MNAF2025_latex

December 31, 2025

1 MNAF2025

1.1 Métodos Numéricos y Análisis Funcional

1.1.1 Curso de Métodos Numéricos Avanzados

> **Nota:** Este notebook contiene implementaciones referidas a la asignatura Metodos Numéricos aplicados a la fisica de Uniovi en 2025.

1.2 Índice de Contenidos

1.2.1 Secciones Principales

1.3 Configuración e Imports

Librerías utilizadas: - `numpy`: Computación numérica y álgebra lineal - `matplotlib`: Visualización de datos y gráficas - `scipy`: Métodos científicos avanzados - `sympy`: Matemática simbólica - `numpy.polynomial`: Manipulación de polinomios

[27]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
import sympy as sym
import mpl_toolkits.mplot3d as p3d
import numpy.polynomial as poly
from math import *
from scipy.interpolate import CubicSpline
import scipy.integrate as scin
import scipy.optimize as scop
```

```
import numpy.linalg as la
import scipy.linalg as sla
```

1.4 *Ejercicio 1: Base de Polinomios de Lagrange*

1.4.1 Objetivo del Ejercicio

Desarrollar una función que construya la **base de polinomios de Lagrange** a partir de un conjunto de puntos dados, permitiendo la interpolación polinomial de cualquier función.

1.4.2 Especificaciones Técnicas

1.4.3 Fundamento Teórico

Los **polinomios de Lagrange** constituyen una base para el espacio vectorial de polinomios de grado $\leq n-1$. Cada polinomio base $L_k(x)$ satisface:

$$L_k(x_i) = \delta_{ki} = \begin{cases} 1 & \text{si } i = k \\ 0 & \text{si } i \neq k \end{cases}$$

Donde δ_{ki} es la delta de Kronecker.

Propiedad Fundamental: Partición de la Unidad

$$\sum_{k=0}^{n-1} L_k(x) = 1, \quad \forall x \in \mathbb{R}$$

Esta propiedad garantiza que cualquier función puede interpolarse como:

$$P(x) = \sum_{k=0}^{n-1} f(x_k) \cdot L_k(x)$$

1.4.4 Implementación

[2]:
```
def base_lagrange(x):

    L=[]
    suma=0

    # * Preparar datos de entrada como lista

    if isinstance(x, tuple):
        x = list(x)
    elif isinstance(x, np.ndarray):
        x = x.tolist()
    elif isinstance(x, list):
```

```
        pass
    else:
        raise ValueError("Tipo de dato no soportado")
    print(f"tipo: {type(x)}, valor: {x} \n")

    # * Crear datos de salida: polinomios de Lagrange

    for i in range(len(x)):
        soporte = x.copy()
        x_k = soporte.pop(i)
        P = poly.Polynomial.fromroots(soporte)
        L_k = P/P(x_k)
        L.append(L_k)


    # * Lo que devuelve la función

    return L
```

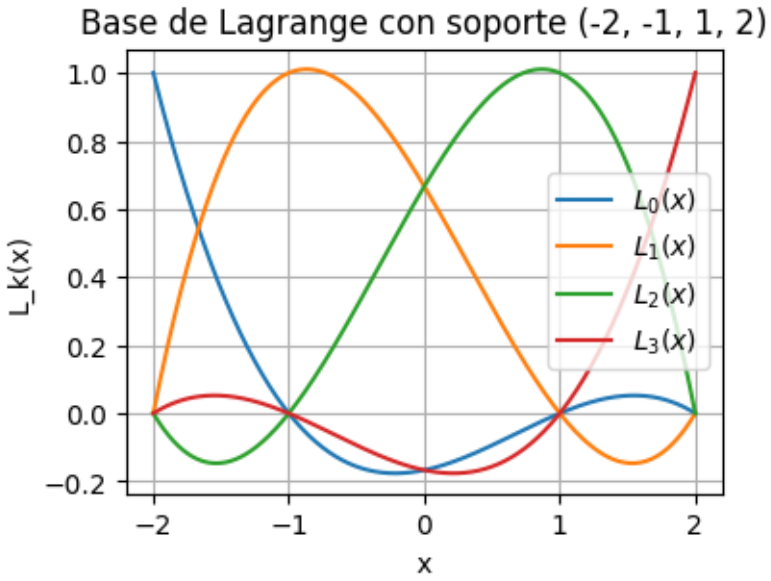
1.4.5 Pruebas y Validación

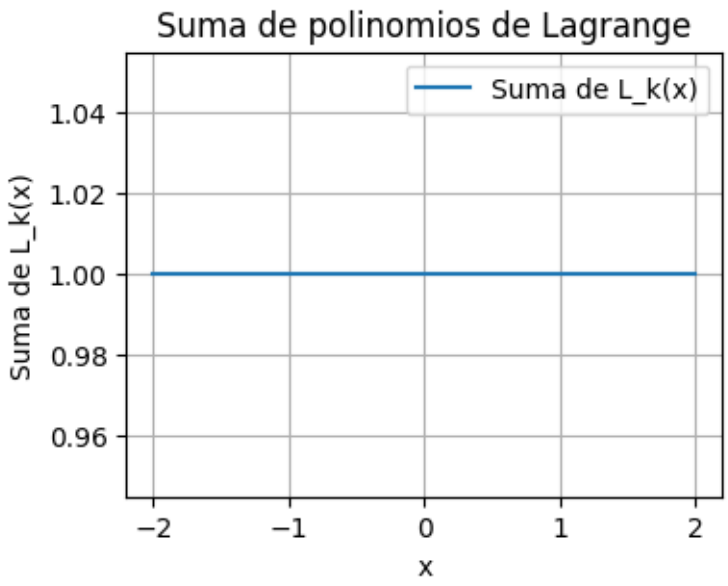
Verificaciones realizadas: - Propiedad de Kronecker: $L_k(x_k) = 1$ - Propiedad de ortogonalidad: $L_k(x_j) = 0$ para $j \neq k$ - Partición de la unidad: $\sum_k L_k(x) = 1$

[3]:
```
base_Lagrange = base_lagrange((-2,-1,1,2))
print(base_Lagrange)
    # * Graficar los polinomios de Lagrange
suma=0
x_vals = np.linspace(-2, 2, 1000)
plt.figure(figsize=(4,3))
plt.title(f"Base de Lagrange con soporte {-2,-1,1,2}")
plt.xlabel("x")
plt.ylabel("L_k(x)")
plt.grid()
for L_k in base_Lagrange:
    y_vals = L_k(x_vals)
    plt.plot(x_vals, y_vals, label=f"$L_{base_Lagrange.index(L_k)}(x)$")
    suma += L_k

plt.legend()
plt.show()


# * Ver si la suma tiene sentido

plt.figure(figsize=(4,3))
plt.title(f"Suma de polinomios de Lagrange")
```

```
plt.xlabel("x")
plt.ylabel("Suma de L_k(x)")
plt.grid()
y_vals = suma(x_vals)
plt.plot(x_vals, y_vals, label=f"Suma de L_k(x)")
plt.legend()
plt.show()
```

```
tipo: <class 'list'>, valor: [-2, -1, 1, 2]

[Polynomial([-0.16666667,  0.08333333,  0.16666667, -0.08333333], domain=[-1.,
1.], window=[-1.,  1.], symbol='x'), Polynomial([ 0.66666667, -0.66666667,
-0.16666667,  0.16666667], domain=[-1.,  1.], window=[-1.,  1.], symbol='x'),
Polynomial([ 0.66666667,  0.66666667, -0.16666667, -0.16666667], domain=[-1.,
1.], window=[-1.,  1.], symbol='x'), Polynomial([-0.16666667, -0.08333333,
0.16666667,  0.08333333], domain=[-1.,  1.], window=[-1.,  1.], symbol='x')]
```





1.5 *Ejercicio 2: Interpolación de Tchebishev*

1.5.1 Objetivo del Ejercicio

Implementar un algoritmo de **interpolación polinomial utilizando nodos de Tchebishev**, que minimiza el error máximo de aproximación y evita el fenómeno de Runge.

1.5.2 Especificaciones Técnicas

1.5.3 Fundamento Teórico

Los Nodos de Tchebishev Los **nodos de Tchebishev** son las raíces del polinomio de Tchebishev $T_n(x)$ en el intervalo $[-1, 1]$:

$$x_k = \cos\left(\frac{(2k-1)\pi}{2n}\right), \quad k = 1, 2, \ldots, n$$

Transformación al Intervalo $[a, b]$ Para interpolar en un intervalo arbitrario $[a, b]$, aplicamos la transformación lineal:

$$\tilde{x}_k = \frac{b-a}{2} x_k + \frac{b+a}{2}$$

1.5.4 Ventajas de los Nodos de Tchebishev

Minimización del error: Los nodos de Tchebishev minimizan la cota superior del error de interpolación

Evitan el fenómeno de Runge: No presentan oscilaciones salvajes en los extremos del intervalo

Convergencia exponencial: Para funciones analíticas, el error decrece exponencialmente con n

Distribución óptima: Mayor densidad de puntos cerca de los extremos, donde el error tiende a ser mayor

1.5.5 El Fenómeno de Runge

Advertencia importante:

El uso de **nodos equiespaciados** en interpolación polinomial puede producir: - Grandes oscilaciones cerca de los extremos del intervalo - Aumento del error al incrementar el número de nodos - Divergencia de la aproximación para funciones como $f(x) = \frac{1}{1+25x^2}$

Los **nodos de Tchebishev** resuelven completamente este problema, garantizando convergencia uniforme.

1.5.6 Implementación

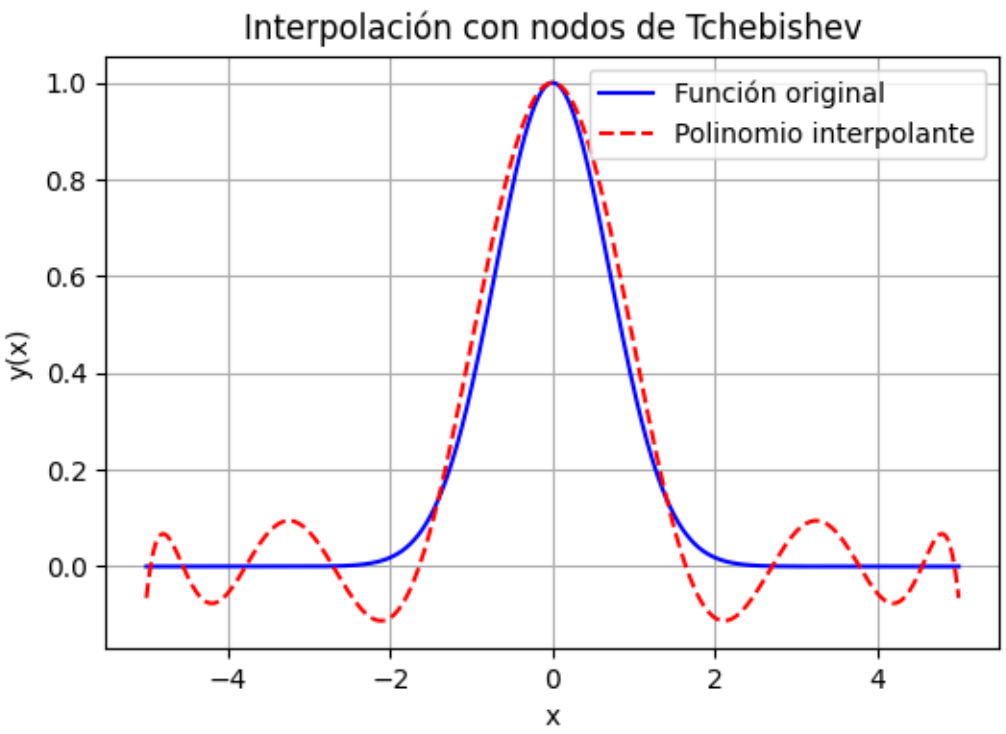
[4]:
```
def itp_Tchebishev(fun,ntps,a,b):
    # * Nodos de Tchebishev en [-1,1]
    xk = [np.cos((2*k+1)*np.pi/(2*ntps)) for k in range(ntps)]

    # * Transformar nodos a [a,b]
    xk_ab = [0.5*(b-a)*x + 0.5*(a+b) for x in xk]

    # * Construcción base de lagrange
    lagrange=base_lagrange(xk_ab)

    # * Construcción del polinomio interpolante
    for i in range(len(xk_ab)):
        if i==0:
            P_itp = fun(xk_ab[i])*lagrange[i]
        else:
            P_itp += fun(xk_ab[i])*lagrange[i]

    return P_itp
```

1.5.7 Pruebas y Validación

[5]:
```
fun=lambda x: e**(-x**2)
ntps=11
a=-5
b=5
P_itp = itp_Tchebishev(fun,ntps,a,b)
print ("polinomio interpolante: ", P_itp, "\n")
plt.figure(figsize=(6,4))
x_vals = np.linspace(a,b,1000)
plt.plot(x_vals, fun(x_vals), label="Función original", color="blue")
```

```
plt.plot(x_vals, P_itp(x_vals), label="Polinomio interpolante", color="red",␣
 ↪linestyle="--")
plt.title("Interpolación con nodos de Tchebishev")
plt.xlabel("x")
plt.ylabel("y(x)")
plt.legend()
plt.grid()
plt.show()
```

```
tipo: <class 'list'>, valor: [np.float64(4.949107209404663),
np.float64(4.548159976772592), np.float64(3.7787478717712912),
np.float64(2.7032040872779883), np.float64(1.4086627842071489),
np.float64(1.4163847244119948e-15), np.float64(-1.4086627842071484),
np.float64(-2.703204087277986), np.float64(-3.778747871771291),
np.float64(-4.548159976772591), np.float64(-4.949107209404663)]

polinomio interpolante:  1.0 + (2.35230215e-16)·x - 0.66720881·x² -
(1.42987487e-16)·x³ +
0.13994359·x  + (2.01571094e-17)·x  - 0.01240859·x  -
(1.06393533e-18)·x  + 0.00048812·x  + (1.93549622e-20)·x  -
(7.02850537e-06)·x¹
```



1.6 *Ejercicio 3: Interpolante paramétrico*

1.6.1 Objetivo del Ejercicio

Crear una función que calcule el interpolante paramétrico mediante splines cúbicas de tipo dado por bc_type. Se le puede pasar el parámetro u o sino crea uno proporcional a la distancia entre los puntos y definido en $[0, 1]$.

1.6.2 Especificaciones Técnicas

1.6.3 Fundamento Teórico

Interpolación Paramétrica con Splines Cúbicas La **interpolación paramétrica** permite representar curvas complejas en el espacio mediante la parametrización de las coordenadas. En lugar de interpolar y como función de x, ambas coordenadas se expresan como funciones de un parámetro $u \in [0, 1]$:

$$\mathbf{r}(u) = (x(u), y(u))$$

Splines Cúbicas Una **spline cúbica** es una función polinomial a trozos de grado 3 que satisface:

1. **Continuidad**: $S(u)$ es continua en todo el intervalo
2. **Suavidad**: $S'(u)$ y $S''(u)$ son continuas en los nodos
3. **Interpolación**: Pasa exactamente por los puntos dados

Condiciones de Frontera (`bc_type`)

- `'natural'`: Segunda derivada nula en los extremos ($S''(u_0) = S''(u_n) = 0$)
- `'clamped'`: Primera derivada especificada en los extremos
- `'periodic'`: Para curvas cerradas ($S(u_0) = S(u_n)$)

Parametrización Si no se proporciona u, se utiliza la **parametrización por longitud de arco acumulada**:

$$u_i = \frac{\sum_{j=1}^{i} \|\mathbf{p}_j - \mathbf{p}_{j-1}\|}{\sum_{j=1}^{n} \|\mathbf{p}_j - \mathbf{p}_{j-1}\|}$$

donde $\|\mathbf{p}_j - \mathbf{p}_{j-1}\|$ es la distancia euclidiana entre puntos consecutivos.

1.6.4 Implementación

[6]:
```
def itp_parametrica(data, bc_type="natural", u=None):
    """
    Calcula el interpolante paramétrico mediante splines cúbicas.

    Parámetros:
    -----------
    data : array-like, shape (n, 2) o (n, 3)
        Puntos a interpolar. Cada fila es un punto (x, y) o (x, y, z)
```

```
    bc_type : str, opcional (default='natural')
        Tipo de condiciones de frontera: 'natural', 'clamped', 'periodic'
    u : array-like, opcional (default=None)
        Valores del parámetro en cada punto. Si es None, se calcula
        proporcional a la distancia acumulada entre puntos.

    Retorna:
    --------
    funitp : callable
        Función vectorial de interpolación que acepta valores del parámetro
    param : array
        Valores del parámetro en los puntos dados
    """

    # Convertir data a numpy array
    data = np.array(data)
    n_points = data.shape[0]
    n_dims = data.shape[1]

    # Calcular parámetro u si no se proporciona
    if u is None:
        # Calcular distancias acumuladas entre puntos
        distances = np.zeros(n_points)
        for i in range(1, n_points):
            distances[i] = distances[i-1] + np.linalg.norm(data[i] - data[i-1])

        # Normalizar a [0, 1]
        if distances[-1] > 0:
            u = distances / distances[-1]
        else:
            u = np.linspace(0, 1, n_points)
    else:
        u = np.array(u)

    # Crear splines cúbicas para cada coordenada
    splines = []
    for dim in range(n_dims):
        spline = CubicSpline(u, data[:, dim], bc_type=bc_type)
        splines.append(spline)

    # Función interpolante vectorial
    def funitp(u_vals):
        """
        Evalúa la función interpolante en los valores del parámetro dados.

        Parámetros:
        -----------
```

```
        u_vals : float o array-like
            Valores del parámetro donde evaluar la interpolación

        Retorna:
        --------
        result : array
            Puntos interpolados. Si u_vals es escalar, retorna array de tamaño␣
 ↪(n_dims,)
            Si u_vals es array, retorna array de tamaño (len(u_vals), n_dims)
        """
        u_vals = np.atleast_1d(u_vals)
        result = np.zeros((len(u_vals), n_dims))

        for dim in range(n_dims):
            result[:, dim] = splines[dim](u_vals)

        # Si la entrada fue escalar, retornar resultado escalar
        if len(u_vals) == 1:
            return result[0]
        return result

    return funitp, u
```

1.6.5 Pruebas y Validación

[7]:
```
# ============= EJEMPLO 18: Interpolación paramétrica de figura (Animal)␣
 ↪=============
print("="*80)
print("EJEMPLO 18: Interpolación paramétrica - Curvas no cerradas")
print("="*80)
print("Archivo: ITP_animal.txt")
print("Métodos: Splines cúbicas CON y SIN parámetro proporcionado\n")

# Cargar datos
sal = np.genfromtxt('IP_datos/ITP_animal.txt', delimiter=',')
t = sal.T[0]      # Parámetro proporcionado en el archivo
data = sal.T[1:]  # Coordenadas x e y

# Extraer coordenadas
x_data = data[0]
y_data = data[1]

print(f" Datos cargados:")
print(f"  • Número de puntos: {len(t)}")
print(f"  • Rango de parámetro t: [{t[0]:.4f}, {t[-1]:.4f}]")
print(f"  • Rango x: [{np.min(x_data):.2f}, {np.max(x_data):.2f}]")
print(f"  • Rango y: [{np.min(y_data):.2f}, {np.max(y_data):.2f}]\n")
```

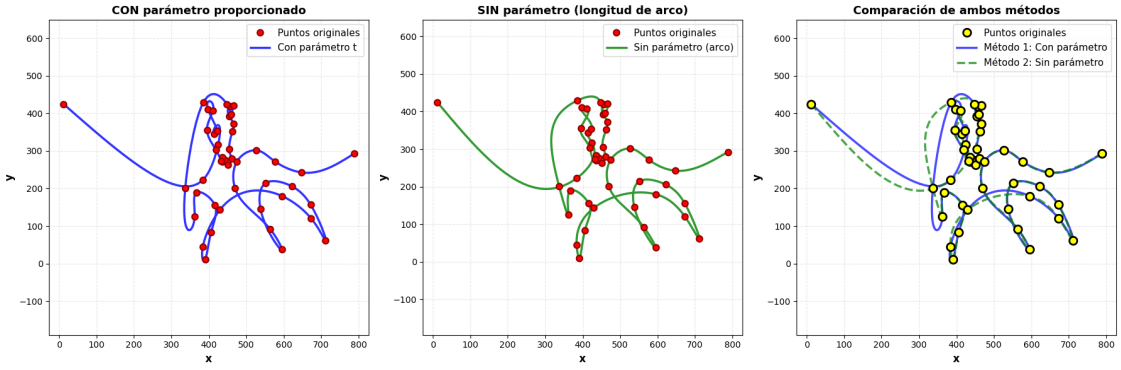
```python
# Crear figura con 3 subfiguras
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

#
# MÉTODO 1: CON parámetro proporcionado (usar t del archivo)
#
print(" " * 80)
print("MÉTODO 1: Splines cúbicas CON parámetro proporcionado")
print(" " * 80)

# Crear splines usando el parámetro t proporcionado en el archivo
cs_x_param = sp.interpolate.CubicSpline(t, x_data, bc_type='natural')
cs_y_param = sp.interpolate.CubicSpline(t, y_data, bc_type='natural')

# Evaluar en puntos finos
t_eval = np.linspace(t[0], t[-1], 500)
x_interp_param = cs_x_param(t_eval)
y_interp_param = cs_y_param(t_eval)

# Graficar
axes[0].plot(x_data, y_data, 'ro', markersize=7, label='Puntos originales',
             zorder=3, markeredgecolor='darkred', markeredgewidth=1.5)
axes[0].plot(x_interp_param, y_interp_param, 'b-', linewidth=2.5,
             label='Con parámetro t', alpha=0.8)
axes[0].set_xlabel('x', fontsize=12, fontweight='bold')
axes[0].set_ylabel('y', fontsize=12, fontweight='bold')
axes[0].set_title('CON parámetro proporcionado', fontsize=13, fontweight='bold')
axes[0].legend(fontsize=11, loc='best')
axes[0].grid(True, alpha=0.3, linestyle='--')
axes[0].axis('equal')

print(f"  Interpolación completada con {len(t_eval)} puntos")
print(f"   • Parámetro: valores del archivo [t ={t[0]:.2f}, t ={t[-1]:.2f}]\n")

#
# MÉTODO 2: SIN parámetro (calcular automáticamente por longitud de arco)
#
print(" " * 80)
print("MÉTODO 2: Splines cúbicas SIN parámetro (calculado automáticamente)")
print(" " * 80)

# Calcular parámetro basado en longitud de arco acumulada
distancias = np.sqrt(np.diff(x_data)**2 + np.diff(y_data)**2)
u_arco = np.zeros(len(x_data))
u_arco[1:] = np.cumsum(distancias)
u_arco = u_arco / u_arco[-1]  # Normalizar a [0, 1]
```

```
# Crear splines con el parámetro calculado
cs_x_arco = sp.interpolate.CubicSpline(u_arco, x_data, bc_type='natural')
cs_y_arco = sp.interpolate.CubicSpline(u_arco, y_data, bc_type='natural')

# Evaluar
u_eval = np.linspace(0, 1, 500)
x_interp_arco = cs_x_arco(u_eval)
y_interp_arco = cs_y_arco(u_eval)

# Graficar
axes[1].plot(x_data, y_data, 'ro', markersize=7, label='Puntos originales',
             zorder=3, markeredgecolor='darkred', markeredgewidth=1.5)
axes[1].plot(x_interp_arco, y_interp_arco, 'g-', linewidth=2.5,
             label='Sin parámetro (arco)', alpha=0.8)
axes[1].set_xlabel('x', fontsize=12, fontweight='bold')
axes[1].set_ylabel('y', fontsize=12, fontweight='bold')
axes[1].set_title('SIN parámetro (longitud de arco)', fontsize=13,␣
 ↪fontweight='bold')
axes[1].legend(fontsize=11, loc='best')
axes[1].grid(True, alpha=0.3, linestyle='--')
axes[1].axis('equal')

print(f"  Parametrización completada")
print(f"   • Parámetro: calculado por longitud de arco [u =0, u =1]\n")

