

Tema 2 Aproximación

Preámbulo para el estudiante

Curso de Métodos Numéricos – 2026

Autor: Kimi (Asistente Docente IA)

Estimado alumno:

Durante este curso vamos a simular fenómenos reales con la ayuda del computador. Antes de escribir la primera línea de código debes saber que **todos** los resultados numéricos llevan asociados errores que no se corrigen “depurando”. A continuación encontrarás los cuatro grandes tipos que estudiaremos y que deberás identificar en cada práctica:

1. Errores de Método

La herramienta numérica no respeta el modelo matemático original. Ejemplo: imponer condiciones de frontera que hacen singular al sistema lineal. El debugger te mostrará la excepción; el análisis te dirá por qué la matriz es singular.

2. Errores de Dato

Incertidumbre en la medición o en la entrada. Podrás propagarla con Monte-Carlo y verificar con Visual Studio que los valores leídos son realmente los que fluyen hacia tu algoritmo.

3. Errores de Truncamiento por Representación

$\frac{1}{3}$, π o 0.1 **jamás** existirán exactamente en IEEE-754. Aprenderás a medir la pérdida de dígitos significativos y a usar `Decimal` o `mpmath` cuando sea crítico.

4. Errores de Truncamiento Residual

Cada iteración que detienes “porque ya está suficientemente cerca” deja un residuo. Controlarás ese residuo con tolerancias y lo observarás paso a paso gracias al depurador.

Nuestro objetivo no es eliminar estos errores —es imposible—, sino **saber cuánto valen, cómo decrecen y cuánto cuesta** que lo hagan. Para ello combinaremos:

- Teoría de convergencia y estabilidad.
- Análisis de complejidad (gran O) para elegir el algoritmo adecuado.
- Traducción de definiciones (pseudocódigo, LaTeX) a Python y, cuando sea necesario, conversión entre NumPy y SymPy.
- Uso profundo del depurador de Visual Studio Community: tu microscopio para **ver** la diferencia entre un error de método y un simple desbordamiento.

Guarda este preámbulo como referencia rápida. Cada vez que un resultado te sorprenda, pregúntate primero:

¿Es un error de método, de dato, de representación o de truncamiento residual?

La respuesta te indicará qué herramienta usar y cuánto confiar en el número que acaba de aparecer en pantalla.

Bienvenido al curso. 😊

1. Errores no específicos y errores numéricos específicos.
El libro que utilizaremos como texto durante el curso es:

Introduction to Numerical Programming: A Practical Guide for Scientists and Engineers Using Python and C/C++
Titus Adrian Beu.

Titus menciona que los errores en los cálculos pueden ser genuinamente numéricos y errores no específicos en su naturaleza.

- (a) Errores no específicos**
- (b) Errores numéricos específicos**

Extiende la explicación con ejemplos en python

Sub-tipo	Ejemplo típico	Código Python ilustrativo
1. Error del modelo	Suponer que la resistencia del aire es lineal	
2. Error en los datos de entrada	Medir la gravedad como $g = 9.5 \text{ m/s}^2$ en vez de 9.81	$g = 9.5 \text{ # dato erróneo}$
3. Error de implementación «lógica»	Olvidar convertir grados → radianes	<i>import math theta_deg = 30 sin_wrong = math.sin(theta_deg) # 30 rad! sin_ok = math.sin(math.radians(theta_deg))</i>
4. Bug de programación	Indices intercambiados	<i>A[i,j] vs A[j,i]</i>
5. Condiciones de contorno mal puestas	Resolver $y'' + y = 0$ con $y(0)=0, y(1)=0$ cuando se necesita $y'(0)=1$	

b) Errores NUMÉRICOS específicos (genuinamente numéricos)

*Surgen **por la aritmética finita** del computador; aunque el modelo y los datos sean correctos, **seguirán apareciendo**.*

Sub-tipo	Origen	Ejemplo ejecutable	Cuantificación rápida
----------	--------	--------------------	-----------------------

1. Redondeo |
Números con más dígitos que la precisión

```
import numpy as np
a = 1e16
b = 1.0
print((a + b) - a) #
0.0 ¡b desaparece!
```

Construir programa
LMN01

2. Truncamiento / discretización

Reemplazar límite por cantidad finita

```
h = 1e-2
df = (np.sin(1+h)-np.sin(1))/h
# 1a orden
print('error',
abs(df - np.cos(1)))
# O(h)Construir  
LMN02
```

