000050805261637
## 10  10  0.000016935081250
## 11  11  0.000005645005223
## 12  12  0.000001881588252
## 13  13  0.000000626902180
## 14  14  0.000000207889744
## 15  15  0.000000065345200
## 16  16  0.000000007293336
## 17  17 -0.000000050693010
## 18  18 -0.000000211686118
## 19  19 -0.000000784786348
## 20  20 -0.000002880417914
```

```
# Parámetros
A = 0.5  # Coeficiente
n = 20   # Número máximo de iteraciones

# 1. Generar los datos
datos = calcular_xn_tabla(A, n)

# 2. Crear el DataFrame de pandas
df = pd.DataFrame(datos)

# 3. Imprimir la tabla
# La simple visualización del DataFrame de pandas genera una tabla bien formateada en R Markdown.
print(f"Resultados de la recurrencia con A = {A}:")
```

```
## Resultados de la recurrencia con A = 0.5:
```

```
df
```

```
##      n               x_n
## 0    0 1.000000000000000
## 1    1 0.333333333333333
## 2    2 0.111111111111111
## 3    3 0.037037037037037
## 4    4 0.012345679012346
## 5    5 0.004115226337449
## 6    6 0.001371742112483
## 7    7 0.000457247370828
## 8    8 0.000152415790276
## 9    9 0.000050805263425
## 10  10 0.000016935087808
## 11  11 0.000005645029269
## 12  12 0.000001881676423
## 13  13 0.000000627225474
## 14  14 0.000000209075158
## 15  15 0.000000069691719
## 16  16 0.000000023230573
## 17  17 0.000000007743524
## 18  18 0.000000002581175
## 19  19 0.000000000860392
## 20  20 0.000000000286797
```

Última revisión: 25 de octubre, 2025