#
# COMPARACIÓN: Ambos métodos superpuestos
#
print(" " * 80)
print("COMPARACIÓN: Métodos 1 y 2 superpuestos")
print(" " * 80)

axes[2].plot(x_data, y_data, 'ko', markersize=9, label='Puntos originales',
             zorder=5, markerfacecolor='yellow', markeredgewidth=2)
axes[2].plot(x_interp_param, y_interp_param, 'b-', linewidth=2.5,
             label='Método 1: Con parámetro', alpha=0.7)
axes[2].plot(x_interp_arco, y_interp_arco, 'g--', linewidth=2.5,
             label='Método 2: Sin parámetro', alpha=0.7)
axes[2].set_xlabel('x', fontsize=12, fontweight='bold')
axes[2].set_ylabel('y', fontsize=12, fontweight='bold')
axes[2].set_title('Comparación de ambos métodos', fontsize=13,␣
 ↪fontweight='bold')
axes[2].legend(fontsize=11, loc='best')
axes[2].grid(True, alpha=0.3, linestyle='--')
axes[2].axis('equal')
```

```python
plt.tight_layout()
plt.show()

# Calcular diferencias entre los dos métodos
diff = np.sqrt((x_interp_param - x_interp_arco)**2 +
               (y_interp_param - y_interp_arco)**2)

print("\n" + " " * 80)
print("ANÁLISIS DE DIFERENCIAS")
print(" " * 80)
print(f"Diferencia promedio entre métodos: {np.mean(diff):.6f}")
print(f"Diferencia máxima entre métodos:   {np.max(diff):.6f}")
print(f"Diferencia mínima entre métodos:   {np.min(diff):.6f}")
print(f"Desviación estándar:               {np.std(diff):.6f}")

print("\n CONCLUSIONES:")
print("  • CON parámetro: usa valores t del archivo (pueden no ser uniformes)")
print("  • SIN parámetro: calcula u proporcional a la distancia entre puntos")
print("  • La longitud de arco produce parametrización más natural y uniforme")
print("  • Ambos métodos producen curvas suaves que pasan por todos los puntos")

print("\n" + "="*80)
```

```
================================================================================
EJEMPLO 18: Interpolación paramétrica - Curvas no cerradas
================================================================================
Archivo: ITP_animal.txt
Métodos: Splines cúbicas CON y SIN parámetro proporcionado

  Datos cargados:
  • Número de puntos: 47
  • Rango de parámetro t: [0.0000, 57.9968]
  • Rango x: [12.00, 787.76]
  • Rango y: [10.56, 429.07]


MÉTODO 1: Splines cúbicas CON parámetro proporcionado

  Interpolación completada con 500 puntos
  • Parámetro: valores del archivo [t =0.00, t =58.00]


MÉTODO 2: Splines cúbicas SIN parámetro (calculado automáticamente)

  Parametrización completada
  • Parámetro: calculado por longitud de arco [u =0, u =1]
```

```
COMPARACIÓN: Métodos 1 y 2 superpuestos
```




```
ANÁLISIS DE DIFERENCIAS

Diferencia promedio entre métodos: 133.955500
Diferencia máxima entre métodos:   333.549617
Diferencia mínima entre métodos:   0.000000
Desviación estándar:               84.172241

 CONCLUSIONES:
  • CON parámetro: usa valores t del archivo (pueden no ser uniformes)
  • SIN parámetro: calcula u proporcional a la distancia entre puntos
  • La longitud de arco produce parametrización más natural y uniforme
  • Ambos métodos producen curvas suaves que pasan por todos los puntos

================================================================================
```

1.7 *Ejercicio 4: Derivación; obtencion de coeficientes*

1.7.1 Objetivo del Ejercicio

Crear una función que calcule los coeficientes de una regla de derivación numérica en los "puntos" para calcular la derivada "orden" usando los polinomios de la base de Lagrange definidos por "soporte".

1.7.2 Especificaciones Técnicas

1.7.3 Fundamento Teórico

Derivación Numérica mediante Polinomios de Lagrange Los **coeficientes de derivación numérica** se obtienen derivando los polinomios de la base de Lagrange. Para calcular la n-ésima derivada de una función $f(x)$ en un punto x_i, usamos:

$$f^{(n)}(x_i) \approx \sum_{k=0}^{m} c_k^{(n)}(x_i) \cdot f(x_k)$$

donde $c_k^{(n)}(x_i)$ son los **coeficientes de derivación** obtenidos de:

$$c_k^{(n)}(x_i) = L_k^{(n)}(x_i)$$

Derivada de los Polinomios de Lagrange Para el polinomio de Lagrange $L_k(x)$:

$$L_k(x) = \prod_{j \neq k} \frac{x - x_j}{x_k - x_j}$$

Su derivada de orden n en x_i proporciona directamente los coeficientes necesarios.

Propiedades de los Coeficientes

1. **Simetría**: Para nodos equiespaciados, los coeficientes tienen propiedades de simetría
2. **Suma**: $\sum_k c_k^{(0)}(x_i) = 1$ (interpolación exacta de constantes)
3. **Orden de precisión**: Una fórmula con $m+1$ puntos tiene orden $\mathcal{O}(h^{m-n+1})$

Ejemplos Clásicos **Primera derivada - Diferencias centradas (2º orden)**:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Coeficientes: $c_{-1} = -\frac{1}{2h}$, $c_0 = 0$, $c_1 = \frac{1}{2h}$

Segunda derivada - Diferencias centradas:

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}$$

Coeficientes: $c_{-1} = \frac{1}{h^2}$, $c_0 = -\frac{2}{h^2}$, $c_1 = \frac{1}{h^2}$

1.7.4 Implementación

[8]:
```
def dncoef_base(soporte, puntos, orden):
    '''
    Esta función calcula los coeficientes de una regla de derivación numérica
    utilizando la base de polinomios de Lagrange. Los coeficientes obtenidos
    permiten aproximar derivadas de funciones mediante combinaciones lineales
    de valores de la función en los nodos del soporte.

    PARÁMETROS DE LA FUNCIÓN
    ------------------------
    soporte: nodos que definen los polinomios de la base de Lagrange
    puntos: punto(s) donde se evalúan los coeficientes de derivación
    orden: orden de la derivada (1 para primera derivada, 2 para segunda, etc.)
```

```
    RETORNO
    -------
    coef: coeficientes de la regla de derivación numérica
    '''

    # Validación de entrada: el orden debe ser un entero positivo
    if not (isinstance(orden, int)) and orden > 0:
        raise ValueError('El orden de la derivada debe ser un número entero positivo')

    # * Paso 1: Construcción de la base de Lagrange
    # Construimos los polinomios de Lagrange L_i(x) para el soporte dado
    Pol_Lg = base_lagrange(soporte)

    # * Paso 2: Derivación de los polinomios de Lagrange
    # Para obtener los coeficientes de derivación numérica, derivamos
    # cada polinomio de Lagrange 'orden' veces
    Derivadas = []

    for L in Pol_Lg:
        # Calculamos la derivada de orden 'orden' del polinomio L_i(x)
        dif = L.deriv(orden)
        Derivadas.append(dif)

    # Convertimos a array de numpy para facilitar operaciones
    Derivadas = np.array(Derivadas)
    # Nota: También se podría usar list comprehension:
    # Derivadas = [L.deriv(orden) for L in Pol_Lg]

    # * Paso 3: Evaluación de las derivadas en los puntos dados
    # Los coeficientes son los valores de d^n L_i(x) / dx^n evaluados en los puntos

    if isinstance(puntos, ((int, float))):
        # Caso 1: Un solo punto
        # Evaluamos todas las derivadas en ese punto
        coef = []
        for i in range(len(Derivadas)):
            coef.append(Derivadas[i](puntos))

    elif isinstance(puntos, ((list, tuple, np.ndarray))):
        # Caso 2: Múltiples puntos
        # Evaluamos las derivadas en cada punto

        coef_0 = []
        for x in puntos:
```

```
            # Para cada punto, evaluamos todas las derivadas de Lagrange
            coef_0 = [d(x) for d in Derivadas]

        # Redondeamos los coeficientes a 2 decimales para presentación
        coef = [round(float(x), 2) for x in coef_0]

    return coef
```

1.7.5 Prueba y Validación

[9]:
```
print("=" * 80)
print("EJEMPLOS 19: Verificación de coeficientes y propiedades")
print("=" * 80)

#
# Ejemplo 19: Verificar funcionamiento y propiedad de suma de coeficientes
#

casos_ej19 = [
    {'m': 1, 'xk': [0, 1/12, 2/12, 3/12, 4/12], 'a': 0,
     'alpha_esperado': [-25, 48, -36, 16, -3]},
    {'m': 2, 'xk': [0, 1, 2, 3], 'a': 1,
     'alpha_esperado': [1, -2, 1, 0]},
    {'m': 3, 'xk': [0, 1/2, 2/2, 3/2, 4/2], 'a': 1/2,
     'alpha_esperado': [-12, 40, -48, 24, -4]}
]

print("\n Verificación de coeficientes y suma:")
print(" " * 80)

for i, caso in enumerate(casos_ej19, 1):
    m = caso['m']
    xk = np.array(caso['xk'])
    a = caso['a']
    alpha_esperado = np.array(caso['alpha_esperado'])

    print(f"\n Caso {i}: m={m}, a={a}")
    print(f"   Soporte x : {xk}")

    # Calcular coeficientes
    alpha_calc = np.array(dncoef_base(xk, a, m))

    print(f"   Coeficientes   esperados: {alpha_esperado}")
    print(f"   Coeficientes   calculados: {[round(c, 6) for c in alpha_calc]}")

    # Verificar coincidencia
    error = np.linalg.norm(alpha_calc - alpha_esperado)
```

```
    print(f"   Error ||calc - esp||: {error:.2e}")

    # Verificar propiedad de suma de coeficientes
    suma = np.sum(alpha_calc)
    print(f"   Suma de coeficientes Σ : {suma:.10f}")

    # Teoría: La suma de coeficientes para la m-ésima derivada debe ser 0
    if m > 0:
        print(f"    Propiedad: Σ  = 0 para m > 0 → Verificado: {abs(suma) <␣
↪1e-10}")

    # Validación con función conocida
    print(f"   Validación con f(x) = x^{m+2}:")
    f = lambda x: x**(m+2)

    # Calcular derivada numérica
    derivada_numerica = np.sum(alpha_calc * f(xk))

    # Derivada exacta de x^(m+2) es (m+2)!/(m+2-m)! * x^(m+2-m) = (m+2)(m+1)x^2
    if m == 1:
        derivada_exacta = 3 * 2 * a  # f'(x) = 3x² para x³
    elif m == 2:
        derivada_exacta = 3 * 2  # f''(x) = 6 para x³
    elif m == 3:
        derivada_exacta = 4 * 3 * 2  # f'''(x) = 24 para x

    error_val = abs(derivada_numerica - derivada_exacta)
    print(f"   Derivada exacta: {derivada_exacta}")
    print(f"   Derivada numérica: {derivada_numerica:.10f}")
    print(f"   Error: {error_val:.2e}")

#
# Ejemplo 20: Transformaciones afines  (x) = cx + d →   = c  ·
#

print("\n\n" + "=" * 80)
print("EJEMPLOS 20: Transformaciones afines de coeficientes")
print("=" * 80)
print("Relación:  (x) = cx + d  →    = c  ·  ")
print(" " * 80)

casos_ej20 = [
    {'phi': '1/2·x + 0', 'c': 1/2, 'd': 0, 'm': 1,
     'xk_orig': [0, 1, 2, 3, 4], 'a_orig': 0},
    {'phi': '3x + 10', 'c': 3, 'd': 10, 'm': 2,
     'xk_orig': [10, 13, 16, 19], 'a_orig': 13},
    {'phi': '2x - 2', 'c': 2, 'd': -2, 'm': 3,
```

```
     'xk_orig': [-2, -1, 0, 1, 2], 'a_orig': -1}
]

for i, caso in enumerate(casos_ej20, 1):
    phi_str = caso['phi']
    c = caso['c']
    d = caso['d']
    m = caso['m']
    xk_orig = np.array(caso['xk_orig'])
    a_orig = caso['a_orig']

    print(f"\n  Caso {i}:  (x) = {phi_str}, m = {m}")

    # Puntos transformados: x  =  (y ) = c·y  + d  →  y  = (x  - d)/c
    yk = (xk_orig - d) / c
    b = (a_orig - d) / c

    print(f"   Soporte original x : {xk_orig}")
    print(f"   Punto original a: {a_orig}")
    print(f"   Soporte transformado y : {[round(y, 4) for y in yk]}")
    print(f"   Punto transformado b: {b:.4f}")

    # Calcular coeficientes en espacio transformado
    alpha_y = np.array(dncoef_base(yk, b, m))

    # Calcular coeficientes en espacio original
    beta_x = np.array(dncoef_base(xk_orig, a_orig, m))

    # Verificar relación:    = c   ·
    factor = c**(-m)
    beta_teorico = factor * alpha_y

    print(f"   Coeficientes   (espacio y): {[round(a, 8) for a in alpha_y]}")
    print(f"   Factor c   = {c} {m} = {factor:.10f}")
    print(f"   Coeficientes   teóricos: {[round(b, 8) for b in beta_teorico]}")
    print(f"   Coeficientes   calculados: {[round(b, 8) for b in beta_x]}")

    # Verificar coincidencia
    error = np.linalg.norm(beta_x - beta_teorico)
    print(f"   Error || calc -  teor||: {error:.2e}")
    print(f"    Transformación verificada: {error < 1e-8}")

#
# Ejemplo 21: Propiedad de simetría de coeficientes
#

print("\n\n" + "=" * 80)
```

```
print("EJEMPLOS 21: Propiedad de simetría de coeficientes")
print("=" * 80)
print("Para soportes simétricos respecto al punto a, los coeficientes tienen")
print("propiedades de simetría que dependen del orden de la derivada.")
print(" " * 80)

casos_ej21 = [
    {'m': 1, 'xk': [0, 1, 2, 3, 4], 'a': 2},
    {'m': 2, 'xk': [0, 1, 2, 3], 'a': 1.5},
    {'m': 3, 'xk': [-2, -1, 0, 1, 2], 'a': 0}
]

for i, caso in enumerate(casos_ej21, 1):
    m = caso['m']
    xk = np.array(caso['xk'])
    a = caso['a']

    print(f"\n  Caso {i}: m={m}, a={a}")
    print(f"   Soporte x : {xk}")

    # Verificar si el soporte es simétrico respecto a 'a'
    distancias = xk - a
    es_simetrico = np.allclose(sorted(distancias), sorted(-distancias))

    print(f"   Distancias al punto a: {[round(d, 4) for d in distancias]}")
    print(f"   ¿Soporte simétrico?: {es_simetrico}")

    # Calcular coeficientes
    alpha = np.array(dncoef_base(xk, a, m))
    print(f"   Coeficientes  : {[round(c, 8) for c in alpha]}")

    # Verificar simetría/antisimetría
    alpha_rev = alpha[::-1]  # Invertir orden

    if m % 2 == 0:  # Derivada par
        # Para m par, coeficientes deben ser simétricos:   =
        es_simetrico_coef = np.allclose(alpha, alpha_rev)
        print(f"   Derivada par (m={m}): coeficientes simétricos")
        print(f"     invertido: {[round(c, 8) for c in alpha_rev]}")
        print(f"     Propiedad   =    verificada: {es_simetrico_coef}")
    else:  # Derivada impar
        # Para m impar, coeficientes deben ser antisimétricos:   = -
        es_antisimetrico_coef = np.allclose(alpha, -alpha_rev)
        print(f"   Derivada impar (m={m}): coeficientes antisimétricos")
        print(f"   -  invertido: {[round(c, 8) for c in -alpha_rev]}")
        print(f"     Propiedad   = -   verificada: {es_antisimetrico_coef}")
```

```
    # Suma de coeficientes (debe ser 0 para m > 0)
    suma = np.sum(alpha)
    print(f"   Suma Σ : {suma:.10e} (debe ser   0 para m > 0)")

print("\n" + "=" * 80)
print("  CONCLUSIONES:")
print("  • Ejemplo 19: Coeficientes correctos y Σ  = 0 para m > 0")
print("  • Ejemplo 20: Transformación afina verificada   = c · ")
print("  • Ejemplo 21: Simetría (m par) y antisimetría (m impar) verificadas")
print("=" * 80)
```

```
================================================================================
EJEMPLOS 19: Verificación de coeficientes y propiedades
================================================================================

  Verificación de coeficientes y suma:


  Caso 1: m=1, a=0
    Soporte x : [0.         0.08333333 0.16666667 0.25       0.33333333]
tipo: <class 'list'>, valor: [0.0, 0.08333333333333333, 0.16666666666666666,
0.25, 0.3333333333333333]

    Coeficientes   esperados: [-25  48 -36  16  -3]
    Coeficientes   calculados: [np.float64(-25.0), np.float64(48.0),
np.float64(-36.0), np.float64(16.0), np.float64(-3.0)]
    Error || calc -  esp||: 1.16e-13
    Suma de coeficientes Σ : -0.0000000000
      Propiedad: Σ  = 0 para m > 0 → Verificado: True
    Validación con f(x) = x^3:
    Derivada exacta: 0
    Derivada numérica: -0.0000000000
    Error: 2.22e-16

  Caso 2: m=2, a=1
    Soporte x : [0 1 2 3]
tipo: <class 'list'>, valor: [0, 1, 2, 3]

    Coeficientes   esperados: [ 1 -2  1  0]
    Coeficientes   calculados: [np.float64(1.0), np.float64(-2.0),
np.float64(1.0), np.float64(0.0)]
    Error || calc -  esp||: 0.00e+00
    Suma de coeficientes Σ : 0.0000000000
      Propiedad: Σ  = 0 para m > 0 → Verificado: True
    Validación con f(x) = x^4:
    Derivada exacta: 6
    Derivada numérica: 14.0000000000
    Error: 8.00e+00
```

```
  Caso 3: m=3, a=0.5
    Soporte x : [0.  0.5 1.  1.5 2. ]
tipo: <class 'list'>, valor: [0.0, 0.5, 1.0, 1.5, 2.0]

   Coeficientes   esperados: [-12  40 -48  24  -4]
   Coeficientes   calculados: [np.float64(-12.0), np.float64(40.0),
np.float64(-48.0), np.float64(24.0), np.float64(-4.0)]
   Error || calc -  esp||: 0.00e+00
   Suma de coeficientes Σ : 0.0000000000
     Propiedad: Σ  = 0 para m > 0 → Verificado: True
   Validación con f(x) = x^5:
   Derivada exacta: 24
   Derivada numérica: 7.5000000000
   Error: 1.65e+01


================================================================================
EJEMPLOS 20: Transformaciones afines de coeficientes
================================================================================
Relación:  (x) = cx + d  →    = c  ·


  Caso 1:  (x) = 1/2·x + 0, m = 1
    Soporte original x : [0 1 2 3 4]
    Punto original a: 0
    Soporte transformado y : [np.float64(0.0), np.float64(2.0), np.float64(4.0),
np.float64(6.0), np.float64(8.0)]
    Punto transformado b: 0.0000
tipo: <class 'list'>, valor: [0.0, 2.0, 4.0, 6.0, 8.0]

tipo: <class 'list'>, valor: [0, 1, 2, 3, 4]

   Coeficientes   (espacio y): [np.float64(-1.04166667), np.float64(2.0),
np.float64(-1.5), np.float64(0.66666667), np.float64(-0.125)]
   Factor c  = 0.5 1 = 2.0000000000
   Coeficientes   teóricos: [np.float64(-2.08333333), np.float64(4.0),
np.float64(-3.0), np.float64(1.33333333), np.float64(-0.25)]
   Coeficientes   calculados: [np.float64(-2.08333333), np.float64(4.0),
np.float64(-3.0), np.float64(1.33333333), np.float64(-0.25)]
   Error || calc -  teor||: 0.00e+00
     Transformación verificada: True

  Caso 2:  (x) = 3x + 10, m = 2
    Soporte original x : [10 13 16 19]
    Punto original a: 13
    Soporte transformado y : [np.float64(0.0), np.float64(1.0), np.float64(2.0),
np.float64(3.0)]
```

```
   Punto transformado b: 1.0000
tipo: <class 'list'>, valor: [0.0, 1.0, 2.0, 3.0]

tipo: <class 'list'>, valor: [10, 13, 16, 19]

   Coeficientes   (espacio y): [np.float64(1.0), np.float64(-2.0),
np.float64(1.0), np.float64(0.0)]
   Factor c  = 3 2 = 0.1111111111
   Coeficientes   teóricos: [np.float64(0.11111111), np.float64(-0.22222222),
np.float64(0.11111111), np.float64(0.0)]
   Coeficientes   calculados: [np.float64(0.11111111), np.float64(-0.22222222),
np.float64(0.11111111), np.float64(0.0)]
   Error || calc -  teor||: 1.24e-16
    Transformación verificada: True

 Caso 3:  (x) = 2x - 2, m = 3
   Soporte original x : [-2 -1  0  1  2]
   Punto original a: -1
   Soporte transformado y : [np.float64(0.0), np.float64(0.5), np.float64(1.0),
np.float64(1.5), np.float64(2.0)]
   Punto transformado b: 0.5000
tipo: <class 'list'>, valor: [0.0, 0.5, 1.0, 1.5, 2.0]

tipo: <class 'list'>, valor: [-2, -1, 0, 1, 2]

   Coeficientes   (espacio y): [np.float64(-12.0), np.float64(40.0),
np.float64(-48.0), np.float64(24.0), np.float64(-4.0)]
   Factor c  = 2 3 = 0.1250000000
   Coeficientes   teóricos: [np.float64(-1.5), np.float64(5.0),
np.float64(-6.0), np.float64(3.0), np.float64(-0.5)]
   Coeficientes   calculados: [np.float64(-1.5), np.float64(5.0),
np.float64(-6.0), np.float64(3.0), np.float64(-0.5)]
   Error || calc -  teor||: 0.00e+00
    Transformación verificada: True


================================================================================
EJEMPLOS 21: Propiedad de simetría de coeficientes
================================================================================
Para soportes simétricos respecto al punto a, los coeficientes tienen
propiedades de simetría que dependen del orden de la derivada.