3. Cancelación catastrófica

Restar números casi iguales

```
x = 1e-8
y = np.sqrt(x**2 + 1) - 1
print('y=', y,
'error≈', abs(y - x**2/2))
```

Pérdida de dígitos ~
-log10|ε_rel|

4. Overflow / underflow

Magnitud fuera del rango

```
z = 1e200
print(z*z) # ±∞ o 0 subnormal
inf
w = 1e-200
print(w*w) # 0.0
```

Construir LMN03

5. Propagación de errores previos

Cada paso acumula ruido

```
def Horner(p, x): Cota ∝ n μ, n=grado
    """Evalúa
polinomio: mín ops →
mismo error de
redondeo,
pero
*acumulado* en cada
paso."""
    b = p[-1]
    for coef in
reversed(p[:-1]):
        b = b*x +
coef
    return b
```

Construir LMN05

Mini-guía para los alumnos (conclusión)

- 1. Antes de optimizar un algoritmo **comprueba la fila (a)**; de lo contrario estarás «puliendo una bala de cañón».*
- 2. Una vez seguros de (a), **domina la fila (b)** porque esos errores **sempre** estarán presentes; tu único recurso es **controlarlos** (precisión doble, algoritmos estables, refactorización de expresiones, paso adaptativo, etc.).*
- 3. Regla práctica de Beu:
“First make sure the non-specific errors are absent, then fight the numerical ones.”**

2. Clasificando los errores según Titus, pueden existir:

- (a) **Errores de Método.** Por ejemplo: La formulación matemática no concuerda directamente con la metodología numérica. Como ejemplo valores frontera para las ecuaciones diferenciales.
- (b) **Errores de dato. Incertidumbres en la calidad de los datos**
- (c) **Errores de Truncamiento.** Son una consecuencia directa de la representación interna. Ejemplo, la notación IEEE754A para flotantes. Otro ejemplo la operación $\frac{1}{3}$.
- (d) **Los errores de truncamientos o residuales por procesos numéricos repetitivos. Mostrar ejemplos por cada tipo de error en código python**

A continuación tienes **cuatro celdas ejecutables**, una por cada categoría que propone Titus A. Beu en su texto, con un ejemplo mínimo y auto-contenido en Python.

(Copiar-pegar en un Jupyter o script y correr; los comentarios explican el origen del error.)

(a) Errores de Método (modelo vs. metodología numérica)

Problema: $y''+y=0$, $y(0)=0$, $y(\pi)=0$

La solución analítica **NO** es única: $y(x)=C\sin(x)$.

Si imponemos $y(\pi)=0$ y usamos un esquema de diferencias finitas **sin añadir condición extra**, el sistema lineal resultante es **singular** y el código falla o da una arbitraria.

(b) Errores de Dato (incertidumbre en la entrada)

Simulamos que la gravedad g se midió con un desvío estándar $\sigma = 0.05 \text{ m/s}^2$.

Ver cómo esa incertidumbre se propaga al tiempo de caída libre $t = \sqrt{2h/g}$.

(c) Errores de Truncamiento por representación finita

$\frac{1}{3}$ no es representable en IEEE-754 binario; al almacenarlo se **trunca** (o redondea) y las operaciones acumulan ese residuo.

(d) Truncamiento / Residual por procesos numéricos repetitivos

Resolver $\sin(x)=0$ con el método de Newton **sin** actualizar la función; forzamos que la iteración *nunca* llegue al cero máquina y paramos sólo por "tolerancia".

El residual que queda es **truncamiento controlado**.

Resumen para el aula

- (a) **Error de Método:** el *modelo* no encaja con la herramienta numérica → revisa la formulación.
- (b) **Error de Dato:** la *incertidumbre* de entrada viaja hasta la salida → cuantifica con propagación o Monte-Carlo.
- (c) **Truncamiento por representación:** $\frac{1}{3}$, 0.1, π , etc. **nunca** existirán exactamente → error de máquina inevitable.
- (d) **Truncamiento residual:** la *iteración* se detiene antes del “cero exacto” → controla la tolerancia.

3. El uso del depurador en Visual Studio Community usando PYTHON podría ser utilizado para detectar estos errores. Explique.