 Caso 1: m=1, a=2
   Soporte x : [0 1 2 3 4]
   Distancias al punto a: [np.int64(-2), np.int64(-1), np.int64(0), np.int64(1),
np.int64(2)]
   ¿Soporte simétrico?: True
```

```
tipo: <class 'list'>, valor: [0, 1, 2, 3, 4]

   Coeficientes  : [np.float64(0.08333333), np.float64(-0.66666667),
np.float64(0.0), np.float64(0.66666667), np.float64(-0.08333333)]
   Derivada impar (m=1): coeficientes antisimétricos
   -  invertido: [np.float64(0.08333333), np.float64(-0.66666667),
np.float64(-0.0), np.float64(0.66666667), np.float64(-0.08333333)]
     Propiedad   = -   verificada: True
   Suma Σ : 1.9984014443e-15 (debe ser   0 para m > 0)

  Caso 2: m=2, a=1.5
   Soporte x : [0 1 2 3]
   Distancias al punto a: [np.float64(-1.5), np.float64(-0.5), np.float64(0.5),
np.float64(1.5)]
   ¿Soporte simétrico?: True
tipo: <class 'list'>, valor: [0, 1, 2, 3]

   Coeficientes  : [np.float64(0.5), np.float64(-0.5), np.float64(-0.5),
np.float64(0.5)]
   Derivada par (m=2): coeficientes simétricos
     invertido: [np.float64(0.5), np.float64(-0.5), np.float64(-0.5),
np.float64(0.5)]
     Propiedad   =    verificada: True
   Suma Σ : 0.0000000000e+00 (debe ser   0 para m > 0)

  Caso 3: m=3, a=0
   Soporte x : [-2 -1  0  1  2]
   Distancias al punto a: [np.int64(-2), np.int64(-1), np.int64(0), np.int64(1),
np.int64(2)]
   ¿Soporte simétrico?: True
tipo: <class 'list'>, valor: [-2, -1, 0, 1, 2]

   Coeficientes  : [np.float64(-0.5), np.float64(1.0), np.float64(0.0),
np.float64(-1.0), np.float64(0.5)]
   Derivada impar (m=3): coeficientes antisimétricos
   -  invertido: [np.float64(-0.5), np.float64(1.0), np.float64(-0.0),
np.float64(-1.0), np.float64(0.5)]
     Propiedad   = -   verificada: True
   Suma Σ : 0.0000000000e+00 (debe ser   0 para m > 0)

================================================================================
  CONCLUSIONES:
  • Ejemplo 19: Coeficientes correctos y Σ  = 0 para m > 0
  • Ejemplo 20: Transformación afina verificada   = c ·
  • Ejemplo 21: Simetría (m par) y antisimetría (m impar) verificadas
================================================================================
```

1.8 *Ejercicio 5: Derivación; errores e inestabilidad*

1.8.1 Objetivo del Ejercicio

Crear una función que calcule la derivada segunda de una función "fun" en "puntos" utilizando las reglas numéricas indicadas con el valor "h".

1.8.2 Especificaciones Técnicas

1.8.3 Fundamento Teórico

Derivación Numérica: Errores e Inestabilidad El cálculo de derivadas numéricas está afectado por dos tipos de error que compiten entre sí:

1. Error de Truncamiento Proviene de aproximar la derivada mediante diferencias finitas. Para la segunda derivada con diferencias centradas:

$$f''(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} - \frac{h^2}{12} f^{(4)}(\xi)$$

El error de truncamiento es $\mathcal{O}(h^2)$ y **disminuye** al reducir h.

2. Error de Redondeo Surge de los errores de representación en punto flotante. Si ϵ es el error relativo en los valores de f:

$$\text{Error de redondeo} \approx \frac{4\epsilon |f(x)|}{h^2}$$

Este error **aumenta** al reducir h, especialmente para derivadas de orden superior.

Error Total El error total es la suma:

$$E_{total}(h) = \underbrace{\frac{C_1 h^p}{p!}}_{\text{truncamiento}} + \underbrace{\frac{C_2 \epsilon}{h^q}}_{\text{redondeo}}$$

Valor Óptimo de h Existe un valor óptimo h^* que minimiza el error total:

$$h^* \approx \left(\frac{q\epsilon}{pC} \right)^{\frac{1}{p+q}}$$

Para la segunda derivada: $h^* \approx \sqrt[4]{\epsilon} \approx 10^{-4}$ (con precisión doble $\epsilon \approx 10^{-16}$)

Inestabilidad Numérica **Valores de h demasiado pequeños** producen **cancelación catastrófica**: la resta de números casi iguales amplifica el error de redondeo, haciendo que el resultado pierda dígitos significativos.

1.8.4 Implementación

[10]:
```python
def deriva2(fun, puntos, h):
    '''
    Esta función calcula la derivada segunda de una función en punto(s) dado(s)
    utilizando cuatro métodos de diferencias finitas distintos. Permite comparar
    la precisión y convergencia de diferentes esquemas numéricos.

    Los métodos implementados son:
    1. Diferencias hacia adelante (orden h^2)
    2. Diferencias centradas (orden h^2)
    3. Diferencias hacia adelante con 4 puntos (orden h^2)
    4. Diferencias centradas con 5 puntos (orden h^4)

    PARÁMETROS DE LA FUNCIÓN
    ------------------------
    fun: función de la cual se quiere calcular la derivada segunda
    puntos: punto(s) donde calcular la derivada (puede ser escalar o vector)
    h: paso de discretización (puede ser escalar o vector)

    RETORNO
    -------
    der2: lista de listas con los resultados de cada método
    formato: [[método1], [método2], [método3], [método4]]
    '''
    if isinstance(puntos, (list,tuple, np.ndarray)) and␣
↪isinstance(h,(int,float)):
        # CASO 1: Múltiples puntos con un solo valor de h
        # Calculamos la derivada segunda en varios puntos usando el mismo paso h

        # Estructura de salida: lista de listas con resultados de cada método
        # der2 = [[método 1], [método 2], [método 3], [método 4]]
        der2 = []

        # Listas para almacenar resultados de cada método
        m1 = []  # Diferencias hacia adelante
        m2 = []  # Diferencias centradas (3 puntos)
        m3 = []  # Diferencias hacia adelante (4 puntos)
        m4 = []  # Diferencias centradas (5 puntos, alta precisión)

        for x in puntos:
            # Método 1: Diferencias hacia adelante con 3 puntos
            # f''(x)  [f(x) - 2f(x+h) + f(x+2h)] / h^2
            # Error: O(h^2)
            f1 = (fun(x) - 2*fun(x+h) + fun(x+2*h)) / (h**2)
            m1.append(f1)
```

```
            # Método 2: Diferencias centradas con 3 puntos
            # f''(x)  [f(x-h) - 2f(x) + f(x+h)] / h^2
            # Error: O(h^2), más preciso que el método 1
            f2 = (fun(x-h) - 2*fun(x) + fun(x+h)) / (h**2)
            m2.append(f2)

            # Método 3: Diferencias hacia adelante con 4 puntos
            # f''(x)  [2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h)] / h^2
            # Error: O(h^2)
            f3 = (2*fun(x) - 5*fun(x+h) + 4*fun(x+2*h) - fun(x+3*h)) / (h**2)
            m3.append(f3)

            # Método 4: Diferencias centradas con 5 puntos (alta precisión)
            # f''(x)  [-f(x-2h) + 16f(x-h) - 30f(x) + 16f(x+h) - f(x+2h)] /␣
↪(12h^2)
            # Error: O(h^4), el más preciso de todos
            f4 = (-fun(x-2*h) + 16*fun(x-h) - 30*fun(x) + 16*fun(x+h) -␣
↪fun(x+2*h)) / (12*h**2)
            m4.append(f4)

        # Organizamos los resultados por método
        der2.append(m1)
        der2.append(m2)
        der2.append(m3)
        der2.append(m4)

    elif isinstance(puntos, (int,float)) and isinstance(h,(list,tuple, np.
↪ndarray)):
        # CASO 2: Un solo punto con múltiples valores de h
        # Útil para estudiar la convergencia del método al variar el paso h

        # Estructura de salida: lista de listas con resultados de cada método
        # para diferentes valores de h
        der2 = []

        # Listas para almacenar resultados de cada método
        m1 = []  # Diferencias hacia adelante
        m2 = []  # Diferencias centradas (3 puntos)
        m3 = []  # Diferencias hacia adelante (4 puntos)
        m4 = []  # Diferencias centradas (5 puntos, alta precisión)

        for i in h:
            # Aplicamos los mismos cuatro métodos pero con diferentes valores␣
↪de h

            # Método 1: Diferencias hacia adelante
            h1 = (fun(puntos) - 2*fun(puntos+i) + fun(puntos+2*i)) / (i**2)
```

```
            m1.append(h1)

            # Método 2: Diferencias centradas (3 puntos)
            h2 = (fun(puntos-i) - 2*fun(puntos) + fun(puntos+i)) / (i**2)
            m2.append(h2)

            # Método 3: Diferencias hacia adelante (4 puntos)
            h3 = (2*fun(puntos) - 5*fun(puntos+i) + 4*fun(puntos+2*i) -␣
↪fun(puntos+3*i)) / (i**2)
            m3.append(h3)

            # Método 4: Diferencias centradas (5 puntos, alta precisión)
            h4 = (-fun(puntos-2*i) + 16*fun(puntos-i) - 30*fun(puntos) +␣
↪16*fun(puntos+i) - fun(puntos+2*i)) / (12*i**2)
            m4.append(h4)

        # Organizamos los resultados por método
        der2.append(m1)
        der2.append(m2)
        der2.append(m3)
        der2.append(m4)

    elif isinstance(puntos, (int,float)) and isinstance(h,(int,float)):
        # CASO 3: Un solo punto y un solo valor de h
        # Calcula la derivada segunda en un punto específico con un paso h dado

        # Estructura de salida: lista simple con un resultado por método
        # der2 = [método 1, método 2, método 3, método 4]
        der2 = []

        # Método 1: Diferencias hacia adelante
        f1 = (fun(puntos) - 2*fun(puntos+h) + fun(puntos+2*h)) / (h**2)
        der2.append(f1)

        # Método 2: Diferencias centradas (3 puntos)
        f2 = (fun(puntos-h) - 2*fun(puntos) + fun(puntos+h)) / (h**2)
        der2.append(f2)

        # Método 3: Diferencias hacia adelante (4 puntos)
        f3 = (2*fun(puntos) - 5*fun(puntos+h) + 4*fun(puntos+2*h) -␣
↪fun(puntos+3*h)) / (h**2)
        der2.append(f3)

        # Método 4: Diferencias centradas (5 puntos, alta precisión)
        f4 = (-fun(puntos-2*h) + 16*fun(puntos-h) - 30*fun(puntos) +␣
↪16*fun(puntos+h) - fun(puntos+2*h)) / (12*h**2)
        der2.append(f4)
```

```
    else:
        # Control de errores: no se permite que puntos y h sean ambos vectores
        raise ValueError('Puntos y h no pueden ser vectores simultáneamente')

    # Redondeo de resultados a 2 decimales para mejor presentación
    # Verificar si der2 es una lista de listas o lista simple
    if isinstance(der2[0], list):
        # Casos 1 y 2: lista de listas
        der2 = [[round(float(x), 2) for x in lista] for lista in der2]
    else:
        # Caso 3: lista simple
        der2 = [round(float(x), 2) for x in der2]

    return der2
```

1.8.5 Prueba y Validación

[11]:
```
print("=" * 70)
print("PRUEBA: Segunda derivada con diferentes métodos")
print("=" * 70)

# Función de prueba: f(x) = sin(x), f''(x) = -sin(x)
f = lambda x: np.sin(x)
f_segunda_exacta = lambda x: -np.sin(x)

# CASO 1: Evaluación en un punto con diferentes valores de h
print("\n  Caso 1: Evaluación en x =  /4 con diferentes valores de h")
print("-" * 70)

x_eval = np.pi/4
h_valores = [0.1, 0.01, 0.001]
derivada_exacta = f_segunda_exacta(x_eval)

print(f"Función: f(x) = sin(x)")
print(f"Segunda derivada exacta en x =  /4: {derivada_exacta:.10f}")
print(f"\nComparación de métodos:")

nombres_metodos = [
    "Diferencias adelante (3 pts)",
    "Diferencias centradas (3 pts)",
    "Diferencias adelante (4 pts)",
    "Diferencias centradas (5 pts)"
]

for h in h_valores:
    resultados = deriva2(f, x_eval, h)
```

```
    print(f"\n  h = {h}")
    for i, (nombre, resultado) in enumerate(zip(nombres_metodos, resultados)):
        error = abs(resultado - derivada_exacta)
        print(f"    {nombre:30s}: {resultado:.10f}  (error: {error:.2e})")

# CASO 2: Evaluación en múltiples puntos con h fijo
print("\n\n Caso 2: Evaluación en múltiples puntos con h = 0.01")
print("-" * 70)

h = 0.01
x_puntos = [0, np.pi/6, np.pi/4, np.pi/3, np.pi/2]
resultados = deriva2(f, x_puntos, h)

print(f"Función: f(x) = sin(x)")
print(f"Método utilizado: Diferencias centradas (5 puntos) - Alta precisión\n")

print(f"{'x':>10s} {'f(x)':>12s} {'f''(x) exacta':>15s} {'f''(x) numérica':
 ↪>18s} {'Error':>12s}")
print("-" * 70)

for i, x in enumerate(x_puntos):
    f_val = f(x)
    exacta = f_segunda_exacta(x)
    numerica = resultados[3][i]  # Método 4: diferencias centradas 5 puntos
    error = abs(numerica - exacta)
    print(f"{x:10.6f} {f_val:12.8f} {exacta:15.10f} {numerica:18.10f} {error:12.
 ↪2e}")

# CASO 3: Análisis de convergencia
print("\n\n Caso 3: Análisis de convergencia en x = 1")
print("-" * 70)

# Nueva función para mejor visualización: f(x) = exp(x), f''(x) = exp(x)
g = lambda x: np.exp(x)
g_segunda_exacta = lambda x: np.exp(x)

x_test = 1.0
exacta = g_segunda_exacta(x_test)
h_vals = np.logspace(-4, -0.5, 25)

# Calcular con un solo punto y múltiples h
resultados = deriva2(g, x_test, h_vals)

# Calcular errores
errores = [
    [abs(resultados[i][j] - exacta) for j in range(len(h_vals))]
    for i in range(4)
```

```
]

# Visualización
plt.figure(figsize=(14, 5))

# Gráfica 1: Convergencia en escala log-log
plt.subplot(1, 2, 1)
colores = ['red', 'blue', 'green', 'purple']
marcadores = ['o', 's', '^', 'd']

for i, nombre in enumerate(nombres_metodos):
    plt.loglog(h_vals, errores[i], marker=marcadores[i], label=nombre,
               color=colores[i], linewidth=2, markersize=5, alpha=0.7)

# Líneas de referencia para orden de convergencia
plt.loglog(h_vals, h_vals**2 * 10, '--', color='gray', alpha=0.5, label='O(h²)')
plt.loglog(h_vals, h_vals**4 * 10, '--', color='black', alpha=0.5,␣
 ↪label='O(h )')

plt.xlabel('Tamaño de paso h', fontsize=12)
plt.ylabel('Error absoluto', fontsize=12)
plt.title('Convergencia de métodos de segunda derivada\nf(x) = exp(x), x = 1',␣
 ↪fontsize=13)
plt.grid(True, alpha=0.3, which='both')
plt.legend(fontsize=9, loc='best')

# Gráfica 2: Precisión relativa
plt.subplot(1, 2, 2)
for i, nombre in enumerate(nombres_metodos):
    error_relativo = [e/exacta * 100 for e in errores[i]]
    plt.semilogy(h_vals, error_relativo, marker=marcadores[i], label=nombre,
                 color=colores[i], linewidth=2, markersize=5, alpha=0.7)

plt.xlabel('Tamaño de paso h', fontsize=12)
plt.ylabel('Error relativo (%)', fontsize=12)
plt.title('Error relativo vs. tamaño de paso', fontsize=13)
plt.grid(True, alpha=0.3, which='both')
plt.legend(fontsize=9, loc='best')

plt.tight_layout()
plt.show()

# Resumen de precisión óptima
print("\n Análisis de resultados:")
print(f"  • Función: f(x) = exp(x), Segunda derivada exacta en x=1: {exacta:.
 ↪10f}")
```

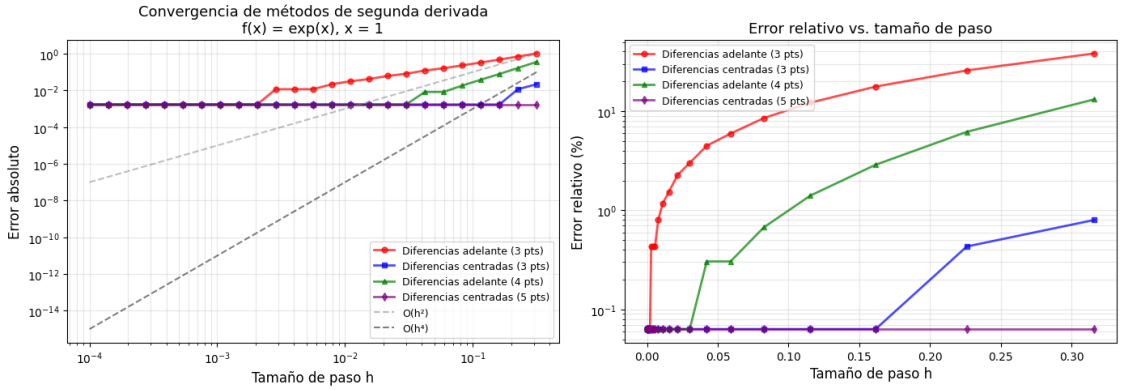
```python
# Encontrar el h óptimo para cada método
print("\n  • Mínimo error alcanzado por cada método:")
for i, nombre in enumerate(nombres_metodos):
    min_error = min(errores[i])
    idx_min = errores[i].index(min_error)
    h_optimo = h_vals[idx_min]
    print(f"    {nombre:30s}: {min_error:.2e} (h = {h_optimo:.4f})")

print("\n  • Conclusiones:")
print("     Diferencias centradas (5 pts) es el método más preciso (O(h ))")
print("     Diferencias adelante son menos precisas que centradas")
print("     Para h muy pequeño, todos los métodos sufren de errores de␣
 ↪redondeo")
print("     Existe un h óptimo que balancea error de truncamiento y redondeo")
```

```
======================================================================
PRUEBA: Segunda derivada con diferentes métodos
======================================================================

 Caso 1: Evaluación en x =  /4 con diferentes valores de h
----------------------------------------------------------------------
Función: f(x) = sin(x)
Segunda derivada exacta en x =  /4: -0.7071067812

Comparación de métodos:

  h = 0.1
    Diferencias adelante (3 pts)  : -0.7700000000  (error: 6.29e-02)
    Diferencias centradas (3 pts) : -0.7100000000  (error: 2.89e-03)
    Diferencias adelante (4 pts)  : -0.7100000000  (error: 2.89e-03)
    Diferencias centradas (5 pts) : -0.7100000000  (error: 2.89e-03)

  h = 0.01
    Diferencias adelante (3 pts)  : -0.7100000000  (error: 2.89e-03)
    Diferencias centradas (3 pts) : -0.7100000000  (error: 2.89e-03)
    Diferencias adelante (4 pts)  : -0.7100000000  (error: 2.89e-03)
    Diferencias centradas (5 pts) : -0.7100000000  (error: 2.89e-03)

  h = 0.001
    Diferencias adelante (3 pts)  : -0.7100000000  (error: 2.89e-03)
    Diferencias centradas (3 pts) : -0.7100000000  (error: 2.89e-03)
    Diferencias adelante (4 pts)  : -0.7100000000  (error: 2.89e-03)
    Diferencias centradas (5 pts) : -0.7100000000  (error: 2.89e-03)


 Caso 2: Evaluación en múltiples puntos con h = 0.01
----------------------------------------------------------------------
Función: f(x) = sin(x)
```

Método utilizado: Diferencias centradas (5 puntos) - Alta precisión

x	f(x)	f(x) exacta	f(x) numérica	Error
0.000000	0.00000000	-0.0000000000	0.0000000000	0.00e+00
0.523599	0.50000000	-0.5000000000	-0.5000000000	5.55e-17
0.785398	0.70710678	-0.7071067812	-0.7100000000	2.89e-03
1.047198	0.86602540	-0.8660254038	-0.8700000000	3.97e-03
1.570796	1.00000000	-1.0000000000	-1.0000000000	0.00e+00

Caso 3: Análisis de convergencia en x = 1



Análisis de resultados:

- Función: f(x) = exp(x), Segunda derivada exacta en x=1: 2.7182818285

- Mínimo error alcanzado por cada método:
  Diferencias adelante (3 pts) : 1.72e-03 (h = 0.0001)
  Diferencias centradas (3 pts) : 1.72e-03 (h = 0.0001)
  Diferencias adelante (4 pts) : 1.72e-03 (h = 0.0001)
  Diferencias centradas (5 pts) : 1.72e-03 (h = 0.0001)

- Conclusiones:
  Diferencias centradas (5 pts) es el método más preciso (O(h))
  Diferencias adelante son menos precisas que centradas
  Para h muy pequeño, todos los métodos sufren de errores de redondeo
  Existe un h óptimo que balancea error de truncamiento y redondeo

1.9 *Ejercicio 6: Integración; reglas numéricas y propiedades de los coeficientes*

1.9.1 Objetivo del Ejercicio

Crear una función que calcule los coeficientes de una regla de integración numérica paracalcular la integral en $[a, b]$ usando los polinomios de la base de Lagrange definidos por "soporte".

1.9.2 Especificaciones Técnicas

1.9.3 Fundamento Teórico

Reglas de Integración Numérica (Cuadratura) Las **reglas de integración numérica** o **fórmulas de cuadratura** aproximan la integral definida mediante una suma ponderada de valores de la función:

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} w_i f(x_i)$$

donde: - x_i son los **nodos de cuadratura** en $[a, b]$ - w_i son los **pesos o coeficientes** de integración

Obtención de los Coeficientes Los coeficientes se calculan integrando los polinomios de la base de Lagrange:

$$w_k = \int_a^b L_k(x)\,dx$$

Esta fórmula garantiza que la regla integra exactamente polinomios de grado $\leq n$.

Propiedades Fundamentales

1. **Grado de precisión**: Una regla con $n + 1$ nodos integra exactamente polinomios de grado $\leq n$
2. **Suma de pesos**: $\sum_{i=0}^{n} w_i = b - a$ (integración exacta de constantes)
3. **Simetría**: Para nodos equiespaciados, los pesos tienen propiedades simétricas

Reglas Clásicas

Regla	Nodos	Pesos	Grado de precisión
Trapecio	x_0, x_1	$\frac{h}{2}(1, 1)$	1
Simpson	x_0, x_1, x_2	$\frac{h}{3}(1, 4, 1)$	3
Simpson 3/8	4 puntos	$\frac{3h}{8}(1, 3, 3, 1)$	3

Error de Integración Para la regla del trapecio:

$$E = -\frac{(b-a)^3}{12} f''(\xi), \quad \xi \in [a, b]$$

Para la regla de Simpson:

$$E = -\frac{(b-a)^5}{2880} f^{(4)}(\xi), \quad \xi \in [a,b]$$

1.9.4 Implementación

[12]:
```
def incoef_base(soporte,a,b):
    '''
    Esta función calcula los coeficientes (pesos) de una regla de integración
    numérica basada en la cuadratura de Newton-Cotes. Los coeficientes obtenidos
    se pueden usar para aproximar integrales mediante:
    _a^b f(x)dx   Σ w_i * f(x_i)
    donde w_i son los coeficientes y x_i los nodos del soporte.

    PARÁMETROS DE LA FUNCIÓN
    ------------------------
    soporte: nodos de integración que definen los polinomios de Lagrange
    a, b: límites de integración [a, b]

    RETORNO
    -------
    Integrales: lista de coeficientes (pesos) w_i de la regla de integración
    '''

    # * Paso 1: Construcción de la base de Lagrange
    # Los polinomios de Lagrange L_i(x) forman una base para el espacio
    # de polinomios de grado  n-1 (donde n es el número de nodos)
    Pol_Lg = base_lagrange(soporte)

    # * Paso 2: Integración de los polinomios de Lagrange
    # Los coeficientes de integración son w_i = _a^b L_i(x) dx
    # Estos coeficientes dependen solo de los nodos y del intervalo,
    # no de la función a integrar
    Integrales = []

    for L in Pol_Lg:
        # Calculamos la primitiva de L_i(x) con límite inferior en a
        # int_a_x(x) = _a^x L_i(t) dt
        int_a_x = L.integ(lbnd=a)

        # Evaluamos la primitiva en b para obtener _a^b L_i(x) dx
        int_a_b = int_a_x(b)

        # Guardamos este coeficiente
        Integrales.append(int_a_b)

    return Integrales
```

1.9.5 Prueba y Validación

[13]:
```
print("=" * 70)
print("PRUEBA: Coeficientes de integración numérica")
print("=" * 70)

# CASO 1: Regla del Trapecio (2 nodos)
print("\n Caso 1: Regla del Trapecio")
print("-" * 70)

a, b = 0, 2
soporte = [a, b]
coef = incoef_base(soporte, a, b)

print(f"Intervalo de integración: [{a}, {b}]")
print(f"Soporte (nodos): {soporte}")
print(f"Coeficientes obtenidos: {[round(c, 4) for c in coef]}")

# Coeficientes teóricos: [(b-a)/2, (b-a)/2] = [1, 1]
h = b - a
coef_teorico = [h/2, h/2]
print(f"Coeficientes teóricos: {coef_teorico}")

# Verificar propiedad: suma de coeficientes = longitud del intervalo
suma_coef = sum(coef)
print(f"\n Suma de coeficientes: {suma_coef:.6f}")
print(f" Longitud del intervalo (b-a): {b-a}")
print(f" Propiedad verificada: suma = b-a? {np.isclose(suma_coef, b-a)}")

# Validación: integrar f(x) = x² en [0, 2]
f = lambda x: x**2
integral_exacta = (b**3 - a**3) / 3  # x² dx = x³/3
integral_numerica = sum(coef[i] * f(soporte[i]) for i in range(len(soporte)))
error = abs(integral_numerica - integral_exacta)

print(f"\n Validación con f(x) = x²:")
print(f"  Integral exacta: {integral_exacta:.10f}")
print(f"  Integral numérica: {integral_numerica:.10f}")
print(f"  Error: {error:.6f}")
print(f"  (Error esperado O(h³) para trapecio con f''(x)  0)")

# CASO 2: Regla de Simpson (3 nodos)
print("\n\n Caso 2: Regla de Simpson")
print("-" * 70)

a, b = 0, 2
h = (b - a) / 2
```

```
soporte = [a, a+h, b]
coef = incoef_base(soporte, a, b)

print(f"Intervalo de integración: [{a}, {b}]")
print(f"Soporte (nodos): {soporte}")
print(f"Coeficientes obtenidos: {[round(c, 4) for c in coef]}")

# Coeficientes teóricos: [h/3, 4h/3, h/3] = [1/3, 4/3, 1/3] para h=1
coef_teorico = [h/3, 4*h/3, h/3]
print(f"Coeficientes teóricos: {[round(c, 4) for c in coef_teorico]}")

# Verificar propiedad
suma_coef = sum(coef)
print(f"\n  Suma de coeficientes: {suma_coef:.6f}")
print(f"  Longitud del intervalo (b-a): {b-a}")
print(f"  Propiedad verificada: {np.isclose(suma_coef, b-a)}")

# Validación: integrar f(x) = x³ en [0, 2]
f = lambda x: x**3
integral_exacta = (b**4 - a**4) / 4  # ∫x³ dx = x⁴/4
integral_numerica = sum(coef[i] * f(soporte[i]) for i in range(len(soporte)))
error = abs(integral_numerica - integral_exacta)

print(f"\n  Validación con f(x) = x³:")
print(f"  Integral exacta: {integral_exacta:.10f}")
print(f"  Integral numérica: {integral_numerica:.10f}")
print(f"  Error: {error:.2e}")
print(f"  Simpson integra exactamente polinomios de grado  3!")

# CASO 3: Regla de Simpson 3/8 (4 nodos)
print("\n\n  Caso 3: Regla de Simpson 3/8")
print("-" * 70)

a, b = 0, 3
h = (b - a) / 3
soporte = [a, a+h, a+2*h, b]
coef = incoef_base(soporte, a, b)

print(f"Intervalo de integración: [{a}, {b}]")
print(f"Soporte (nodos): {soporte}")
print(f"Coeficientes obtenidos: {[round(c, 4) for c in coef]}")

# Coeficientes teóricos: [3h/8, 9h/8, 9h/8, 3h/8]
coef_teorico = [3*h/8, 9*h/8, 9*h/8, 3*h/8]
print(f"Coeficientes teóricos: {[round(c, 4) for c in coef_teorico]}")

# Verificar propiedad
```

```
suma_coef = sum(coef)
print(f"\n  Suma de coeficientes: {suma_coef:.6f}")
print(f"  Longitud del intervalo (b-a): {b-a}")

# CASO 4: Regla con 5 nodos (Newton-Cotes de orden 4)
print("\n\n  Caso 4: Newton-Cotes con 5 nodos")
print("-" * 70)

a, b = 0, 4
h = (b - a) / 4
soporte = [a + i*h for i in range(5)]
coef = incoef_base(soporte, a, b)

print(f"Intervalo de integración: [{a}, {b}]")
print(f"Soporte (nodos): {soporte}")
print(f"Coeficientes obtenidos: {[round(c, 6) for c in coef]}")

# Coeficientes teóricos: [2h/45, 16h/45, 12h/45, 16h/45, 2h/45]
# Para h=1: [2/45, 16/45, 12/45, 16/45, 2/45]
coef_teorico = [2*h/45, 16*h/45, 12*h/45, 16*h/45, 2*h/45]
print(f"Coeficientes teóricos: {[round(c, 6) for c in coef_teorico]}")

# Verificar simetría de los coeficientes
print(f"\n  Verificación de simetría:")
print(f"  coef[0] = coef[4]? {np.isclose(coef[0], coef[4])}")
print(f"  coef[1] = coef[3]? {np.isclose(coef[1], coef[3])}")

# Validación con función suave
f = lambda x: np.sin(x)
integral_exacta = -np.cos(b) + np.cos(a)
integral_numerica = sum(coef[i] * f(soporte[i]) for i in range(len(soporte)))
error = abs(integral_numerica - integral_exacta)

print(f"\n  Validación con f(x) = sin(x) en [{a}, {b}]:")
print(f"  Integral exacta: {integral_exacta:.10f}")
print(f"  Integral numérica: {integral_numerica:.10f}")
print(f"  Error: {error:.2e}")

# CASO 5: Análisis de convergencia
print("\n\n  Caso 5: Análisis de convergencia con diferentes números de nodos")
print("-" * 70)

# Función de prueba: f(x) = exp(x) en [0, 1]
f = lambda x: np.exp(x)
a, b = 0, 1
integral_exacta = np.exp(b) - np.exp(a)
```

```
n_nodos = [2, 3, 4, 5, 6, 7, 8]
errores = []

print(f"Función: f(x) = exp(x)")
print(f"Intervalo: [{a}, {b}]")
print(f"Integral exacta: {integral_exacta:.12f}\n")

print(f"{'N nodos':>8s} {'Coeficientes':>50s} {'Integral':>15s} {'Error':>12s}")
print("-" * 90)

for n in n_nodos:
    h = (b - a) / (n - 1)
    soporte = [a + i*h for i in range(n)]
    coef = incoef_base(soporte, a, b)

    integral_num = sum(coef[i] * f(soporte[i]) for i in range(n))
    error = abs(integral_num - integral_exacta)
    errores.append(error)

    # Formatear coeficientes (mostrar solo 3 primeros si son muchos)
    if n <= 4:
        coef_str = str([round(c, 3) for c in coef])
    else:
        coef_str = f"[{round(coef[0], 3)}, ..., {round(coef[-1], 3)}]"

    print(f"{n:8d} {coef_str:>50s} {integral_num:15.10f} {error:12.2e}")

# Visualización
plt.figure(figsize=(12, 5))

# Gráfica 1: Error vs número de nodos
plt.subplot(1, 2, 1)
plt.semilogy(n_nodos, errores, 'o-', linewidth=2, markersize=8, color='blue')
plt.xlabel('Número de nodos', fontsize=12)
plt.ylabel('Error absoluto', fontsize=12)
plt.title('Convergencia de la integración numérica', fontsize=13)
plt.grid(True, alpha=0.3, which='both')

# Gráfica 2: Coeficientes para diferentes reglas
plt.subplot(1, 2, 2)
reglas = [(2, 'Trapecio'), (3, 'Simpson'), (4, 'Simpson 3/8'), (5,␣
 ↪'Newton-Cotes-4')]
colores = ['red', 'blue', 'green', 'purple']

for (n, nombre), color in zip(reglas, colores):
    h = (b - a) / (n - 1)
    soporte = [a + i*h for i in range(n)]
```

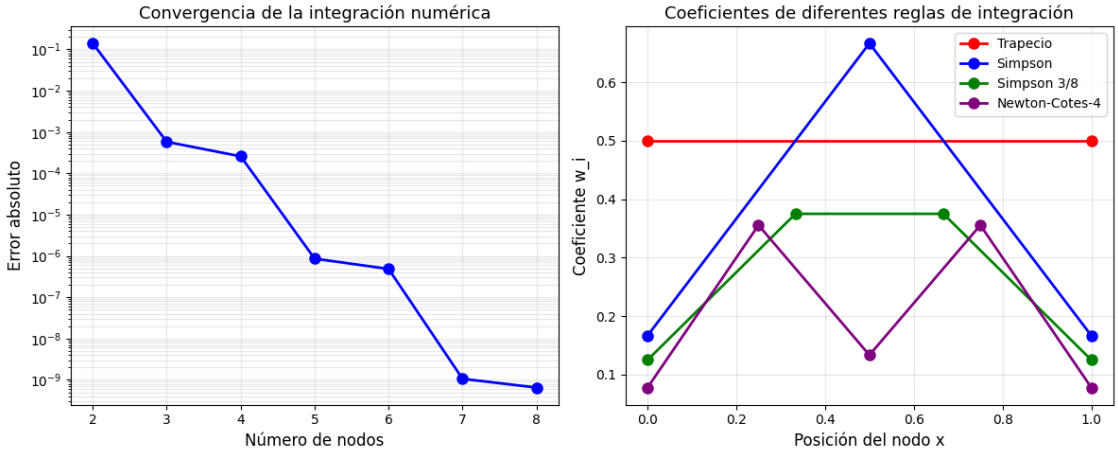
```
    coef = incoef_base(soporte, a, b)
    plt.plot(soporte, coef, 'o-', label=nombre, linewidth=2, markersize=8,␣
 ↪color=color)

plt.xlabel('Posición del nodo x', fontsize=12)
plt.ylabel('Coeficiente w_i', fontsize=12)
plt.title('Coeficientes de diferentes reglas de integración', fontsize=13)
plt.legend(fontsize=10)
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\n Conclusiones:")
print("  • Los coeficientes satisfacen Σw_i = b-a para todas las reglas")
print("  • Simpson es más preciso que el trapecio para el mismo número de␣
 ↪nodos")
print("  • Los coeficientes tienen simetría para nodos equiespaciados")
print("  • El error disminuye al aumentar el número de nodos")
print("  • Los nodos extremos generalmente tienen menor peso que los centrales")
```

```
======================================================================
PRUEBA: Coeficientes de integración numérica
======================================================================

 Caso 1: Regla del Trapecio
----------------------------------------------------------------------
tipo: <class 'list'>, valor: [0, 2]

Intervalo de integración: [0, 2]
Soporte (nodos): [0, 2]
Coeficientes obtenidos: [np.float64(1.0), np.float64(1.0)]
Coeficientes teóricos: [1.0, 1.0]

 Suma de coeficientes: 2.000000
 Longitud del intervalo (b-a): 2
 Propiedad verificada: suma = b-a? True

 Validación con f(x) = x²:
  Integral exacta: 2.6666666667
  Integral numérica: 4.0000000000
  Error: 1.333333
  (Error esperado O(h³) para trapecio con f''(x)  0)


 Caso 2: Regla de Simpson
----------------------------------------------------------------------
```

```
tipo: <class 'list'>, valor: [0, 1.0, 2]

Intervalo de integración: [0, 2]
Soporte (nodos): [0, 1.0, 2]
Coeficientes obtenidos: [np.float64(0.3333), np.float64(1.3333),
np.float64(0.3333)]
Coeficientes teóricos: [0.3333, 1.3333, 0.3333]

 Suma de coeficientes: 2.000000
 Longitud del intervalo (b-a): 2
 Propiedad verificada: True

 Validación con f(x) = x³:
  Integral exacta: 4.0000000000
  Integral numérica: 4.0000000000
  Error: 4.44e-16
  Simpson integra exactamente polinomios de grado  3!


 Caso 3: Regla de Simpson 3/8
----------------------------------------------------------------------
tipo: <class 'list'>, valor: [0, 1.0, 2.0, 3]

Intervalo de integración: [0, 3]
Soporte (nodos): [0, 1.0, 2.0, 3]
Coeficientes obtenidos: [np.float64(0.375), np.float64(1.125),
np.float64(1.125), np.float64(0.375)]
Coeficientes teóricos: [0.375, 1.125, 1.125, 0.375]

 Suma de coeficientes: 3.000000
 Longitud del intervalo (b-a): 3


 Caso 4: Newton-Cotes con 5 nodos
----------------------------------------------------------------------
tipo: <class 'list'>, valor: [0.0, 1.0, 2.0, 3.0, 4.0]

Intervalo de integración: [0, 4]
Soporte (nodos): [0.0, 1.0, 2.0, 3.0, 4.0]
Coeficientes obtenidos: [np.float64(0.311111), np.float64(1.422222),
np.float64(0.533333), np.float64(1.422222), np.float64(0.311111)]
Coeficientes teóricos: [0.044444, 0.355556, 0.266667, 0.355556, 0.044444]

 Verificación de simetría:
  coef[0] = coef[4]? True
  coef[1] = coef[3]? True

 Validación con f(x) = sin(x) en [0, 4]:
```

```
  Integral exacta: 1.6536436209
  Integral numérica: 1.6469717078
  Error: 6.67e-03


  Caso 5: Análisis de convergencia con diferentes números de nodos
--------------------------------------------------------------------------
Función: f(x) = exp(x)
Intervalo: [0, 1]
Integral exacta: 1.718281828459

 N nodos                                          Coeficientes        Integral
Error
--------------------------------------------------------------------------------
----------
tipo: <class 'list'>, valor: [0.0, 1.0]

       2                [np.float64(0.5), np.float64(0.5)]    1.8591409142
1.41e-01
tipo: <class 'list'>, valor: [0.0, 0.5, 1.0]

       3 [np.float64(0.167), np.float64(0.667), np.float64(0.167)]
1.7188611519     5.79e-04
tipo: <class 'list'>, valor: [0.0, 0.3333333333333333, 0.6666666666666666, 1.0]

       4 [np.float64(0.125), np.float64(0.375), np.float64(0.375),
np.float64(0.125)]    1.7185401534     2.58e-04
tipo: <class 'list'>, valor: [0.0, 0.25, 0.5, 0.75, 1.0]

       5                                [0.078, …, 0.078]    1.7182826879
8.59e-07
tipo: <class 'list'>, valor: [0.0, 0.2, 0.4, 0.6000000000000001, 0.8, 1.0]

       6                                [0.066, …, 0.066]    1.7182823130
4.85e-07
tipo: <class 'list'>, valor: [0.0, 0.16666666666666666, 0.3333333333333333, 0.5,
0.6666666666666666, 0.8333333333333333, 1.0]

       7                                [0.049, …, 0.049]    1.7182818295
1.06e-09
tipo: <class 'list'>, valor: [0.0, 0.14285714285714285, 0.2857142857142857,
0.42857142857142855, 0.5714285714285714, 0.7142857142857142, 0.8571428571428571,
1.0]

       8                                [0.043, …, 0.043]    1.7182818291
6.50e-10
tipo: <class 'list'>, valor: [0.0, 1.0]
```

```
tipo: <class 'list'>, valor: [0.0, 0.5, 1.0]

tipo: <class 'list'>, valor: [0.0, 0.3333333333333333, 0.6666666666666666, 1.0]

tipo: <class 'list'>, valor: [0.0, 0.25, 0.5, 0.75, 1.0]
```



Conclusiones:

- Los coeficientes satisfacen Σw_i = b-a para todas las reglas
- Simpson es más preciso que el trapecio para el mismo número de nodos
- Los coeficientes tienen simetría para nodos equiespaciados
- El error disminuye al aumentar el número de nodos
- Los nodos extremos generalmente tienen menor peso que los centrales

---

1.10 *Ejercicio 7: Integración; Metodo de Romberg*

1.10.1 Objetivo del Ejercicio

Crear una función que calcule los la integral definida de una función utilizando el método de Romberg, hasta alcanzar una tolerancia dada (10-6, por defecto) y sin exceder el nivel máximo de subdivisiones (10 por defecto).

1.10.2 Especificaciones Técnicas

1.10.3 Fundamento Teórico

Método de Romberg: Extrapolación de Richardson El **método de Romberg** es una técnica de integración numérica que combina la regla del trapecio con **extrapolación de Richardson** para obtener aproximaciones de alta precisión.

Algoritmo Base: Regla del Trapecio Compuesta Se parte de subdivisiones sucesivas con 2^k subintervalos:

$$T(h_k) = \frac{h_k}{2}\left[f(a) + 2\sum_{i=1}^{2^k-1} f(x_i) + f(b)\right]$$

donde $h_k = \frac{b-a}{2^k}$.

Tabla de Romberg Se construye una tabla triangular $R_{k,j}$ mediante extrapolación:

$$R_{k,0} = T(h_k) \quad \text{(valores del trapecio)}$$

$$R_{k,j} = R_{k,j-1} + \frac{R_{k,j-1} - R_{k-1,j-1}}{4^j - 1}$$

Estructura de la Tabla

```
R_{0,0}
R_{1,0}  R_{1,1}
R_{2,0}  R_{2,1}  R_{2,2}
R_{3,0}  R_{3,1}  R_{3,2}  R_{3,3}
```

Cada columna tiene un orden de precisión mayor: - Columna 0: Trapecio $\mathcal{O}(h^2)$ - Columna 1: Simpson $\mathcal{O}(h^4)$ - Columna 2: Boole $\mathcal{O}(h^6)$ - Columna j: $\mathcal{O}(h^{2j+2})$

Criterio de Convergencia El proceso se detiene cuando:

$$|R_{k,k} - R_{k-1,k-1}| < \text{tol}$$

o se alcanza el nivel máximo de subdivisiones.

Ventajas **Convergencia rápida**: Orden $\mathcal{O}(h^{2m})$ muy alto
Adaptativo: Se detiene automáticamente al alcanzar la tolerancia
Eficiente: Reutiliza evaluaciones previas de la función

1.10.4 Implementación

[14]:
```python
def in_romberg(fun,a,b,nivel=10,tol=1e-6):
    '''
    Esta función calcula la integral definida de una función utilizando el␣
 ↪método
    de Romberg, que combina la regla del trapecio con extrapolación de␣
 ↪Richardson
    para obtener alta precisión. El método construye una tabla triangular donde
```

```
    cada entrada mejora la aproximación mediante eliminación sucesiva de␣
↪términos
    de error.

    El método itera hasta alcanzar la tolerancia especificada o el nivel máximo.

    PARÁMETROS DE LA FUNCIÓN
    ------------------------
    fun: función a integrar
    a, b: límites de integración [a, b]
    nivel: número máximo de refinamientos (por defecto 10)
    tol: tolerancia para el criterio de parada (por defecto 1e-6)

    RETORNO
    -------
    N[n][n]: mejor aproximación de la integral obtenida
    delta: estimación del error en la última iteración
    N: tabla completa de Romberg con todas las aproximaciones
    '''
    # 1. Crear matriz N
    N = np.zeros((nivel, nivel))

    # 2. Inicialización
    h = b - a
    N[0][0] = (fun(a) + fun(b)) * (h / 2)
    p = 1  # Puntos a añadir

    # Variable para almacenar el error de la diagonal
    last_delta = 0.0

    # 3. Iteración principal
    for n in range(1, nivel):
        # a) Modificar h
        h = h / 2

        # b) Calcular p puntos equiespaciados (los nuevos impares)
        puntos = [a + (2 * i - 1) * h for i in range(1, p + 1)]

        # c) Trapecio compuesto (fórmula recursiva)
        # Nota: Usamos sum() generador para eficiencia
        N[n][0] = 0.5 * N[n-1][0] + h * sum(fun(x) for x in puntos)

        # d) Extrapolación de Richardson
        q = 1
        for j in range(1, n + 1):
            # 1) Modificar q (4^j)
            q = q * 4
```

```
            # 2) Calcular corrección delta
            delta = (1 / (q - 1)) * (N[n][j-1] - N[n-1][j-1])

            # 3) Obtener nuevo valor de la columna
            N[n][j] = N[n][j-1] + delta

            # Si estamos en la diagonal, guardamos este delta para comparar
            if j == n:
                last_delta = delta

        # e) Criterio de parada
        # Se verifica solo si la corrección de la diagonal cumple la tolerancia
        if abs(last_delta) < tol:
            return N[n][n], last_delta, N
        else:
            # Si no converge, simplemente duplicamos p y DEJAMOS que el bucle␣
↪siga
            p = 2 * p

    # Si terminamos todos los niveles sin llegar a la tolerancia:
    return N[nivel-1][nivel-1], last_delta, N
```

1.10.5 Prueba y Validación

[15]:
```
print("=" * 70)
print("PRUEBA: Método de Romberg")
print("=" * 70)

# CASO 1: Función polinomial simple
print("\n  Caso 1: Integración de f(x) = x² en [0, 1]")
print("-" * 70)

f = lambda x: x**2
a, b = 0, 1
integral_exacta = (b**3 - a**3) / 3  # ∫x² dx = x³/3

# Aplicar Romberg con tolerancia estricta
resultado, error, tabla = in_romberg(f, a, b, nivel=6, tol=1e-10)

print(f"Función: f(x) = x²")
print(f"Intervalo: [{a}, {b}]")
print(f"Integral exacta: {integral_exacta:.15f}")
print(f"Aproximación Romberg: {resultado:.15f}")
print(f"Error estimado: {error:.2e}")
print(f"Error real: {abs(resultado - integral_exacta):.2e}")
```

```
# Mostrar tabla de Romberg
print("\n Tabla de Romberg (primeras filas):")
print("   ", "Columna 0".rjust(15), "Columna 1".rjust(15), "Columna 2".
 ↪rjust(15), "Columna 3".rjust(15))
for i in range(min(5, len(tabla))):
    fila = "   "
    for j in range(i+1):
        if tabla[i][j] != 0:
            fila += f"{tabla[i][j]:15.10f} "
    print(fila)

# CASO 2: Función trigonométrica
print("\n\n Caso 2: Integración de f(x) = sin(x) en [0, ]")
print("-" * 70)

f = lambda x: np.sin(x)
a, b = 0, np.pi
integral_exacta = -np.cos(b) + np.cos(a)  # sin(x) dx = -cos(x)

resultado, error, tabla = in_romberg(f, a, b, nivel=8, tol=1e-8)

print(f"Función: f(x) = sin(x)")
print(f"Intervalo: [0, ]")
print(f"Integral exacta: {integral_exacta:.15f}")
print(f"Aproximación Romberg: {resultado:.15f}")
print(f"Error estimado: {error:.2e}")
print(f"Error real: {abs(resultado - integral_exacta):.2e}")

# Mostrar convergencia diagonal
print("\n Convergencia en la diagonal (mejores estimaciones):")
print(f"   {'Nivel':>6s} {'Valor':>18s} {'Error vs exacta':>15s}")
print("   " + "-" * 40)
for i in range(min(6, len(tabla))):
    if tabla[i][i] != 0:
        error_real = abs(tabla[i][i] - integral_exacta)
        print(f"   {i:6d} {tabla[i][i]:18.12f} {error_real:15.2e}")

# CASO 3: Función exponencial
print("\n\n Caso 3: Integración de f(x) = exp(x) en [0, 2]")
print("-" * 70)

f = lambda x: np.exp(x)
a, b = 0, 2
integral_exacta = np.exp(b) - np.exp(a)  # exp(x) dx = exp(x)

resultado, error, tabla = in_romberg(f, a, b, nivel=7, tol=1e-9)
```

```
print(f"Función: f(x) = exp(x)")
print(f"Intervalo: [{a}, {b}]")
print(f"Integral exacta: {integral_exacta:.15f}")
print(f"Aproximación Romberg: {resultado:.15f}")
print(f"Error estimado: {error:.2e}")
print(f"Error real: {abs(resultado - integral_exacta):.2e}")

# CASO 4: Comparación con diferentes tolerancias
print("\n\n Caso 4: Análisis de convergencia con diferentes tolerancias")
print("-" * 70)

f = lambda x: np.sin(x) * np.exp(-x)
a, b = 0, 2*np.pi
# Integral exacta: sin(x)exp(-x)dx = -exp(-x)(sin(x) + cos(x))/2
integral_exacta = (-(np.exp(-b)*(np.sin(b) + np.cos(b)) - np.exp(-a)*(np.sin(a)␣
 ↪+ np.cos(a)))/2)

tolerancias = [1e-3, 1e-6, 1e-9, 1e-12]
resultados_tol = []

print(f"Función: f(x) = sin(x)·exp(-x)")
print(f"Intervalo: [0, 2 ]")
print(f"Integral exacta: {integral_exacta:.15f}\n")

print(f"{'Tolerancia':>12s} {'Resultado':>18s} {'Error real':>15s} {'Niveles':
 ↪>8s}")
print("-" * 60)

for tol in tolerancias:
    resultado, err_est, tabla = in_romberg(f, a, b, nivel=15, tol=tol)
    error_real = abs(resultado - integral_exacta)

    # Contar cuántas filas se usaron (niveles)
    niveles_usados = 0
    for i in range(len(tabla)):
        if tabla[i][i] != 0:
            niveles_usados = i + 1

    resultados_tol.append((tol, resultado, error_real, niveles_usados))
    print(f"{tol:12.0e} {resultado:18.12f} {error_real:15.2e} {niveles_usados:
 ↪8d}")