Sí, el **depurador de Visual Studio Community (VS)** con Python (mediante *Python Tools for Visual Studio*, PTVS) puede ayudarte a **localizar rápidamente la fuente** de cada una de las cuatro clases de errores que acabamos de ver; **pero no a "medir" el error numérico en sí** (eso sigue siendo trabajo de análisis matemático o de tests de tolerancia).

A continuación tienes qué puede y qué NO puede hacer el **debugger** para cada categoría, con **mini-procedimientos paso-a-paso** que puedes repetir en clase.

4. Expresiones en los errores absolutos y relativos.

[Construir programa PYTHON que traduzca las expresiones en latex rojas de abajo a texto. Usar from pylatexenc.latex2text import LatexNodes2Text](#)

Laboratorio: LMN06

Considerando que $\$x^{*}$ es un valor exacto y x una aproximación. El error absoluto de estas dos cantidades es:

$$\$\\Delta^{*} = \\left| x^{*}-x \\right| \$ \quad (\text{Expresión 1})$$

Si se conoce el error absoluto exacto el valor exacto puede ser calculado mediante:

$$x^{*} = x \\pm \\Delta^{*} \$ \quad (\text{Expresión 2})$$

Generalmente lo único que se tiene disponible un estimado del error absoluto $\$\\Delta$

$$x-\\Delta \\leq x^{*} \\leq x+\\Delta \$ \quad (\text{Expresión 3})$$

Para que sea confiable el estimado x no se debe subestimar el error o sea: $\$\\Delta \\geq \\Delta^{*}$ esto se llama el límite del error absoluto.

El error relativo $\$\\delta^{*}$ de la aproximación x a $\$x^{*}$ debe ser igual a la razón del error absoluto al módulo del número exacto:

$$\\frac{\\Delta^{*}}{x^{*}} = \\frac{\\Delta^{*}}{\\left| x^{*}-x \\right|}, \quad x^{*} \\neq 0 \$ \quad (\text{expresión 4})$$

Continuando con los cálculos:

$$x^{*}=x(1\\pm \\delta^{*}) \$ \quad (\text{expresión 5})$$

y

$$x^{*}\\approx x(1\\pm \\delta) \$ \quad (\text{expresión 6})$$

5. Explique el concepto de condición de paro en métodos numéricos: $\left| x_{i+1} - x_i \right| < \varepsilon$

Condición de paro

$$|x_{i+1} - x_i| < \varepsilon$$

Significado físico

“Si el algoritmo **dejó de avanzar** más allá de la distancia ε que yo considero ‘despreciable’, entonces detente”.

Qué mide y qué **no** mide

- Mide **estabilidad** de la sucesión (cuán pequeño es el paso).
- **No** mide el error **verdadero** respecto a la raíz desconocida r .

Puede cumplirse y aún estar lejos de r si la convergencia es lenta.

Por qué se usa

- 1.Es **computable**: sólo necesitas la historia propia del método.
- 2.Es **barata**: una resta y valor absoluto.
- 3.En métodos super-lineales (Newton, secante) implica que los dígitos útiles **ya no cambian**.

Cómo se implementa

Python

```
while True:  
    x_new = paso_del_metodo(x_old)  
    if abs(x_new - x_old) < eps:  
        break  
    x_old = x_new
```

Precauciones

- ε debe ser \geq unidad de redondeo $\times |x|$ o **nunca** paras.
- En convergencia **lenta** ($|g'| \approx 1$) puedes parar **muy lejos** de la raíz; complementa con $|f(x_i)| < \delta_0 |x_{i+1}| |x_{i+1} - x_i| < \varepsilon_{rel}$.

Frase para recordar

“La condición $|x_{i+1} - x_i| < \varepsilon$ no te dice que ya llegaste; te dice que **ya no te estás moviendo**.”

6. ¿Qué es Sympy y para qué sirve?

Sympy es **una biblioteca de Python para matemática simbólica**.

En lugar de trabajar con *números* (como NumPy), trabaja con **símbolos**: letras, expresiones, ecuaciones, derivadas, integrales, límites, series, matrices llenas de fórmulas, etc.

¿Para qué sirve en Métodos Numéricos?