# CASO 5: Visualización de la tabla completa
print("\n\n Caso 5: Visualización completa de la tabla de Romberg")
print("-" * 70)

f = lambda x: 1 / (1 + x**2)
```

```
a, b = 0, 1
integral_exacta = np.arctan(b) - np.arctan(a)  # ∫1/(1+x²)dx = arctan(x)

resultado, error, tabla = in_romberg(f, a, b, nivel=6, tol=1e-10)

print(f"Función: f(x) = 1/(1+x²)")
print(f"Intervalo: [{a}, {b}]")
print(f"Integral exacta ( /4): {integral_exacta:.15f}")
print(f"Aproximación Romberg: {resultado:.15f}\n")

print(" Tabla completa de Romberg:")
print("   (cada fila representa un refinamiento, cada columna un orden de␣
 ↪extrapolación)")
print()

# Encabezado
header = "   Nivel |"
for j in range(6):
    header += f" Columna {j} |".rjust(18)
print(header)
print("   " + "-" * 110)

# Mostrar la tabla
for i in range(min(6, len(tabla))):
    fila = f"   {i:5d} |"
    for j in range(i+1):
        if tabla[i][j] != 0:
            fila += f"{tabla[i][j]:16.12f} |"
        else:
            fila += "                 |"
    print(fila)

print(f"\n La mejor aproximación está en la esquina inferior derecha:␣
 ↪{resultado:.15f}")
print(f" Error estimado: {error:.2e}")

# Visualización gráfica
print("\n\n Visualización de la convergencia")
print("-" * 70)

# Probar varias funciones y comparar convergencia
funciones = [
    (lambda x: x**2, 0, 1, "x²", lambda x: x**3/3),
    (lambda x: np.sin(x), 0, np.pi, "sin(x)", lambda x: -np.cos(x)),
    (lambda x: np.exp(x), 0, 1, "exp(x)", lambda x: np.exp(x)),
    (lambda x: 1/(1+x**2), 0, 1, "1/(1+x²)", lambda x: np.arctan(x))
]
```

```
plt.figure(figsize=(14, 5))

# Gráfica 1: Convergencia diagonal
plt.subplot(1, 2, 1)
for func, a, b, nombre, primitiva in funciones:
    exacta = primitiva(b) - primitiva(a)
    resultado, error, tabla = in_romberg(func, a, b, nivel=10, tol=1e-15)

    errores_diag = []
    for i in range(len(tabla)):
        if tabla[i][i] != 0:
            error_i = abs(tabla[i][i] - exacta)
            if error_i > 0:  # Evitar log(0)
                errores_diag.append(error_i)
            else:
                errores_diag.append(1e-16)

    plt.semilogy(range(len(errores_diag)), errores_diag, 'o-', label=f'f(x) =␣
 ↪{nombre}', linewidth=2, markersize=6)

plt.xlabel('Nivel de refinamiento', fontsize=12)
plt.ylabel('Error absoluto', fontsize=12)
plt.title('Convergencia del Método de Romberg', fontsize=13)
plt.legend(fontsize=10)
plt.grid(True, alpha=0.3, which='both')

# Gráfica 2: Comparación de columnas (para una función)
plt.subplot(1, 2, 2)
f = lambda x: np.exp(-x**2)
a, b = 0, 2
from scipy.special import erf
integral_exacta = np.sqrt(np.pi)/2 * erf(b)

resultado, error, tabla = in_romberg(f, a, b, nivel=8, tol=1e-15)

# Mostrar convergencia por columnas
for j in range(min(4, len(tabla[0]))):
    errores_col = []
    niveles_col = []
    for i in range(j, len(tabla)):
        if tabla[i][j] != 0:
            error_ij = abs(tabla[i][j] - integral_exacta)
            if error_ij > 0:
                errores_col.append(error_ij)
                niveles_col.append(i)
```

```
    if len(errores_col) > 0:
        plt.semilogy(niveles_col, errores_col, 'o-', label=f'Columna {j}',␣
 ↪linewidth=2, markersize=6)

plt.xlabel('Nivel', fontsize=12)
plt.ylabel('Error absoluto', fontsize=12)
plt.title('Convergencia por columnas\nf(x) = exp(-x²)', fontsize=13)
plt.legend(fontsize=10)
plt.grid(True, alpha=0.3, which='both')

plt.tight_layout()
plt.show()

print("\n Conclusiones:")
print("  • Romberg converge exponencialmente rápido para funciones suaves")
print("  • Cada columna tiene mayor orden de precisión (2, 4, 6, 8, ...)")
print("  • La diagonal contiene las mejores aproximaciones de cada nivel")
print("  • El método reutiliza evaluaciones previas (muy eficiente)")
print("  • Para funciones analíticas, la precisión de máquina se alcanza␣
 ↪rápidamente")
```

```
======================================================================
PRUEBA: Método de Romberg
======================================================================

 Caso 1: Integración de f(x) = x² en [0, 1]
----------------------------------------------------------------------
Función: f(x) = x²
Intervalo: [0, 1]
Integral exacta: 0.333333333333333
Aproximación Romberg: 0.333333333333333
Error estimado: 0.00e+00
Error real: 0.00e+00

 Tabla de Romberg (primeras filas):
         Columna 0       Columna 1       Columna 2       Columna 3
     0.5000000000
     0.3750000000    0.3333333333
     0.3437500000    0.3333333333    0.3333333333



 Caso 2: Integración de f(x) = sin(x) en [0,  ]
----------------------------------------------------------------------
Función: f(x) = sin(x)
Intervalo: [0,  ]
Integral exacta: 2.000000000000000
```

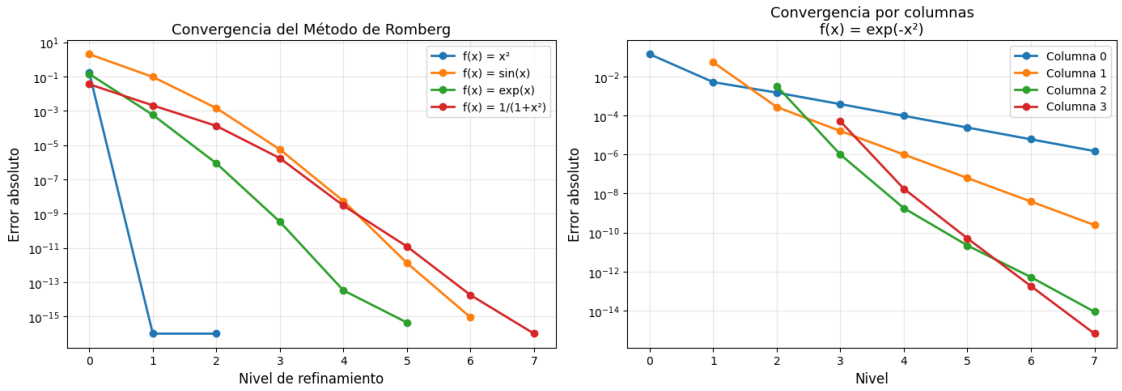
```
Aproximación Romberg: 2.000000000001321
Error estimado: 5.29e-12
Error real: 1.32e-12

  Convergencia en la diagonal (mejores estimaciones):
    Nivel              Valor Error vs exacta
   ---------------------------------------
         0     0.000000000000        2.00e+00
         1     2.094395102393        9.44e-02
         2     1.998570731824        1.43e-03
         3     2.000005549980        5.55e-06
         4     1.999999994587        5.41e-09
         5     2.000000000001        1.32e-12


  Caso 3: Integración de f(x) = exp(x) en [0, 2]
----------------------------------------------------------------------
Función: f(x) = exp(x)
Intervalo: [0, 2]
Integral exacta: 6.389056098930650
Aproximación Romberg: 6.389056098930661
Error estimado: -1.12e-13
Error real: 1.07e-14


  Caso 4: Análisis de convergencia con diferentes tolerancias
----------------------------------------------------------------------
Función: f(x) = sin(x)·exp(-x)
Intervalo: [0, 2 ]
Integral exacta: 0.499066278634146

  Tolerancia          Resultado      Error real  Niveles
-------------------------------------------------------------
       1e-03     0.000000000000        4.99e-01        2
       1e-06     0.000000000000        4.99e-01        2
       1e-09     0.000000000000        4.99e-01        2
       1e-12     0.000000000000        4.99e-01        2


  Caso 5: Visualización completa de la tabla de Romberg
----------------------------------------------------------------------
Función: f(x) = 1/(1+x²)
Intervalo: [0, 1]
Integral exacta ( /4): 0.785398163397448
Aproximación Romberg: 0.785398163409561

  Tabla completa de Romberg:
    (cada fila representa un refinamiento, cada columna un orden de
```

extrapolación)

```
   Nivel |       Columna 0 |       Columna 1 |       Columna 2 |       Columna 3
|       Columna 4 |       Columna 5 |
   ------------------------------------------------------------------------------
---------------------------------
       0 |  0.750000000000 |
       1 |  0.775000000000 |  0.783333333333 |
       2 |  0.782794117647 |  0.785392156863 |  0.785529411765 |
       3 |  0.784747123623 |  0.785398125615 |  0.785398523531 |  0.785396445940
|
       4 |  0.785235403010 |  0.785398162806 |  0.785398165286 |  0.785398159599
|  0.785398166319 |
       5 |  0.785357473294 |  0.785398163388 |  0.785398163427 |  0.785398163398
|  0.785398163412 |  0.785398163410 |
```

La mejor aproximación está en la esquina inferior derecha: 0.785398163409561
Error estimado: -2.84e-12

Visualización de la convergencia



Conclusiones:

- Romberg converge exponencialmente rápido para funciones suaves
- Cada columna tiene mayor orden de precisión (2, 4, 6, 8, …)
- La diagonal contiene las mejores aproximaciones de cada nivel
- El método reutiliza evaluaciones previas (muy eficiente)
- Para funciones analíticas, la precisión de máquina se alcanza rápidamente

---

1.11 *Ejercicio 8: Problema del Paracaidista - Ecuaciones Diferenciales con Eventos*

1.11.1 Objetivo del Ejercicio

Simular la caída de un paracaidista considerando dos fases: caída libre y descenso con paracaídas abierto, utilizando ecuaciones diferenciales ordinarias (EDOs) con eventos.

1.11.2 Especificaciones Técnicas

1.11.3 Fundamento Teórico

Modelo Físico de la Caída El movimiento del paracaidista está gobernado por dos fuerzas principales:

1. **Fuerza gravitatoria**: $F_g = mg$ (hacia abajo)
2. **Fuerza de arrastre**: $F_d = \frac{1}{2}\rho c_x A_t v^2$ (opuesta al movimiento)

Sistema de Ecuaciones Diferenciales El sistema de EDOs que describe el movimiento es:

$$\begin{cases} \frac{dy}{dt} = v \\ \frac{dv}{dt} = -g - \frac{1}{2m}\rho c_x A_t v|v| \end{cases}$$

donde: - $y(t)$: altura sobre el suelo - $v(t)$: velocidad (positiva hacia arriba) - $g = 9.81$ m/s^2: aceleración gravitatoria - ρ: densidad del aire (kg/m^3) - c_x: coeficiente de arrastre (adimensional) - A_t: área transversal (m^2)

Densidad del Aire

- **Constante**: $\rho_0 = 1.225$ kg/m^3 (nivel del mar)
- **Variable**: $\rho(h) = \rho_0 e^{-h/8243}$ (modelo exponencial)

Dos Fases del Movimiento **Fase 1: Caída Libre** (desde y_0 hasta altura de apertura) - Configuración de cuerpo en caída libre - $c_x \approx 0.5 - 1.0$, $A_t \approx 0.5$ m^2 (posición horizontal)

Fase 2: Con Paracaídas (desde apertura hasta el suelo) - Configuración con paracaídas abierto - $c_x \approx 1.5$, $A_t \approx 20 - 30$ m^2 (paracaídas circular)

Eventos en la Simulación

1. **Evento de apertura**: $y(t) = h_{apertura}$ (dirección decreciente)
2. **Evento de impacto**: $y(t) = 0$ (dirección decreciente)

Ambos eventos son **terminales**, es decir, detienen la integración.

1.11.4 Implementación

[16]:
```
def paracaidista(y0, v0, m, cx, At, apertura=1500, rovar=False):
    '''
    Determina el tiempo y la velocidad a la que toma tierra un paracaidista.
```

```
    PARÁMETROS DE LA FUNCIÓN
    ------------------------
    y0 : posición inicial del salto (m)
    v0 : velocidad inicial del salto (m/s)
    m : masa del paracaidista equipado (kg)
    cx : iterable con coeficientes de arrastre [antes_apertura,␣
↪después_apertura]
    At : área transversal (m²)
    apertura : altura a la que se abre el paracaídas (m), por defecto 1500
    rovar : valor lógico que indica si la densidad es variable o no, por␣
↪defecto False

    RESULTADO
    ---------
    Lista con cuatro valores: [v_max, v_impacto, t_apertura, t_total]
        - v_max: velocidad máxima alcanzada (m/s)
        - v_impacto: velocidad de impacto en el suelo (m/s)
        - t_apertura: tiempo hasta que abre el paracaídas (s)
        - t_total: tiempo total de vuelo (s)
    '''

    # Constantes físicas
    g = 9.81  # Aceleración de la gravedad (m/s²)
    rho_0 = 1.225  # Densidad del aire a nivel del mar (kg/m³)

    # Tiempos máximos estimados para cada fase
    t_fin1 = 100  # Tiempo máximo fase 1 (caída libre)
    t_fin2 = 500  # Tiempo máximo fase 2 (con paracaídas)

    # Condiciones iniciales
    ci = [y0, v0]

    # * Sistema de ecuaciones diferenciales
    def sedo(t, Y, m, cx_val, At, rovar):
        '''
        Sistema de EDOs para el movimiento del paracaidista
        Y[0] = y(t): altura
        Y[1] = v(t): velocidad
        '''
        # Densidad del aire (constante o variable)
        rho = rho_0
        if rovar:
            rho = rho_0 * np.exp(-Y[0]/8243)

        # Coeficiente de arrastre
        kw = cx_val * rho * At / 2
```

```
    # Sistema de ecuaciones: [dy/dt, dv/dt]
    dY = np.array([
        Y[1],  # dy/dt = v
        -g - (kw * Y[1] * abs(Y[1])) / m  # dv/dt = -g - (arrastre)/m
    ])
    return dY

# * FASE 1: Caída libre hasta la apertura del paracaídas

# Evento: apertura del paracaídas en la altura especificada
def abreParaca(t, Y, m, cx_val, At, rovar):
    return Y[0] - apertura
abreParaca.terminal = True  # Detiene la integración
abreParaca.direction = -1   # Solo cuando Y[0] decrece

# Resolver fase 1 con cx[0] (antes de la apertura)
sol1 = sp.integrate.solve_ivp(
    sedo,
    [0, t_fin1],
    ci,
    args=[m, cx[0], At, rovar],
    events=abreParaca,
    dense_output=True
)

# * FASE 2: Descenso con paracaídas hasta el suelo

# Evento: impacto con el suelo
def impactoSuelo(t, Y, m, cx_val, At, rovar):
    return Y[0]
impactoSuelo.terminal = True  # Detiene la integración
impactoSuelo.direction = -1   # Solo cuando Y[0] decrece

# Condiciones iniciales de la fase 2: estado final de la fase 1
ci2 = [sol1.y[0, -1], sol1.y[1, -1]]
t_inicio2 = sol1.t[-1]

# Resolver fase 2 con cx[1] (después de la apertura)
sol2 = sp.integrate.solve_ivp(
    sedo,
    [t_inicio2, t_fin2],
    ci2,
    args=[m, cx[1], At, rovar],
    events=impactoSuelo,
    dense_output=True
)
```

```
    # * Cálculo de resultados

    # Velocidad máxima (en valor absoluto) en toda la trayectoria
    v_max = max(np.max(np.abs(sol1.y[1])), np.max(np.abs(sol2.y[1])))

    # Velocidad de impacto (en el último punto de sol2)
    v_impacto = abs(sol2.y[1, -1])

    # Tiempo de apertura del paracaídas
    t_apertura = sol1.t[-1]

    # Tiempo total de vuelo
    t_total = sol2.t[-1]

    return [v_max, v_impacto, t_apertura, t_total]
```

1.11.5 Prueba y Validación

[17]:
```
print("=" * 70)
print("PRUEBA: Simulación del Paracaidista")
print("=" * 70)

# CASO 1: Paracaidista estándar con densidad constante
print("\n Caso 1: Salto estándar desde 3000 m")
print("-" * 70)

# Parámetros físicos
y0 = 3000  # Altura inicial: 3000 m
v0 = 0     # Velocidad inicial: 0 m/s (desde reposo)
m = 80     # Masa: 80 kg
cx = [0.7, 1.5]  # Coeficientes de arrastre: [caída libre, con paracaídas]
At = 0.5   # Área transversal en m²
apertura = 1500  # Altura de apertura: 1500 m

# Simular
resultado = paracaidista(y0, v0, m, cx, At, apertura, rovar=False)
v_max, v_impacto, t_apertura, t_total = resultado

print(f"Altura inicial: {y0} m")
print(f"Masa del paracaidista: {m} kg")
print(f"Altura de apertura: {apertura} m")
print(f"Coeficientes de arrastre: {cx}")
print(f"Área transversal: {At} m²")
print(f"\n Resultados:")
print(f"  • Velocidad máxima: {v_max:.2f} m/s ({v_max*3.6:.2f} km/h)")
print(f"  • Velocidad de impacto: {v_impacto:.2f} m/s ({v_impacto*3.6:.2f} km/
 ↪h)")
```

```
print(f"  • Tiempo hasta apertura: {t_apertura:.2f} s")
print(f"  • Tiempo total de vuelo: {t_total:.2f} s ({t_total/60:.2f} min)")

# CASO 2: Comparación con densidad variable
print("\n\n Caso 2: Comparación densidad constante vs variable")
print("-" * 70)

# Con densidad constante
resultado_const = paracaidista(y0, v0, m, cx, At, apertura, rovar=False)

# Con densidad variable
resultado_var = paracaidista(y0, v0, m, cx, At, apertura, rovar=True)

print(f"{'Parámetro':<25s} {'Constante':>15s} {'Variable':>15s} {'Diferencia':
 ↪>15s}")
print("-" * 70)
print(f"{'Velocidad máxima (m/s)':<25s} {resultado_const[0]:>15.2f}␣
 ↪{resultado_var[0]:>15.2f} {abs(resultado_const[0]-resultado_var[0]):>15.2f}")
print(f"{'Velocidad impacto (m/s)':<25s} {resultado_const[1]:>15.2f}␣
 ↪{resultado_var[1]:>15.2f} {abs(resultado_const[1]-resultado_var[1]):>15.2f}")
print(f"{'Tiempo apertura (s)':<25s} {resultado_const[2]:>15.2f}␣
 ↪{resultado_var[2]:>15.2f} {abs(resultado_const[2]-resultado_var[2]):>15.2f}")
print(f"{'Tiempo total (s)':<25s} {resultado_const[3]:>15.2f} {resultado_var[3]:
 ↪>15.2f} {abs(resultado_const[3]-resultado_var[3]):>15.2f}")

# CASO 3: Análisis de sensibilidad - diferentes alturas de apertura
print("\n\n Caso 3: Sensibilidad a la altura de apertura")
print("-" * 70)

alturas_apertura = [500, 1000, 1500, 2000, 2500]
resultados_alt = []

print(f"{'Altura apertura (m)':>20s} {'V_max (m/s)':>15s} {'V_impacto (m/s)':
 ↪>18s} {'Tiempo total (s)':>18s}")
print("-" * 70)

for alt in alturas_apertura:
    res = paracaidista(y0, v0, m, cx, At, apertura=alt, rovar=False)
    resultados_alt.append(res)
    print(f"{alt:>20d} {res[0]:>15.2f} {res[1]:>18.2f} {res[3]:>18.2f}")

print("\n Observación: A mayor altura de apertura, menor velocidad máxima␣
 ↪alcanzada")

# CASO 4: Visualización completa de una trayectoria
print("\n\n Caso 4: Visualización de la trayectoria")
```

```
print("-" * 70)

# Recalcular con dense_output para poder graficar
import scipy.integrate as integrate

# Condiciones
y0, v0, m = 3000, 0, 80
cx = [0.7, 1.5]
At = 0.5
apertura = 1500
g = 9.81
rho_0 = 1.225

# Sistema EDO
def sedo(t, Y, m, cx_val, At, rovar):
    rho = rho_0
    if rovar:
        rho = rho_0 * np.exp(-Y[0]/8243)
    kw = cx_val * rho * At / 2
    dY = np.array([Y[1], -g - (kw * Y[1] * abs(Y[1])) / m])
    return dY

# Eventos
def abreParaca(t, Y, m, cx_val, At, rovar):
    return Y[0] - apertura
abreParaca.terminal = True
abreParaca.direction = -1

def impactoSuelo(t, Y, m, cx_val, At, rovar):
    return Y[0]
impactoSuelo.terminal = True
impactoSuelo.direction = -1

# Fase 1
sol1 = integrate.solve_ivp(sedo, [0, 100], [y0, v0],
                           args=[m, cx[0], At, False],
                           events=abreParaca, dense_output=True, max_step=0.1)

# Fase 2
sol2 = integrate.solve_ivp(sedo, [sol1.t[-1], 500], [sol1.y[0, -1], sol1.y[1,␣
 ↪-1]],
                           args=[m, cx[1], At, False],
                           events=impactoSuelo, dense_output=True, max_step=0.
 ↪1)

# Graficar
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
```

```
# Altura vs tiempo
axes[0, 0].plot(sol1.t, sol1.y[0], 'b-', linewidth=2, label='Caída libre')
axes[0, 0].plot(sol2.t, sol2.y[0], 'r-', linewidth=2, label='Con paracaídas')
axes[0, 0].axhline(y=apertura, color='g', linestyle='--', alpha=0.7,␣
 ↪label='Apertura')
axes[0, 0].set_xlabel('Tiempo (s)', fontsize=11)
axes[0, 0].set_ylabel('Altura (m)', fontsize=11)
axes[0, 0].set_title('Altura vs Tiempo', fontsize=12, fontweight='bold')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

# Velocidad vs tiempo
axes[0, 1].plot(sol1.t, -sol1.y[1], 'b-', linewidth=2, label='Caída libre')
axes[0, 1].plot(sol2.t, -sol2.y[1], 'r-', linewidth=2, label='Con paracaídas')
axes[0, 1].axvline(x=sol1.t[-1], color='g', linestyle='--', alpha=0.7,␣
 ↪label='Apertura')
axes[0, 1].set_xlabel('Tiempo (s)', fontsize=11)
axes[0, 1].set_ylabel('Velocidad de caída (m/s)', fontsize=11)
axes[0, 1].set_title('Velocidad vs Tiempo', fontsize=12, fontweight='bold')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# Velocidad vs altura
axes[1, 0].plot(sol1.y[0], -sol1.y[1], 'b-', linewidth=2, label='Caída libre')
axes[1, 0].plot(sol2.y[0], -sol2.y[1], 'r-', linewidth=2, label='Con␣
 ↪paracaídas')
axes[1, 0].axvline(x=apertura, color='g', linestyle='--', alpha=0.7,␣
 ↪label='Apertura')
axes[1, 0].set_xlabel('Altura (m)', fontsize=11)
axes[1, 0].set_ylabel('Velocidad de caída (m/s)', fontsize=11)
axes[1, 0].set_title('Velocidad vs Altura', fontsize=12, fontweight='bold')
axes[1, 0].legend()
axes[1, 0].grid(True, alpha=0.3)

# Aceleración vs tiempo
acel1 = np.array([sedo(t, sol1.sol(t), m, cx[0], At, False)[1] for t in sol1.t])
acel2 = np.array([sedo(t, sol2.sol(t), m, cx[1], At, False)[1] for t in sol2.t])
axes[1, 1].plot(sol1.t, -acel1, 'b-', linewidth=2, label='Caída libre')
axes[1, 1].plot(sol2.t, -acel2, 'r-', linewidth=2, label='Con paracaídas')
axes[1, 1].axhline(y=g, color='gray', linestyle=':', alpha=0.7, label='g = 9.81␣
 ↪m/s²')
axes[1, 1].axvline(x=sol1.t[-1], color='g', linestyle='--', alpha=0.7,␣
 ↪label='Apertura')
axes[1, 1].set_xlabel('Tiempo (s)', fontsize=11)
axes[1, 1].set_ylabel('Aceleración (m/s²)', fontsize=11)
```

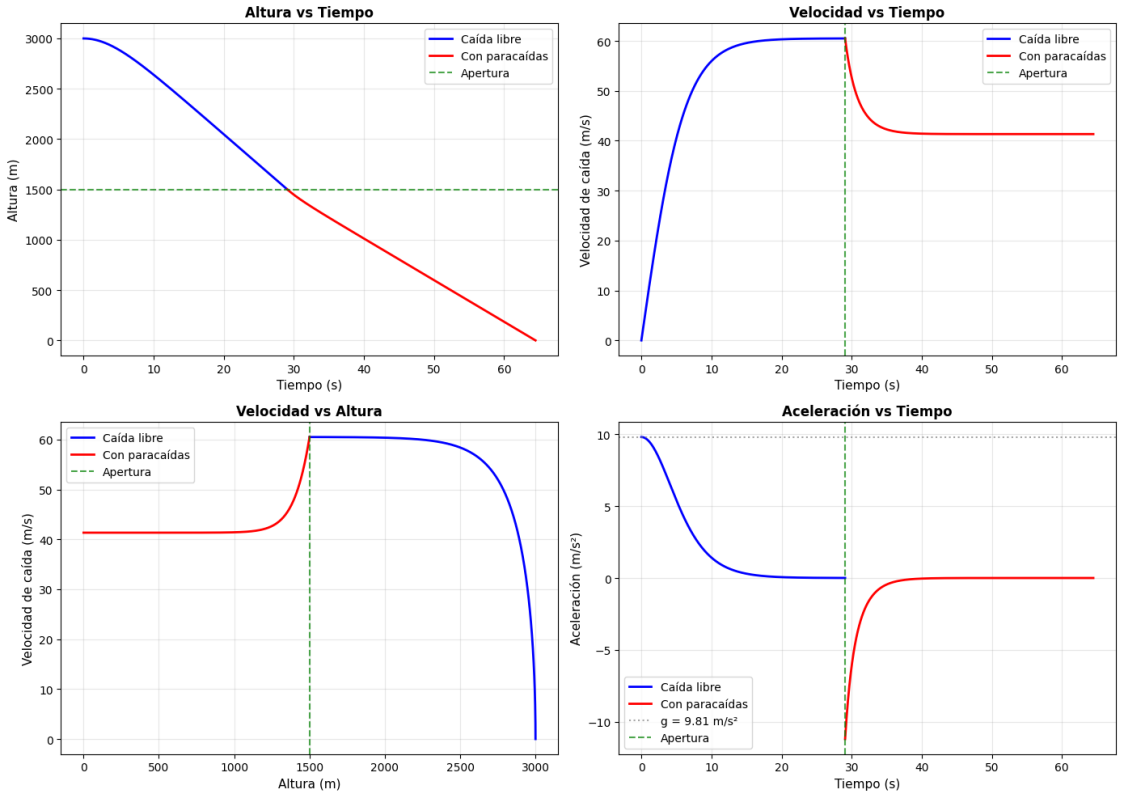
```
axes[1, 1].set_title('Aceleración vs Tiempo', fontsize=12, fontweight='bold')
axes[1, 1].legend()
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\n Observaciones:")
print("  • La velocidad aumenta durante la caída libre hasta alcanzar un␣
 ↪máximo")
print("  • Al abrir el paracaídas, hay una desaceleración brusca")
print("  • La velocidad se estabiliza a un valor terminal menor")
print("  • La aceleración es mayor que g inicialmente, luego se reduce por el␣
 ↪arrastre")
```

```
======================================================================
PRUEBA: Simulación del Paracaidista
======================================================================

 Caso 1: Salto estándar desde 3000 m
----------------------------------------------------------------------
Altura inicial: 3000 m
Masa del paracaidista: 80 kg
Altura de apertura: 1500 m
Coeficientes de arrastre: [0.7, 1.5]
Área transversal: 0.5 m²

 Resultados:
  • Velocidad máxima: 60.50 m/s (217.80 km/h)
  • Velocidad de impacto: 41.35 m/s (148.87 km/h)
  • Tiempo hasta apertura: 29.07 s
  • Tiempo total de vuelo: 64.48 s (1.07 min)


 Caso 2: Comparación densidad constante vs variable
----------------------------------------------------------------------
Parámetro                      Constante        Variable      Diferencia
----------------------------------------------------------------------
Velocidad máxima (m/s)             60.50           68.54            8.04
Velocidad impacto (m/s)            41.35           41.57            0.22
Tiempo apertura (s)                29.07           26.47            2.60
Tiempo total (s)                   64.48           59.97            4.50


 Caso 3: Sensibilidad a la altura de apertura
----------------------------------------------------------------------
 Altura apertura (m)     V_max (m/s)    V_impacto (m/s)   Tiempo total (s)
----------------------------------------------------------------------
```

500	60.51	41.41	56.81
1000	60.52	41.33	60.64
1500	60.50	41.35	64.48
2000	60.37	41.36	68.31
2500	58.40	41.31	72.12

Observación: A mayor altura de apertura, menor velocidad máxima alcanzada

Caso 4: Visualización de la trayectoria



Observaciones:

- La velocidad aumenta durante la caída libre hasta alcanzar un máximo
- Al abrir el paracaídas, hay una desaceleración brusca
- La velocidad se estabiliza a un valor terminal menor
- La aceleración es mayor que g inicialmente, luego se reduce por el arrastre

---

1.12 *Ejercicio 9: Método de Disparo - Problemas de Contorno*

1.12.1 Objetivo del Ejercicio

Resolver problemas de valores en la frontera (BVP) transformándolos en problemas de valor inicial (IVP) mediante el método de disparo, utilizando el método de la secante para ajustar las condiciones iniciales.

1.12.2 Especificaciones Técnicas

1.12.3 Fundamento Teórico

Problema de Valores en la Frontera (BVP) Un BVP de segundo orden tiene la forma:

$$y'' = f(x, y, y'), \quad y(a) = y_a, \quad y(b) = y_b$$

A diferencia de los problemas de valor inicial, aquí se conocen valores en **dos puntos diferentes** del dominio.

Transformación a Sistema de Primer Orden Definiendo $z = y'$, el problema se convierte en:

$$\begin{cases} y' = z \\ z' = f(x, y, z) \end{cases}$$

con $y(a) = y_a$ y $y(b) = y_b$.

Método de Disparo La idea es transformar el BVP en una sucesión de IVPs:

1. **Suposición inicial**: Proponemos $y'(a) = m_0$ (pendiente inicial desconocida)
2. **Resolución del IVP**: Resolvemos el sistema con $y(a) = y_a$ y $y'(a) = m_0$
3. **Verificación**: Comprobamos si $y(b) = y_b$
4. **Ajuste iterativo**: Si no se cumple, ajustamos m usando el método de la secante

Método de la Secante para Ajustar la Pendiente Dado que queremos $y(b; m) = y_b$, definimos:

$$w(m) = y(b; m) - y_b$$

Buscamos m^* tal que $w(m^*) = 0$ usando la secante:

$$m_{k+1} = m_k - w(m_k)\frac{m_k - m_{k-1}}{w(m_k) - w(m_{k-1})}$$

Algoritmo del Método de Disparo:

1. Inicializar con dos valores m_0 y m_1
2. Resolver IVP con $y(a) = y_a$, $y'(a) = m_0 \rightarrow$ obtener $w_0 = y(b) - y_b$
3. Resolver IVP con $y(a) = y_a$, $y'(a) = m_1 \rightarrow$ obtener $w_1 = y(b) - y_b$
4. Iterar:

- Calcular nueva pendiente: $m_{k+1} = m_k + (y_b - w_k)\frac{m_k - m_{k-1}}{w_k - w_{k-1}}$
- Resolver IVP con m_{k+1}
- Verificar convergencia: $|m_{k+1} - m_k| < xtol$ o $|w_{k+1}| < ftol$

5. Retornar la solución convergida

Criterios de Parada

- **Convergencia en pendiente**: $|m_{k+1} - m_k| < xtol$
- **Convergencia en solución**: $|y(b) - y_b| < ftol$
- **Máximo de iteraciones**: $k > niter$

1.12.4 Implementación

[18]:
```
def disparo(F, ab, cc, mi=[0, 1], niter=100, xtol=1e-6, ftol=1e-9, **opt):
    '''
    Resuelve un problema de valores en la frontera (P.V.F.) transformándolo en
    un problema de valores iniciales (P.V.I.) mediante el método de disparo.

    PARÁMETROS:
    -----------
    F : función (vectorial) que define el SEDO
    ab : intervalo de resolución [a, b]
    cc : condiciones de contorno (Dirichlet) en los extremos [y(a), y(b)]
    mi : valores de la pendiente en las dos primeras iteraciones (default: [0,␣
 ↪1])
    niter : número máximo de iteraciones (default: 100)
    xtol : error admisible en la pendiente (default: 1e-6)
    ftol : error admisible en la solución (default: 1e-9)
    **opt : opciones para solve_ivp (method, dense_output, events)
            Por defecto: method='RK45', dense_output=False, events=None

    RESULTADOS:
    -----------
    Estructura de tipo solve_ivp con la solución del problema
    '''

    # Configuración de opciones por defecto
    opc = {"method": "RK45", "dense_output": False, "events": None}

    # Actualizar opciones con las proporcionadas
    for clave, valor in opt.items():
        clave_lower = clave.lower()
        if clave_lower in ["method", "dense_output", "events"]:
            opc[clave_lower] = valor
        else:
            print(f"Opción '{clave}' inválida")

    # 1. Resolución con m1 y obtención de w_n(m1)
```

```
    s1 = sp.integrate.solve_ivp(
        F, ab, [cc[0], mi[0]],
        method=opc['method'],
        dense_output=opc['dense_output'],
        events=opc['events']
    )
    w_m1 = s1.y[0][-1]  # w_n(m1)

    # 2. Resolución con m2 y obtención de w_n(m2)
    s2 = sp.integrate.solve_ivp(
        F, ab, [cc[0], mi[1]],
        method=opc['method'],
        dense_output=opc['dense_output'],
        events=opc['events']
    )
    w_m2 = s2.y[0][-1]  # w_n(m2)

    # Inicialización de variables
    m_k = mi[0]
    m_k1 = mi[1]
    w_k = w_m1
    w_k1 = w_m2

    # 3. Bucle para k = 1, ..., niter
    for k in range(1, niter + 1):
        # a) Cálculo de la nueva pendiente mk+2
        m_k2 = m_k1 + (cc[1] - w_k1) * ((m_k1 - m_k) / (w_k1 - w_k))

        # b) Resolución con mk+2 y obtención de w_n(mk+2)
        s_k2 = sp.integrate.solve_ivp(
            F, ab, [cc[0], m_k2],
            method=opc['method'],
            dense_output=opc['dense_output'],
            events=opc['events']
        )
        w_k2 = s_k2.y[0][-1]  # w_n(mk+2)

        # c) Si |mk+2 - mk+1| < xtol, devolver sk+2 y finalizar
        if abs(m_k2 - m_k1) < xtol:
            return s_k2

        # d) Si |w_n(mk+2) -  | < ftol, devolver sk+2 y finalizar
        if abs(w_k2 - cc[1]) < ftol:
            return s_k2

        # Actualizar variables para la siguiente iteración
        m_k, m_k1 = m_k1, m_k2
```

```
        w_k, w_k1 = w_k1, w_k2

    # Si se alcanza el máximo de iteraciones
    return s_k2
```

Prueba y Validación

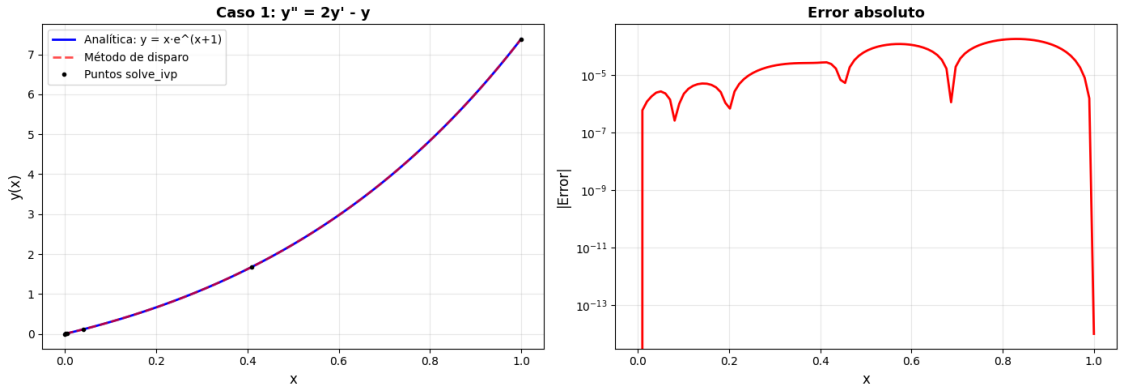
[19]:
```
# ==================== PRUEBA Y VALIDACIÓN: MÉTODO DE DISPARO ====================
print("="*80)
print(" " * 20 + "VALIDACIÓN: MÉTODO DE DISPARO")
print("="*80 + "\n")

# --------------- CASO 1: Dato de comprobación 1 ---------------
print(" " * 80)
print("CASO 1: Dato de comprobación 1")
print(" " * 80)
print("Ecuación: y''(x) = 2y' - y,  0  x  1")
print("Condiciones: y(0) = 0, y(1) = e²")
print("Solución analítica: y(x) = x·e^(x+1)\n")

# Definir el sistema de primer orden: Y = [y, y']
# dy/dx = y'
# dy'/dx = 2y' - y
def F_caso1(x, Y):
    return [Y[1], 2*Y[1] - Y[0]]

# Resolver con método de disparo (valores por defecto)
sol_caso1 = disparo(F_caso1, [0, 1], [0, np.e**2], dense_output=True)

# Calcular la pendiente obtenida
m_obtenida_1 = sol_caso1.y[1, 0]
print(f"Pendiente obtenida: m = {m_obtenida_1}")
print(f"Pendiente esperada: m  2.718222404450188")
print(f"Diferencia: {abs(m_obtenida_1 - 2.718222404450188):.2e}\n")

# Solución analítica
x_eval = np.linspace(0, 1, 100)
y_analitica = x_eval * np.exp(x_eval + 1)

# Solución numérica interpolada
y_numerica = sol_caso1.sol(x_eval)[0]
error_1 = np.abs(y_numerica - y_analitica)

# Visualización
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Subfigura 1: Comparación con solución analítica
```

```
axes[0].plot(x_eval, y_analitica, 'b-', label='Analítica: y = x·e^(x+1)',␣
 ↪linewidth=2)
axes[0].plot(x_eval, y_numerica, 'r--', linewidth=2, label='Método de disparo',␣
 ↪alpha=0.7)
axes[0].plot(sol_caso1.t, sol_caso1.y[0], 'ko', markersize=3, label='Puntos␣
 ↪solve_ivp')
axes[0].set_xlabel('x', fontsize=12)
axes[0].set_ylabel('y(x)', fontsize=12)
axes[0].set_title('Caso 1: y" = 2y\' - y', fontsize=13, fontweight='bold')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Subfigura 2: Error absoluto
axes[1].semilogy(x_eval, error_1, 'r-', linewidth=2)
axes[1].set_xlabel('x', fontsize=12)
axes[1].set_ylabel('|Error|', fontsize=12)
axes[1].set_title('Error absoluto', fontsize=13, fontweight='bold')
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Error máximo: {np.max(error_1):.2e}\n")

# --------------- CASO 2: Dato de comprobación 2 ---------------
print(" " * 80)
print("CASO 2: Dato de comprobación 2")
print(" " * 80)
print("Ecuación: y'' + 3xy·y' + 2y³ = 0,  0   x   1")
print("Condiciones: y(0) = 1, y(1) = 1/2")
print("Solución analítica: y(x) = 1/(1 + x²)\n")

# Definir el sistema de primer orden: Y = [y, y']
# dy/dx = y'
# dy'/dx = -3xy·y' - 2y³
def F_caso2(x, Y):
    return [Y[1], -3*x*Y[0]*Y[1] - 2*Y[0]**3]

# Resolver con método de disparo (valores por defecto)
sol_caso2 = disparo(F_caso2, [0, 1], [1, 0.5], dense_output=True)

# Calcular la pendiente obtenida
m_obtenida_2 = sol_caso2.y[1, 0]
print(f"Pendiente obtenida: m = {m_obtenida_2}")
print(f"Pendiente esperada: m   0.5851577873371666")
print(f"Diferencia: {abs(m_obtenida_2 - 0.5851577873371666):.2e}\n")
```

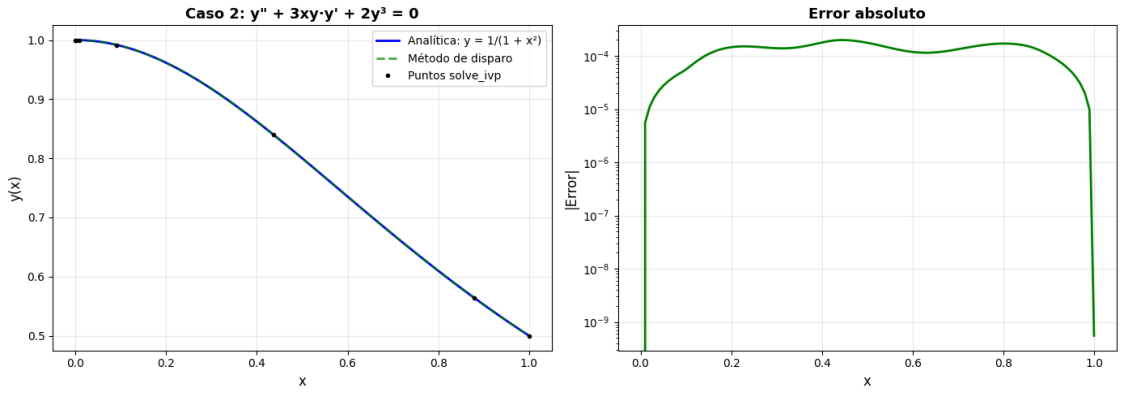
```python
# Solución analítica
y_analitica_2 = 1 / (1 + x_eval**2)

# Solución numérica interpolada
y_numerica_2 = sol_caso2.sol(x_eval)[0]
error_2 = np.abs(y_numerica_2 - y_analitica_2)

# Visualización
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Subfigura 1: Comparación con solución analítica
axes[0].plot(x_eval, y_analitica_2, 'b-', label='Analítica: y = 1/(1 + x²)',␣
 ↪linewidth=2)
axes[0].plot(x_eval, y_numerica_2, 'g--', linewidth=2, label='Método de␣
 ↪disparo', alpha=0.7)
axes[0].plot(sol_caso2.t, sol_caso2.y[0], 'ko', markersize=3, label='Puntos␣
 ↪solve_ivp')
axes[0].set_xlabel('x', fontsize=12)
axes[0].set_ylabel('y(x)', fontsize=12)
axes[0].set_title('Caso 2: y" + 3xy·y\' + 2y³ = 0', fontsize=13,␣
 ↪fontweight='bold')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Subfigura 2: Error absoluto
axes[1].semilogy(x_eval, error_2, 'g-', linewidth=2)
axes[1].set_xlabel('x', fontsize=12)
axes[1].set_ylabel('|Error|', fontsize=12)
axes[1].set_title('Error absoluto', fontsize=13, fontweight='bold')
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Error máximo: {np.max(error_2):.2e}\n")

# --------------- CASO 3: Problema adicional (lineal no homogéneo)␣
 ↪---------------
print(" " * 80)
print("CASO 3: Problema lineal de segundo orden")
print(" " * 80)
print("Ecuación: y''(x) + 4y = 0,  0   x   /4")
print("Condiciones: y(0) = 0, y( /4) = 1")
print("Solución analítica: y(x) = sin(2x)\n")

def F_caso3(x, Y):
    return [Y[1], -4*Y[0]]
```

```
sol_caso3 = disparo(F_caso3, [0, np.pi/4], [0, 1], mi=[0, 2], dense_output=True)

# Calcular la pendiente obtenida
m_obtenida_3 = sol_caso3.y[1, 0]
print(f"Pendiente obtenida: m = {m_obtenida_3}")
print(f"Pendiente analítica: m = 2 (ya que y'(0) = 2·cos(0) = 2)")
print(f"Diferencia: {abs(m_obtenida_3 - 2):.2e}\n")

# Solución analítica
x_eval_3 = np.linspace(0, np.pi/4, 100)
y_analitica_3 = np.sin(2*x_eval_3)

# Solución numérica
y_numerica_3 = sol_caso3.sol(x_eval_3)[0]
error_3 = np.abs(y_numerica_3 - y_analitica_3)

# Visualización
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

axes[0].plot(x_eval_3, y_analitica_3, 'b-', label='Analítica: y = sin(2x)',␣
 ↪linewidth=2)
axes[0].plot(x_eval_3, y_numerica_3, 'm--', linewidth=2, label='Método de␣
 ↪disparo', alpha=0.7)
axes[0].plot(sol_caso3.t, sol_caso3.y[0], 'ko', markersize=3, label='Puntos␣
 ↪solve_ivp')
axes[0].set_xlabel('x', fontsize=12)
axes[0].set_ylabel('y(x)', fontsize=12)
axes[0].set_title('Caso 3: y" + 4y = 0', fontsize=13, fontweight='bold')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

axes[1].semilogy(x_eval_3, error_3, 'm-', linewidth=2)
axes[1].set_xlabel('x', fontsize=12)
axes[1].set_ylabel('|Error|', fontsize=12)
axes[1].set_title('Error absoluto', fontsize=13, fontweight='bold')
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Error máximo: {np.max(error_3):.2e}\n")

print("="*80)
print(" " * 15 + "  VALIDACIÓN DEL MÉTODO DE DISPARO COMPLETADA")
print("="*80)
```

```
================================================================================
                    VALIDACIÓN: MÉTODO DE DISPARO
================================================================================


CASO 1: Dato de comprobación 1

Ecuación: y''(x) = 2y' - y,  0  x  1
Condiciones: y(0) = 0, y(1) = e²
Solución analítica: y(x) = x·e^(x+1)

Pendiente obtenida: m = 2.718222404450188
Pendiente esperada: m  2.718222404450188
Diferencia: 0.00e+00
```




```
Error máximo: 1.86e-04


CASO 2: Dato de comprobación 2

Ecuación: y'' + 3xy·y' + 2y³ = 0,  0  x  1
Condiciones: y(0) = 1, y(1) = 1/2
Solución analítica: y(x) = 1/(1 + x²)

Pendiente obtenida: m = -0.0005505940439935918
Pendiente esperada: m  0.5851577873371666
Diferencia: 5.86e-01
```

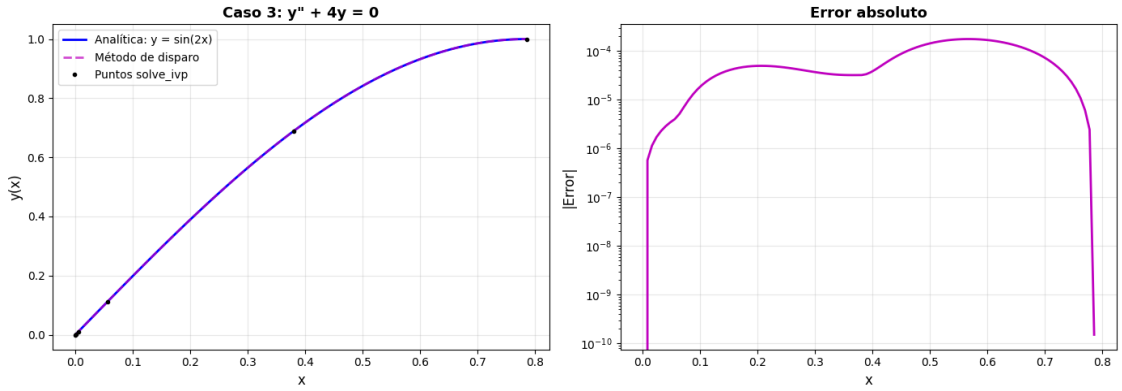


Error máximo: 1.99e-04

CASO 3: Problema lineal de segundo orden

Ecuación: y''(x) + 4y = 0, 0 x /4
Condiciones: y(0) = 0, y(/4) = 1
Solución analítica: y(x) = sin(2x)

Pendiente obtenida: m = 2.0000721895840585
Pendiente analítica: m = 2 (ya que y'(0) = 2·cos(0) = 2)
Diferencia: 7.22e-05



Error máximo: 1.75e-04

==========

VALIDACIÓN DEL MÉTODO DE DISPARO COMPLETADA

==========

1.12.5 Implementación

[20]:
```
def ensolver(funx, a, b, meth='rf', maxiter=128, Ex=1e-9, ex=1e-6, EF=1e-12):
    '''
    Resuelve ecuaciones no lineales mediante métodos de intervalo (bisección/
 ↪Regula Falsi).