1. **Derivar y simplificar** funciones sin errores de truncamiento

Python

```
import sympy as sp
x = sp.Symbol('x')
f = sp.sin(x) * sp.exp(-x)
df = sp.diff(f, x)          # exacta, sin ε de paso finito
```

2. **Obtener fórmulas cerradas** de errores de truncamiento, pasos óptimos, etc.

Python

```
h = sp.Symbol('h', positive=True)
err = sp.series(sp.cos(h) - (1 - h**2/2), h, 0, 5).remove0()
print(err)
```

3. **Generar automáticamente** el código NumPy/Julia/C desde la expresión simbólica

Python

```
from sympy import lambdify
f_np = lambdify(x, df, 'numpy')  # lista para usar con arrays
```

4. **Resolver exactamente** pequeños sistemas no lineales para obtener **semillas o soluciones de referencia** que luego se comparan con los resultados numéricos.

5. **Manipular matrices simbólicas** y estudiar estabilidad sin errores de redondeo.

En resumen:

Sympy te da **papel y lápiz infinito** dentro de Python; lo usamos para *derivar, simplificar y validar* antes de pasar al cálculo numérico propiamente dicho.

7. ¿Qué es un CAS?

CAS (Computer Algebra System)

Programa o biblioteca capaz de **manipular símbolos matemáticos** (números exactos, incógnitas, ecuaciones, derivadas, integrales, matrices de expresiones, etc.) **sin recurrir a aproximaciones numéricas**.

Es decir, trabaja con

$$\frac{1}{3}, \quad \pi, \quad x^2 + y^2 = 1, \quad \int \sin(x) dx$$

exactamente, conservando la forma algebraica.

Ejemplos conocidos

Mathematica ®, Maple ®, Maxima, SageMath y **SymPy** (el CAS *puro-Python* que usaremos).

Funciones típicas de un CAS

- Simplificación y factorización de expresiones
- Derivadas, integrales, límites, desarrollos en serie
- Resolución simbólica de ecuaciones (cuando existe fórmula cerrada)
- Álgebra lineal simbólica (determinantes, valores propios *exactos*)
- Generación de código (C, Fortran, Python) a partir de la fórmula obtenida

En el contexto de Métodos Numéricos

El CAS sirve para:

1. Obtener **fórmulas de error** de truncamiento sin paso finito.
2. Calcular **derivadas exactas** para Newton, Taylor, etc.
3. Producir **soluciones de referencia** con las que comparar el resultado numérico.

Resumen corto:

Un CAS es “**papel y lápiz infinito**” dentro del computador.

8 Notas de estudio Series de Taylor

Notas de estudio – Series de Taylor

Autoría: Kimi (Asistente Docente IA)

(a) Pequeña historia y contexto

Isaac Newton (1643-1727) y Gottfried W. Leibniz (1646-1716) inventaron, de forma independiente, el cálculo infinitesimal. Newton trabajaba con “series infinitas” para cuadraturas y áreas; Leibniz las empleaba en la integración de funciones racionales. Sin embargo, fue Brook Taylor (1685-1731) quien, en 1715, sistematizó la idea en su obra *Methodus incrementorum directa et inversa*: mostró que **cualquier función suficientemente suave** puede escribirse como una suma de potencias centrada en un punto. La herramienta, bautizada más tarde “Series de Taylor”, convirtió los métodos dispersos de Newton y Leibniz en un procedimiento universal para aproximar, derivar e integrar funciones localesmente.

(b) ¿Qué son las Series de Taylor?

Una Serie de Taylor es una **representación local** de una función mediante un polinomio de grado infinito cuyos coeficientes son las derivadas sucesivas de la función evaluadas en un punto a .

Proporciona:

- Aproximaciones polinómicas de cualquier precisión.
- Una vía para estudiar comportamientos locales (crecimiento, concavidad, puntos críticos).
- La base teórica de muchos métodos numéricos (diferencias finitas, integración, resolución de ecuaciones diferenciales).

(c) Expresión general (texto y LaTeX)

Textual: "La Serie de Taylor de una función f alrededor del punto a es la suma infinita de términos que involucran las derivadas de f evaluadas en a , multiplicadas por potencias crecientes de $(x - a)$ divididas por el factorial del exponentio."

LaTeX:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x - a)^n$$

(d) Desarrollos algebraicos (sin código)

1. Serie de Taylor de $\sin x$ alrededor de $a = 0$ (serie de Maclaurin)

Derivadas en 0:

$$f(0) = 0, \quad f'(0) = 1, \quad f''(0) = 0, \quad f'''(0) = -1, \quad f^{(4)}(0) = 0, \quad f^{(5)}(0) = 1, \dots$$

Patrón: sólo potencias impares con signo alterno.