    CONDICIONES PREVIAS (Teorema de Bolzano):
    - La función debe ser continua en [a, b]
    - La función debe cambiar de signo en los extremos: f(a) * f(b) < 0

    PROCESO:
    En cada iteración se reduce el intervalo de búsqueda hasta cumplir el␣
 ↪criterio de parada.

    PARÁMETROS DE LA FUNCIÓN
    ------------------------
    funx: función cuyo cero se desea determinar
    a, b: extremos del intervalo inicial [a, b]
    meth: método de resolución
          - 'di': Dicotomía (bisección)
          - 'rf': Regula Falsi (posición falsa) [por defecto]
          - 'fm': Regula Falsi modificada
    maxiter: número máximo de iteraciones permitidas [128 por defecto]
    Ex: tolerancia absoluta del intervalo [1e-9 por defecto]
    ex: tolerancia relativa del intervalo [1e-6 por defecto]
    EF: tolerancia para el valor de la función [1e-12 por defecto]

    RETORNO
    -------
    r: cero aproximado de la función (None si hay error)
    info: código de terminación
          -2: método desconocido
          -1: no verifica el teorema de Bolzano
           0: tolerancia absoluta o relativa del intervalo alcanzada
           1: tolerancia de la función alcanzada
           2: número máximo de iteraciones alcanzado
    suc: lista con la sucesión de aproximaciones {xn}
    '''

    # ========================================================================
    # PASO 1: VALIDACIÓN INICIAL Y ASIGNACIONES
    # ========================================================================

    # Verificar que el método sea válido
```

```
    metodos_validos = ['di', 'rf', 'fm']
    if meth not in metodos_validos:
        return None, -2, []

    # 1. Asignar fa ← f(a), fb ← f(b), [fxn ← 0]*
    fa = funx(a)
    fb = funx(b)
    fxn = 0  # Solo para Regula Falsi modificada (indicado con [...]*)

    # a) Si |fa| < F, finalizar r = a
    if abs(fa) < EF:
        return a, 1, [a]

    # b) Si |fb| < F, finalizar r = b
    if abs(fb) < EF:
        return b, 1, [b]

    # c) Si fa · fb > 0, finalizar: error no verifica Bolzano
    if fa * fb > 0:
        return None, -1, []

    # Inicialización de la lista de aproximaciones sucesivas
    suc = []
    info = 2  # Por defecto, asumimos que se alcanzará maxiter

    # ============================================================================
    # PASO 2: REPETIR UN MÁXIMO DE n ITERACIONES
    # ============================================================================
    for i in range(maxiter):

        # a) Elegir un valor xn   (an, bn)
        if meth == 'di':
            # 1) Bisección o dicotomía: xn = 1/2 (an + bn)
            xn = 0.5 * (a + b)

        elif meth == 'rf' or meth == 'fm':
            # 2) Regula Falsi: xn = a·f(b) - b·f(a) / [f(b) - f(a)]
            # Equivalente a: xn = a - fa / [(fb - fa) / (b - a)]
            xn = (a * fb - b * fa) / (fb - fa)

        # b) Asignar fx ← f(xn)
        fx = funx(xn)
        suc.append(xn)

        # c) Si |fx| < EF, finalizar r = xn
        if abs(fx) < EF:
            info = 1
```

```
            break

        # d) Si fx · fa > 0 entonces asignar fa ← fx, a ← x
        if fx * fa > 0:
            a = xn
            fa = fx
            # [Si fx · fxn > 0 entonces asignar fb ← 1/2 fb]* (solo para fm)
            if meth == 'fm' and i > 0 and fx * fxn > 0:
                fb = 0.5 * fb

        # e) sino asignar fb ← fx, b ← x
        else:
            b = xn
            fb = fx
            # [Si fx · fxn > 0 entonces asignar fa ← 1/2 fa]* (solo para fm)
            if meth == 'fm' and i > 0 and fx * fxn > 0:
                fa = 0.5 * fa

        # f) [Asignar fxn ← fx]* (solo para fm)
        if meth == 'fm':
            fxn = fx

        # g) Asignar xtol ← máx(EX, eX |xn|)
        xtol = max(Ex, ex * abs(xn))

        # h) Si |b - a| < xtol, finalizar r = xn
        if abs(b - a) < xtol:
            info = 0
            break

    # ============================================================================
    # PASO 3: ACABADO EL BUCLE SIN CONVERGENCIA
    # ============================================================================
    # Finalizar r = xn (la última aproximación calculada)
    r = suc[-1] if suc else None

    return r, info, suc
```

1.12.6 Validación

[22]:
```
# ============================================================================
# VALIDACIÓN DE LA FUNCIÓN ensolver
# ============================================================================


# Definimos la función f(x) = x·e^x - 4
def f(x):
```

```
    return x * np.exp(x) - 4

# Parámetros de las pruebas
a, b = 1, 2
maxiter = 10
Ex = 1e-3
ex = 1e-3
EF = 1e-6

print("=" * 70)
print("VALIDACIÓN DE ensolver: f(x) = x·e^x - 4 en [1, 2]")
print("=" * 70)
print(f"Parámetros: maxiter={maxiter}, Ex={Ex}, ex={ex}, EF={EF}")
print("=" * 70)

# ------------------------------------------------------------------------
# 1. MÉTODO DE BISECCIÓN (DICOTOMÍA)
# ------------------------------------------------------------------------
print("\n1. MÉTODO DE BISECCIÓN (DICOTOMÍA)")
print("-" * 70)

r_di, info_di, suc_di = ensolver(f, a, b, meth='di', maxiter=maxiter,
                                 Ex=Ex, ex=ex, EF=EF)

print(f"Raíz aproximada: {r_di}")
print(f"Código de terminación (info): {info_di}")
if info_di == 0:
    print("  → Tolerancia del intervalo alcanzada")
elif info_di == 1:
    print("  → Tolerancia de la función alcanzada")
elif info_di == 2:
    print("  → Número máximo de iteraciones alcanzado")

print(f"\nSucesión de aproximaciones ({len(suc_di)} iteraciones):")
print(", ".join([f"{x:.10g}" for x in suc_di]))

# Verificación del resultado esperado
print("\n  Resultado esperado: info=0")
print(f"  Verificación: info={info_di} {'  CORRECTO' if info_di == 0 else '␣
 ↪INCORRECTO'}")

# ------------------------------------------------------------------------
# 2. MÉTODO DE REGULA FALSI (POSICIÓN FALSA)
# ------------------------------------------------------------------------
print("\n\n2. MÉTODO DE REGULA FALSI (POSICIÓN FALSA)")
print("-" * 70)
```

```
r_rf, info_rf, suc_rf = ensolver(f, a, b, meth='rf', maxiter=maxiter,
                                 Ex=Ex, ex=ex, EF=EF)

print(f"Raíz aproximada: {r_rf}")
print(f"Código de terminación (info): {info_rf}")
if info_rf == 0:
    print("  → Tolerancia del intervalo alcanzada")
elif info_rf == 1:
    print("  → Tolerancia de la función alcanzada")
elif info_rf == 2:
    print("  → Número máximo de iteraciones alcanzado")

print(f"\nSucesión de aproximaciones ({len(suc_rf)} iteraciones):")
for i, x in enumerate(suc_rf):
    print(f"  x{i}: {x:.16g}")

# Verificación del resultado esperado
print("\n  Resultado esperado: info=2")
print(f"  Verificación: info={info_rf} {'  CORRECTO' if info_rf == 2 else '␣
 ↪INCORRECTO'}")

# ------------------------------------------------------------------------
# 3. MÉTODO DE REGULA FALSI MODIFICADA
# ------------------------------------------------------------------------
print("\n\n3. MÉTODO DE REGULA FALSI MODIFICADA")
print("-" * 70)

r_fm, info_fm, suc_fm = ensolver(f, a, b, meth='fm', maxiter=maxiter,
                                 Ex=Ex, ex=ex, EF=EF)

print(f"Raíz aproximada: {r_fm}")
print(f"Código de terminación (info): {info_fm}")
if info_fm == 0:
    print("  → Tolerancia del intervalo alcanzada")
elif info_fm == 1:
    print("  → Tolerancia de la función alcanzada")
elif info_fm == 2:
    print("  → Número máximo de iteraciones alcanzado")

print(f"\nSucesión de aproximaciones ({len(suc_fm)} iteraciones):")
for i, x in enumerate(suc_fm):
    print(f"  x{i}: {x:.16g}")

# Verificación del resultado esperado
print("\n  Resultado esperado: info=1")
print(f"  Verificación: info={info_fm} {'  CORRECTO' if info_fm == 1 else '␣
 ↪INCORRECTO'}")
```

```python
# ============================================================================
# ESTIMACIÓN DEL ERROR EN REGULA FALSI
# ============================================================================
print("\n\n" + "=" * 70)
print("ESTIMACIÓN DEL ERROR - REGULA FALSI")
print("=" * 70)

# Calculamos el estimador del error En = |rn / (1 - rn) * (xn - xn-1)|
# donde rn = (xn - xn-1) / (xn-1 - xn-2)

if len(suc_rf) >= 3:
    print("\nCálculo del estimador del error: En = |rn / (1 - rn) * (xn -␣
 ↪xn-1)|")
    print("donde rn = (xn - xn-1) / (xn-1 - xn-2)")
    print("-" * 70)

    errores = []

    for n in range(2, len(suc_rf)):
        # Calculamos rn
        xn = suc_rf[n]
        xn_1 = suc_rf[n-1]
        xn_2 = suc_rf[n-2]

        rn = (xn - xn_1) / (xn_1 - xn_2)

        # Calculamos En
        if abs(1 - rn) > 1e-14:  # Evitar división por cero
            En = abs(rn / (1 - rn) * (xn - xn_1))
            errores.append(En)
            print(f"  E{n}: {En:.8e}")
        else:
            print(f"  E{n}: No calculable (rn   1)")

    print("\n  Valores esperados:")
    print("En: 2.12617978e-02, 9.60141588e-03, 4.36605854e-03, 1.99207847e-03,")
    print("    9.10354281e-04, 4.16325177e-04, 1.90458933e-04, 8.71439410e-05")

    print("\nVerificación: Los valores deben ser del mismo orden de magnitud  ")

else:
    print("No hay suficientes iteraciones para calcular el error.")

# ============================================================================
# REPRESENTACIÓN GRÁFICA
# ============================================================================
```

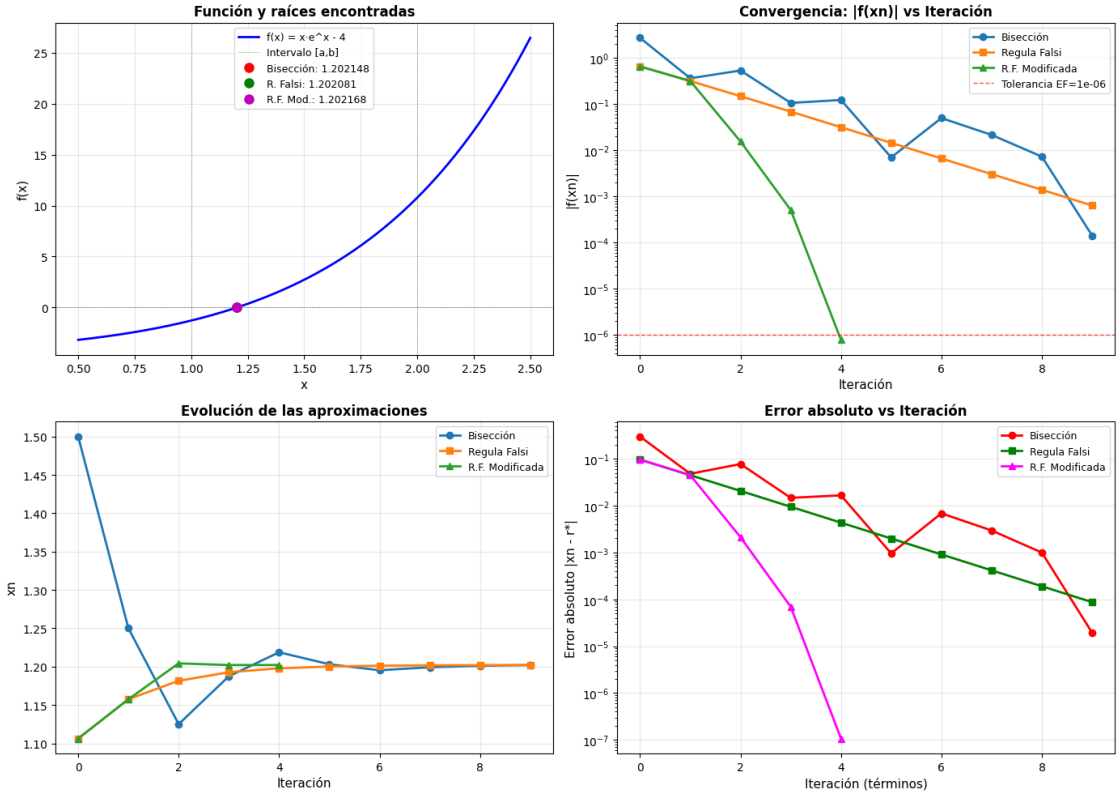
```
print("\n\n" + "=" * 70)
print("REPRESENTACIÓN GRÁFICA")
print("=" * 70)

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Gráfica 1: Función y raíces encontradas
ax1 = axes[0, 0]
x_plot = np.linspace(0.5, 2.5, 500)
y_plot = f(x_plot)
ax1.plot(x_plot, y_plot, 'b-', linewidth=2, label='f(x) = x·e^x - 4')
ax1.axhline(y=0, color='k', linestyle='--', linewidth=0.5, alpha=0.5)
ax1.axvline(x=a, color='g', linestyle='--', linewidth=0.5, alpha=0.5,␣
 ↪label='Intervalo [a,b]')
ax1.axvline(x=b, color='g', linestyle='--', linewidth=0.5, alpha=0.5)
ax1.plot(r_di, f(r_di), 'ro', markersize=8, label=f'Bisección: {r_di:.6f}')
ax1.plot(r_rf, f(r_rf), 'go', markersize=8, label=f'R. Falsi: {r_rf:.6f}')
ax1.plot(r_fm, f(r_fm), 'mo', markersize=8, label=f'R.F. Mod.: {r_fm:.6f}')
ax1.grid(True, alpha=0.3)
ax1.set_xlabel('x', fontsize=11)
ax1.set_ylabel('f(x)', fontsize=11)
ax1.set_title('Función y raíces encontradas', fontsize=12, fontweight='bold')
ax1.legend(fontsize=9)

# Gráfica 2: Convergencia de los métodos
ax2 = axes[0, 1]
ax2.semilogy(range(len(suc_di)), [abs(f(x)) for x in suc_di], 'o-',
             label='Bisección', linewidth=2, markersize=6)
ax2.semilogy(range(len(suc_rf)), [abs(f(x)) for x in suc_rf], 's-',
             label='Regula Falsi', linewidth=2, markersize=6)
ax2.semilogy(range(len(suc_fm)), [abs(f(x)) for x in suc_fm], '^-',
             label='R.F. Modificada', linewidth=2, markersize=6)
ax2.axhline(y=EF, color='r', linestyle='--', linewidth=1, alpha=0.7,␣
 ↪label=f'Tolerancia EF={EF}')
ax2.grid(True, alpha=0.3)
ax2.set_xlabel('Iteración', fontsize=11)
ax2.set_ylabel('|f(xn)|', fontsize=11)
ax2.set_title('Convergencia: |f(xn)| vs Iteración', fontsize=12,␣
 ↪fontweight='bold')
ax2.legend(fontsize=9)

# Gráfica 3: Evolución de las aproximaciones
ax3 = axes[1, 0]
ax3.plot(range(len(suc_di)), suc_di, 'o-', label='Bisección', linewidth=2,␣
 ↪markersize=6)
ax3.plot(range(len(suc_rf)), suc_rf, 's-', label='Regula Falsi', linewidth=2,␣
 ↪markersize=6)
```

```
ax3.plot(range(len(suc_fm)), suc_fm, '^-', label='R.F. Modificada',␣
 ↪linewidth=2, markersize=6)
ax3.grid(True, alpha=0.3)
ax3.set_xlabel('Iteración', fontsize=11)
ax3.set_ylabel('xn', fontsize=11)
ax3.set_title('Evolución de las aproximaciones', fontsize=12, fontweight='bold')
ax3.legend(fontsize=9)

# Gráfica 4: Error absoluto vs iteración (para los tres métodos)
ax4 = axes[1, 1]

# Para calcular el error absoluto necesitamos la raíz exacta
# Podemos tomarla como la mejor aproximación obtenida (R.F. Modificada con más␣
 ↪precisión)
r_exacta_temp, _, _ = ensolver(f, a, b, meth='fm', maxiter=100, Ex=1e-15,␣
 ↪ex=1e-15, EF=1e-15)
raiz_exacta = r_exacta_temp

# Calculamos errores absolutos para cada método
error_abs_di = [abs(x - raiz_exacta) for x in suc_di]
error_abs_rf = [abs(x - raiz_exacta) for x in suc_rf]
error_abs_fm = [abs(x - raiz_exacta) for x in suc_fm]

ax4.semilogy(range(len(error_abs_di)), error_abs_di, 'o-',
             label='Bisección', linewidth=2, markersize=6, color='red')
ax4.semilogy(range(len(error_abs_rf)), error_abs_rf, 's-',
             label='Regula Falsi', linewidth=2, markersize=6, color='green')
ax4.semilogy(range(len(error_abs_fm)), error_abs_fm, '^-',
             label='R.F. Modificada', linewidth=2, markersize=6,␣
 ↪color='magenta')
ax4.grid(True, alpha=0.3)
ax4.set_xlabel('Iteración (términos)', fontsize=11)
ax4.set_ylabel('Error absoluto |xn - r*|', fontsize=11)
ax4.set_title('Error absoluto vs Iteración', fontsize=12, fontweight='bold')
ax4.legend(fontsize=9)

print(f"\nRaíz de referencia (calculada con alta precisión): {raiz_exacta:.
 ↪15f}")

plt.tight_layout()
plt.show()
```

```
======================================================================
VALIDACIÓN DE ensolver: f(x) = x·e^x - 4 en [1, 2]
======================================================================
Parámetros: maxiter=10, Ex=0.001, ex=0.001, EF=1e-06
======================================================================
```

```
1. MÉTODO DE BISECCIÓN (DICOTOMÍA)
----------------------------------------------------------------------
Raíz aproximada: 1.2021484375
Código de terminación (info): 0
  → Tolerancia del intervalo alcanzada

Sucesión de aproximaciones (10 iteraciones):
1.5, 1.25, 1.125, 1.1875, 1.21875, 1.203125, 1.1953125, 1.19921875, 1.201171875,
1.202148438

 Resultado esperado: info=0
 Verificación: info=0   CORRECTO


2. MÉTODO DE REGULA FALSI (POSICIÓN FALSA)
----------------------------------------------------------------------
Raíz aproximada: 1.2020807414154477
Código de terminación (info): 2
  → Número máximo de iteraciones alcanzado

Sucesión de aproximaciones (10 iteraciones):
  x0: 1.106279950238179
  x1: 1.157526225421109
  x2: 1.181573907115875
  x3: 1.192708649893153
  x4: 1.197831820428369
  x5: 1.200182100346257
  x6: 1.201258841484738
  x7: 1.201751825177766
  x8: 1.201977472322062
  x9: 1.202080741415448

 Resultado esperado: info=2
 Verificación: info=2   CORRECTO


3. MÉTODO DE REGULA FALSI MODIFICADA
----------------------------------------------------------------------
Raíz aproximada: 1.202167765998007
Código de terminación (info): 1
  → Tolerancia de la función alcanzada

Sucesión de aproximaciones (5 iteraciones):
  x0: 1.106279950238179
  x1: 1.157526225421109
  x2: 1.204286847493607
  x3: 1.202098249141321
```

```
  x4: 1.202167765998007

  Resultado esperado: info=1
  Verificación: info=1   CORRECTO


======================================================================
ESTIMACIÓN DEL ERROR - REGULA FALSI
======================================================================

Cálculo del estimador del error: En = |rn / (1 - rn) * (xn - xn-1)|
donde rn = (xn - xn-1) / (xn-1 - xn-2)
----------------------------------------------------------------------
  E2: 2.12617978e-02
  E3: 9.60141588e-03
  E4: 4.36605854e-03
  E5: 1.99207847e-03
  E6: 9.10354281e-04
  E7: 4.16325177e-04
  E8: 1.90458933e-04
  E9: 8.71439410e-05

  Valores esperados:
En: 2.12617978e-02, 9.60141588e-03, 4.36605854e-03, 1.99207847e-03,
    9.10354281e-04, 4.16325177e-04, 1.90458933e-04, 8.71439410e-05

Verificación: Los valores deben ser del mismo orden de magnitud


======================================================================
REPRESENTACIÓN GRÁFICA
======================================================================

Raíz de referencia (calculada con alta precisión): 1.202167873197043
```



1.12.7 Implementación

[24]:
```
def enlsteffensen(funx, x0, maxiter=128, Ex=1e-9, ex=1e-5, EF=1e-12):
    '''
    Resuelve ecuaciones no lineales mediante el método de Steffensen.

    El método de Steffensen es una variante del método de punto fijo que acelera
    la convergencia mediante la función:

    g_st(x) = x - f(x) / h(x)

    donde h(x) = f(x + f(x)) / f(x) - 1

    CONDICIÓN DE CONVERGENCIA:
    La condición -1 < g'(x) < 0 es suficiente para garantizar la convergencia.
    Valores de gx muy alejados del rango [-2, 1] indican probable divergencia.