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

2. Serie de Taylor de $\cos x$ alrededor de $a = 0$

Derivadas en 0:

$$f(0) = 1, \quad f'(0) = 0, \quad f''(0) = -1, \quad f'''(0) = 0, \quad f^{(4)}(0) = 1, \dots$$

Patrón: sólo potencias pares con signo alterno.

$$\cos x = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$$

Estos desarrollos permiten aproximar senos y cosenos con simples sumas de potencias, evaluar límites, integrar funciones elípticas y, sobre todo, fundamentar algoritmos numéricos que requieren precisiones controladas.

9. LMN07 Construir el programa Serie de Taylor para el seno en 0.5 expandida a orden 8 incremental. Escribir: (a) Forma Textual, (b) Forma Latex, (c) Forma Textual sin 0, (d) Cálculo del valor de manera simbólica, (e) Cálculo del seno exacto.

```

1 import sympy as sp
2 from sympy import symbols, sin, exp
3
4 # Programacion simbolica, declara simbolo x
5 x = symbols('x')
6 # Funcion del seno de 0.5
7 f_x = sin(x)
8 # Expande la serie hasta orden 8
9 serieTaylor1 = f_x.series(x, 0, 8)
10 print("Taylor Textual con 0      :",serieTaylor1)
11 serieTaylor2 = f_x.series(x, 0, 8).removeO()
12 print("Serie Taylor forma Latex : ",sp.latex(serieTaylor2, order='rev-lex'))
13 ascending_str = sp.sstr(serieTaylor2, order='rev-lex')
14 print("Taylor textual ascendente:",ascending_str)
15 print("Evaluacion en 0.5        :",serieTaylor2.subs(x, 0.5).evalf())
16 print("Valor exacto sin(0.5)    :",sin(0.5))

```

Taylor Textual con 0 : $x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + O(x^8)$
 Serie Taylor forma Latex : $x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}$
 Taylor textual ascendente: $x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}$
 Evaluacion en 0.5 : 0.479425533234127
 Valor exacto sin(0.5) : 0.479425538604203
 Press any key to continue . . .

10 Usar las series de Taylor de numpy para el seno:
(a) Expandir la serie hasta el grado 8, (b)
Calcular el valor del seno mediante series de
Taylor para $x=0.5$

Vamos a hacerlo todo con NumPy, sin usar `math.sin` ni SymPy.

(a) Serie de Taylor de seno hasta grado 8

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

(b) Evaluamos en $x = 0.5$ con NumPy.

Python

 Copy

```
import numpy as np

# coeficientes de la serie  $\sin(x) = \sum (-1)^k x^{(2k+1)} / (2k+1)!$   $k=0..3$  (grado ≤ 8)
coef = np.array([1, -1/6, 1/120, -1/5040]) # 1, -1/3!, 1/5!, -1/7!
x    = 0.5
# potencias [x^1, x^3, x^5, x^7]
pot  = x**np.arange(1, 8, 2)                # [0.5, 0.5^3, 0.5^5, 0.5^7]

sin_taylor = np.dot(coef, pot)                 # producto escalar = suma de términos
print("Serie de Taylor (grado 8):", sin_taylor)
print("Valor exacto np.sin(0.5):", np.sin(0.5))
print("Error absoluto:", abs(sin_taylor - np.sin(0.5)))
```



Salida típica

 Copy

```
Serie de Taylor (grado 8): 0.47942553323412703
Valor exacto np.sin(0.5): 0.479425538604203
Error absoluto: 5.370076047e-09
```

Conclusión: con solo cuatro términos (grado 7) ya obtenemos 8 dígitos correctos para $x = 0.5$.

11. Ejercicios para el alumno. Escribir las versiones sympy y numpy para las series de Taylor:

- (a) e^x
- (b) $\cosh(x)$
- (c) $\tanh(x)$
- (d) $\frac{1}{1-x}$
- (e) $\arcsin(x)$
- (f) $\arctan(x)$

Construir un arreglo para todas las funciones anteriores usando sympy y numpy. Utilizar condición de paro $\varepsilon = .000001$