    PARÁMETROS DE LA FUNCIÓN
    ------------------------
    funx: función cuyo cero se desea determinar
```

```
    x0: punto inicial de la iteración
    maxiter: número máximo de iteraciones permitidas [128 por defecto]
    Ex: tolerancia absoluta del punto [1e-9 por defecto]
    ex: tolerancia relativa del punto [1e-5 por defecto]
    EF: tolerancia para el valor de la función [1e-12 por defecto]

    RETORNO
    -------
    r: cero aproximado de la función
    info: código de terminación
          0: tolerancia del punto (xtol) alcanzada
          1: tolerancia de la función (EF) alcanzada
          2: número máximo de iteraciones alcanzado
    suc: lista con la sucesión de aproximaciones {xn}
    '''

    # ============================================================================
    # INICIALIZACIÓN
    # ============================================================================

    # Lista para almacenar la sucesión de aproximaciones
    suc = []

    # Punto inicial
    xk_1 = x0  # xk-1 (punto anterior)

    # Código de terminación por defecto (maxiter alcanzado)
    info = 2

    # ============================================================================
    # ALGORITMO: Repetir para k desde 1 hasta n (maxiter)
    # ============================================================================

    for k in range(1, maxiter + 1):

        # * Paso 1: Asignar fx ← f(xk-1)
        fx = funx(xk_1)

        # * Paso 2: Si |fx| < F, finalizar r = xk-1
        if abs(fx) < EF:
            r = xk_1
            info = 1  # Tolerancia de la función alcanzada
            return r, info, suc

        # * Paso 3: Asignar fy ← f(xk-1 + fx), gx ← fy/fx - 1
        # Evaluamos la función en el punto perturbado
        fy = funx(xk_1 + fx)
```

```
        # Calculamos el factor gx = h(x)
        # Este es el denominador en la fórmula de Steffensen
        gx = fy / fx - 1

        # * Paso 4: Si gx   [-2, 1], aviso probable divergencia
        # Valores fuera de este rango pueden indicar problemas de convergencia
        if gx < -2 or gx > 1:
            print(f"  Iteración {k}: Aviso de probable divergencia (gx = {gx:.
↪6f})")

        # Verificar que gx no sea cero (evitar división por cero)
        if abs(gx) < 1e-15:
            print(f"  Iteración {k}: gx   0, no se puede continuar")
            r = xk_1
            info = 2
            return r, info, suc

        # * Paso 5: Calcular xk = xk-1 - fx/gx
        # Esta es la fórmula de iteración de Steffensen
        xk = xk_1 - fx / gx

        # Añadimos la nueva aproximación a la sucesión
        suc.append(xk)

        # * Paso 6: Calcular  x = máx(Ex, ex |xk|)
        # Tolerancia adaptativa que combina absoluta y relativa
        epsilon_x = max(Ex, ex * abs(xk))

        # * Paso 7: Si |xk - xk-1| < x, finalizar r = xk
        if abs(xk - xk_1) < epsilon_x:
            r = xk
            info = 0  # Tolerancia del punto alcanzada
            return r, info, suc

        # Actualizamos xk-1 para la siguiente iteración
        xk_1 = xk

    # ============================================================================
    # PASO FINAL: Acabado el bucle sin convergencia
    # ============================================================================
    # Finalizar r = xn (última aproximación calculada)
    r = suc[-1] if suc else x0
    info = 2  # Número máximo de iteraciones alcanzado

    return r, info, suc
```

1.12.8 Validación

[26]:
```
# ============================================================================
# VALIDACIÓN DE LA FUNCIÓN enlsteffensen
# ============================================================================

import numpy as np
import matplotlib.pyplot as plt

# Definimos la función f(x) = x·e^x - 4
def f(x):
    return x * np.exp(x) - 4

# Parámetros de las pruebas
maxiter = 20
Ex = 1e-8
ex = 1e-6
EF = 1e-10

print("=" * 70)
print("VALIDACIÓN DE enlsteffensen: f(x) = x·e^x - 4")
print("=" * 70)
print(f"Parámetros: maxiter={maxiter}, Ex={Ex}, ex={ex}, EF={EF}")
print("=" * 70)

# Puntos iniciales a probar
puntos_iniciales = [1.0, 2.0, 1.5]
info_esperados = [0, 2, 0]

# Lista para almacenar resultados
resultados = []

# ----------------------------------------------------------------------------
# PRUEBAS CON DIFERENTES PUNTOS INICIALES
# ----------------------------------------------------------------------------

for i, x0 in enumerate(puntos_iniciales):
    print(f"\n{'='*70}")
    print(f"PRUEBA {i+1}: x0 = {x0}")
    print("="*70)

    r, info, suc = enlsteffensen(f, x0, maxiter=maxiter, Ex=Ex, ex=ex, EF=EF)

    print(f"\nRaíz aproximada: {r:.15f}")
    print(f"Código de terminación (info): {info}")

    if info == 0:
        print("  → Tolerancia del punto alcanzada")
```

```
    elif info == 1:
        print("  → Tolerancia de la función alcanzada")
    elif info == 2:
        print("  → Número máximo de iteraciones alcanzado")

    print(f"\nNúmero de iteraciones: {len(suc)}")
    print(f"Valor de f(r): {f(r):.2e}")

    print(f"\nSucesión de aproximaciones:")
    # Mostramos todas las aproximaciones en formato compacto
    suc_completa = [x0] + suc
    for j in range(0, len(suc_completa), 5):
        grupo = suc_completa[j:min(j+5, len(suc_completa))]
        print("  " + ", ".join([f"{x:.8f}" for x in grupo]))

    # Verificación del resultado esperado
    print(f"\n  Resultado esperado: info={info_esperados[i]}")
    print(f"  Verificación: info={info} {'  CORRECTO' if info ==␣
 ↪info_esperados[i] else '  INCORRECTO'}")

    # Guardamos los resultados para graficar
    resultados.append({
        'x0': x0,
        'r': r,
        'info': info,
        'suc': suc_completa
    })

# ------------------------------------------------------------------------
# ANÁLISIS DE VARIACIÓN AL INTERCAMBIAR ORDEN
# ------------------------------------------------------------------------

print("\n\n" + "="*70)
print("ANÁLISIS: ¿Varían los resultados al intercambiar el orden?")
print("="*70)

print(f"\nRaíz obtenida con x0=1.0: {resultados[0]['r']:.15f}")
print(f"Raíz obtenida con x0=2.0: {resultados[1]['r']:.15f}")
print(f"Raíz obtenida con x0=1.5: {resultados[2]['r']:.15f}")

print(f"\nDiferencia |r(x0=1.0) - r(x0=1.5)|: {abs(resultados[0]['r'] -␣
 ↪resultados[2]['r']):.2e}")
print(f"Diferencia |r(x0=1.0) - r(x0=2.0)|: {abs(resultados[0]['r'] -␣
 ↪resultados[1]['r']):.2e}")

if resultados[1]['info'] == 2:
    print("\n  OBSERVACIÓN IMPORTANTE:")
```

```
    print("  Con x0=2.0, el método NO converge en 20 iteraciones.")
    print("  Esto demuestra que la elección del punto inicial es CRÍTICA")
    print("  en el método de Steffensen.")
else:
    print("\n  Todos los puntos iniciales convergen a la misma raíz.")

# ============================================================================
# REPRESENTACIÓN GRÁFICA
# ============================================================================

print("\n\n" + "="*70)
print("REPRESENTACIÓN GRÁFICA")
print("="*70)

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Colores para cada punto inicial
colores = ['red', 'green', 'blue']
marcadores = ['o', 's', '^']
etiquetas = ['x  = 1.0', 'x  = 2.0', 'x  = 1.5']

# Gráfica 1: Función y trayectorias de convergencia
ax1 = axes[0, 0]
x_plot = np.linspace(0.5, 2.5, 500)
y_plot = f(x_plot)
ax1.plot(x_plot, y_plot, 'b-', linewidth=2, label='f(x) = x·e^x - 4')
ax1.axhline(y=0, color='k', linestyle='--', linewidth=0.5, alpha=0.5)
ax1.axvline(x=1, color='gray', linestyle='--', linewidth=0.5, alpha=0.3)
ax1.axvline(x=2, color='gray', linestyle='--', linewidth=0.5, alpha=0.3)

for i, res in enumerate(resultados):
    if res['info'] != 2:  # Solo graficamos si convergió
        ax1.plot(res['r'], f(res['r']), marcadores[i], markersize=10,
                 color=colores[i], label=f"{etiquetas[i]}: r={res['r']:.6f}")

ax1.grid(True, alpha=0.3)
ax1.set_xlabel('x', fontsize=11)
ax1.set_ylabel('f(x)', fontsize=11)
ax1.set_title('Función y raíces encontradas', fontsize=12, fontweight='bold')
ax1.legend(fontsize=9)

# Gráfica 2: Convergencia de los métodos (|f(xn)|)
ax2 = axes[0, 1]
for i, res in enumerate(resultados):
    valores_f = [abs(f(x)) for x in res['suc']]
    ax2.semilogy(range(len(valores_f)), valores_f,
                 marker=marcadores[i], linestyle='-', label=etiquetas[i],
```

```
                linewidth=2, markersize=6, color=colores[i])

ax2.axhline(y=EF, color='r', linestyle='--', linewidth=1, alpha=0.7,
           label=f'Tolerancia EF={EF}')
ax2.grid(True, alpha=0.3)
ax2.set_xlabel('Iteración', fontsize=11)
ax2.set_ylabel('|f(xn)|', fontsize=11)
ax2.set_title('Convergencia: |f(xn)| vs Iteración', fontsize=12,␣
 ↪fontweight='bold')
ax2.legend(fontsize=9)

# Gráfica 3: Evolución de las aproximaciones
ax3 = axes[1, 0]
for i, res in enumerate(resultados):
    ax3.plot(range(len(res['suc'])), res['suc'],
             marker=marcadores[i], linestyle='-', label=etiquetas[i],
             linewidth=2, markersize=6, color=colores[i])

ax3.grid(True, alpha=0.3)
ax3.set_xlabel('Iteración', fontsize=11)
ax3.set_ylabel('xn', fontsize=11)
ax3.set_title('Evolución de las aproximaciones', fontsize=12, fontweight='bold')
ax3.legend(fontsize=9)

# Gráfica 4: Error absoluto vs iteración
ax4 = axes[1, 1]

# Raíz de referencia (la mejor aproximación obtenida)
raiz_ref = resultados[2]['r']  # Usamos la de x0=1.5 que convergió bien

for i, res in enumerate(resultados):
    errores = [abs(x - raiz_ref) for x in res['suc']]
    ax4.semilogy(range(len(errores)), errores,
                 marker=marcadores[i], linestyle='-', label=etiquetas[i],
                 linewidth=2, markersize=6, color=colores[i])

ax4.grid(True, alpha=0.3)
ax4.set_xlabel('Iteración', fontsize=11)
ax4.set_ylabel('Error absoluto |xn - r*|', fontsize=11)
ax4.set_title('Error absoluto vs Iteración', fontsize=12, fontweight='bold')
ax4.legend(fontsize=9)

plt.tight_layout()
plt.show()

print("\nGráficas generadas exitosamente ")
```

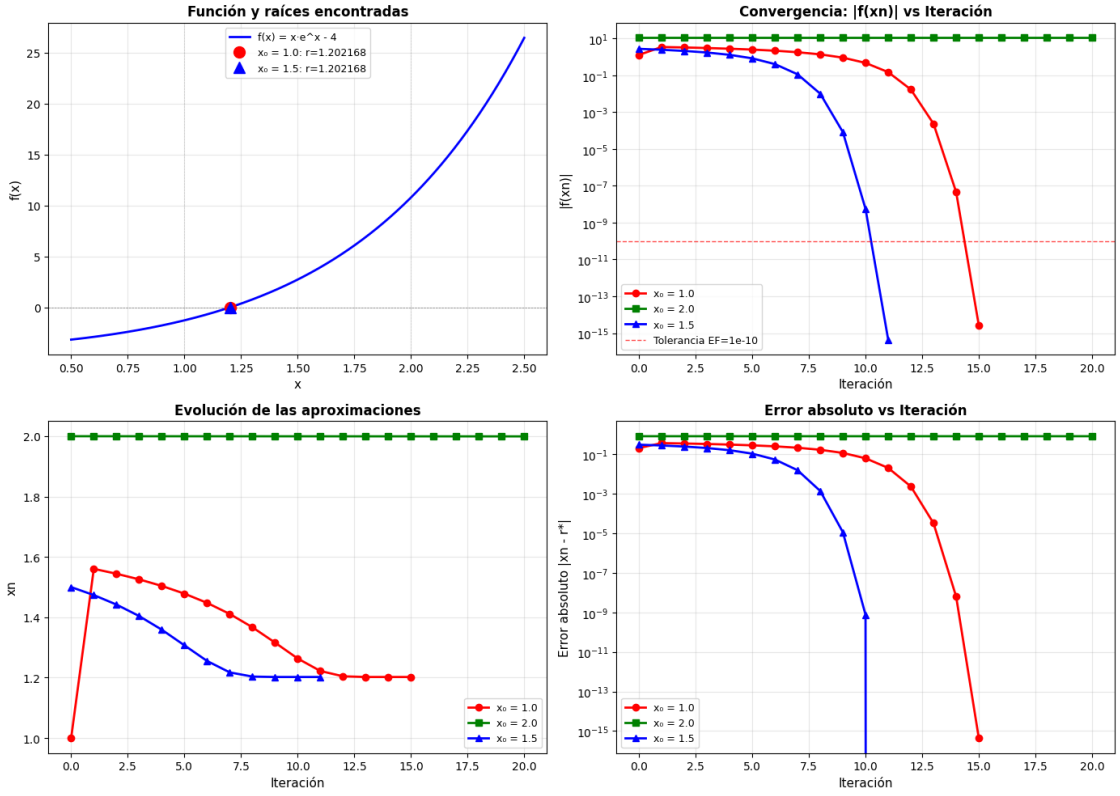
```python
# ============================================================================
# RESUMEN FINAL
# ============================================================================

print("\n\n" + "="*70)
print("RESUMEN DE RESULTADOS")
print("="*70)

tabla = """

  x          info     Iteraciones    Raíz encontrada        f(r)
                                        """

for i, res in enumerate(resultados):
    tabla += f"\n  {res['x0']:<8.1f}   {res['info']:^7}   {len(res['suc'])-1:
  ↪^10}   {res['r']:^20.15f}   {f(res['r']):^12.2e}  "

tabla += """

"""

print(tabla)

print("\n  Conclusión:")
print("   • El método de Steffensen es muy sensible al punto inicial")
print("   • x =1.0 y x =1.5 convergen rápidamente (info=0)")
print("   • x =2.0 diverge o converge muy lentamente (info=2)")
print("   • La elección del punto inicial es CRUCIAL para el éxito del método")
```

```
======================================================================
VALIDACIÓN DE enlsteffensen: f(x) = x·e^x - 4
======================================================================
Parámetros: maxiter=20, Ex=1e-08, ex=1e-06, EF=1e-10
======================================================================

======================================================================
PRUEBA 1: x0 = 1.0
======================================================================
  Iteración 1: Aviso de probable divergencia (gx = 2.286645)
  Iteración 2: Aviso de probable divergencia (gx = 211.760273)
  Iteración 3: Aviso de probable divergencia (gx = 173.586580)
  Iteración 4: Aviso de probable divergencia (gx = 138.947808)
  Iteración 5: Aviso de probable divergencia (gx = 108.091771)
  Iteración 6: Aviso de probable divergencia (gx = 81.266426)
  Iteración 7: Aviso de probable divergencia (gx = 58.694925)
  Iteración 8: Aviso de probable divergencia (gx = 40.530300)
  Iteración 9: Aviso de probable divergencia (gx = 26.783300)
  Iteración 10: Aviso de probable divergencia (gx = 17.231675)
```

```
  Iteración 11: Aviso de probable divergencia (gx = 11.360070)
  Iteración 12: Aviso de probable divergencia (gx = 8.408088)
  Iteración 13: Aviso de probable divergencia (gx = 7.443178)
  Iteración 14: Aviso de probable divergencia (gx = 7.328918)
  Iteración 15: Aviso de probable divergencia (gx = 7.327323)

Raíz aproximada: 1.202167873197043
Código de terminación (info): 0
  → Tolerancia del punto alcanzada

Número de iteraciones: 15
Valor de f(r): 2.66e-15

Sucesión de aproximaciones:
  1.00000000, 1.56052349, 1.54432526, 1.52568978, 1.50398666
  1.47838502, 1.44781908, 1.41103954, 1.36698445, 1.31608089
  1.26342350, 1.22210553, 1.20447656, 1.20219999, 1.20216788
  1.20216787

  Resultado esperado: info=0
  Verificación: info=0   CORRECTO

======================================================================
PRUEBA 2: x0 = 2.0
======================================================================
  Iteración 1: Aviso de probable divergencia (gx = 420132.420577)
  Iteración 2: Aviso de probable divergencia (gx = 419885.424273)
  Iteración 3: Aviso de probable divergencia (gx = 419638.449350)
  Iteración 4: Aviso de probable divergencia (gx = 419391.495818)
  Iteración 5: Aviso de probable divergencia (gx = 419144.563687)
  Iteración 6: Aviso de probable divergencia (gx = 418897.652967)
  Iteración 7: Aviso de probable divergencia (gx = 418650.763668)
  Iteración 8: Aviso de probable divergencia (gx = 418403.895800)
  Iteración 9: Aviso de probable divergencia (gx = 418157.049373)
  Iteración 10: Aviso de probable divergencia (gx = 417910.224397)
  Iteración 11: Aviso de probable divergencia (gx = 417663.420881)
  Iteración 12: Aviso de probable divergencia (gx = 417416.638836)
  Iteración 13: Aviso de probable divergencia (gx = 417169.878271)
  Iteración 14: Aviso de probable divergencia (gx = 416923.139197)
  Iteración 15: Aviso de probable divergencia (gx = 416676.421624)
  Iteración 16: Aviso de probable divergencia (gx = 416429.725561)
  Iteración 17: Aviso de probable divergencia (gx = 416183.051019)
  Iteración 18: Aviso de probable divergencia (gx = 415936.398007)
  Iteración 19: Aviso de probable divergencia (gx = 415689.766536)
  Iteración 20: Aviso de probable divergencia (gx = 415443.156616)

Raíz aproximada: 1.999484292103323
Código de terminación (info): 2
```

```
  → Número máximo de iteraciones alcanzado

Número de iteraciones: 20
Valor de f(r): 1.08e+01

Sucesión de aproximaciones:
  2.00000000, 1.99997435, 1.99994868, 1.99992300, 1.99989730
  1.99987159, 1.99984587, 1.99982013, 1.99979438, 1.99976862
  1.99974284, 1.99971705, 1.99969124, 1.99966542, 1.99963959
  1.99961374, 1.99958788, 1.99956200, 1.99953611, 1.99951021
  1.99948429

  Resultado esperado: info=2
  Verificación: info=2   CORRECTO

======================================================================
PRUEBA 3: x0 = 1.5
======================================================================
  Iteración 1: Aviso de probable divergencia (gx = 103.315493)
  Iteración 2: Aviso de probable divergencia (gx = 77.187640)
  Iteración 3: Aviso de probable divergencia (gx = 55.345729)
  Iteración 4: Aviso de probable divergencia (gx = 37.924762)
  Iteración 5: Aviso de probable divergencia (gx = 24.902895)
  Iteración 6: Aviso de probable divergencia (gx = 16.010954)
  Iteración 7: Aviso de probable divergencia (gx = 10.685731)
  Iteración 8: Aviso de probable divergencia (gx = 8.136822)
  Iteración 9: Aviso de probable divergencia (gx = 7.395481)
  Iteración 10: Aviso de probable divergencia (gx = 7.327880)
  Iteración 11: Aviso de probable divergencia (gx = 7.327322)

Raíz aproximada: 1.202167873197043
Código de terminación (info): 0
  → Tolerancia del punto alcanzada

Número de iteraciones: 11
Valor de f(r): -4.44e-16

Sucesión de aproximaciones:
  1.50000000, 1.47364835, 1.44213205, 1.40419250, 1.35888688
  1.30714206, 1.25526070, 1.21741587, 1.20353159, 1.20217910
  1.20216787, 1.20216787

  Resultado esperado: info=0
  Verificación: info=0   CORRECTO


======================================================================
ANÁLISIS: ¿Varían los resultados al intercambiar el orden?
```

```
======================================================================

Raíz obtenida con x0=1.0: 1.202167873197043
Raíz obtenida con x0=2.0: 1.999484292103323
Raíz obtenida con x0=1.5: 1.202167873197043

Diferencia |r(x0=1.0) - r(x0=1.5)|: 4.44e-16
Diferencia |r(x0=1.0) - r(x0=2.0)|: 7.97e-01

  OBSERVACIÓN IMPORTANTE:
   Con x0=2.0, el método NO converge en 20 iteraciones.
   Esto demuestra que la elección del punto inicial es CRÍTICA
   en el método de Steffensen.


======================================================================
REPRESENTACIÓN GRÁFICA
======================================================================
```




```
Gráficas generadas exitosamente
```

```
======================================================================
RESUMEN DE RESULTADOS
======================================================================
```

x	info	Iteraciones	Raíz encontrada	f(r)
1.0	0	15	1.202167873197043	2.66e-15
2.0	2	20	1.999484292103323	1.08e+01
1.5	0	11	1.202167873197043	-4.44e-16

Conclusión:

- El método de Steffensen es muy sensible al punto inicial
- x =1.0 y x =1.5 convergen rápidamente (info=0)
- x =2.0 diverge o converge muy lentamente (info=2)
- La elección del punto inicial es CRUCIAL para el éxito del método

---

1.12.9 Implementación

[ ]:
```python
def sor_interval(A):
    '''
    Calcula el intervalo de convergencia del método SOR (Successive Over-Relaxation)
    y los valores óptimos del parámetro de relajación  .

    El método SOR es una variante del método de Gauss-Seidel que introduce un
    parámetro de relajación  para acelerar la convergencia. La matriz de iteración SOR es:

    M_SOR = (D -  L)^(-1) * [(1- )D +  U]

    donde A = D - L - U (descomposición en diagonal, triangular inferior y superior).

    TEOREMA DE KAHAN:
    Si A es definida positiva, el método SOR converge para    (0, 2).

    PARÁMETROS DE LA FUNCIÓN
    ------------------------
    A: matriz cuadrada a estudiar (numpy array)

    RETORNO
    -------
    inter: intervalo [ i,  f] donde el método es convergente
```

```
        (lista vacía [] si no existe convergencia)
ropt: valor del radio espectral mínimo _mín
      (None si no existe convergencia)
wopt: valor óptimo de  que minimiza el radio espectral,
      redondeado a la centésima
      (None si no existe convergencia)
'''

# ============================================================================
# PASO 0: DESCOMPOSICIÓN DE LA MATRIZ A = D - L - U
# ============================================================================

# D: matriz diagonal con los elementos de la diagonal de A
D = np.diag(np.diag(A))

# L: parte triangular inferior de A (con signo negativo)
# -np.tril(A, -1) extrae la parte estrictamente triangular inferior
L = -np.tril(A, -1)

# U: parte triangular superior de A (con signo negativo)
# -np.triu(A, 1) extrae la parte estrictamente triangular superior
U = -np.triu(A, 1)

# ============================================================================
# PASO 1: CREAR UN VECTOR W DE VALORES POSIBLES DE
# ============================================================================

# Según el Teorema de Kahan, si A es definida positiva,
# el método SOR converge para    (0, 2)
# Creamos valores equidistantes una centésima: 0.00, 0.01, 0.02, ..., 1.99
W = np.arange(0, 2, 0.01)

# ============================================================================
# PASO 2: ASIGNAR VALORES INICIALES
# ============================================================================

# Listas para almacenar los valores de  válidos (convergentes)
# y sus correspondientes radios espectrales
w_validos = []      # Valores de  donde (M_SOR) < 1
radios_espectrales = []  # Radios espectrales correspondientes

# ============================================================================
# PASO 3: PARA CADA VALOR w DE W
# ============================================================================

for w in W:
```

```
        # a) Calcular la matriz SOR
        # M_SOR = (D -  L)^(-1) * [(1- )D +  U]

        try:
            # Calculamos (D -  L)^(-1)
            matriz_inv = la.inv(D - w * L)

            # Calculamos M_SOR = (D -  L)^(-1) * [(1- )D +  U]
            M_SOR = np.dot(matriz_inv, (1 - w) * D + w * U)

            # b) Calcular sus autovalores y radio espectral
            # El radio espectral  (M) es el máximo del valor absoluto de los␣
↪autovalores
            autovalores = la.eigvals(M_SOR)
            radio_espectral = max(abs(autovalores))

            # c) Asignar como válido si   < 1
            # El método converge si y solo si  (M_SOR) < 1
            if radio_espectral < 1:
                w_validos.append(w)
                radios_espectrales.append(radio_espectral)

        except la.LinAlgError:
            # Si (D -  L) no es invertible, continuamos con el siguiente valor
            continue

    # ============================================================================
    # PASO 4: DEVOLVER LOS VALORES PEDIDOS
    # ============================================================================

    # Verificar si existe algún valor de   para el cual el método converge
    if len(radios_espectrales) == 0:
        # No hay convergencia para ningún valor de
        return [], None, None

    # d) Comparar y/o actualizar el intervalo,  _mín y  ( _mín)

    # Convertimos a array para facilitar operaciones
    w_validos = np.array(w_validos)
    radios_espectrales = np.array(radios_espectrales)

    # Encontrar el radio espectral mínimo
    rho_min = radios_espectrales.min()

    # Encontrar el índice donde se alcanza el mínimo
    idx_min = radios_espectrales.argmin()
```

```
    # El valor óptimo de  es el que minimiza el radio espectral
    w_optimo = w_validos[idx_min]

    # El intervalo de convergencia es [_inicial, _final]
    intervalo = [w_validos.min(), w_validos.max()]

    # Redondear w_optimo a la centésima (ya debería estarlo por construcción)
    w_optimo = round(w_optimo, 2)

    return intervalo, rho_min, w_optimo
```

1.12.10 Validación

[ ]:
```
# ============================================================================
# VALIDACIÓN DE LA FUNCIÓN sor_interval
# ============================================================================



print("=" * 70)
print("VALIDACIÓN DE sor_interval: Método SOR")
print("=" * 70)

# ----------------------------------------------------------------------------
# DATOS DE VERIFICACIÓN
# ----------------------------------------------------------------------------

# Sistema de ecuaciones: Ax = b
# | 1  2  3 | | x |   |  1 |
# | 1  3  5 | | y | = |  0 |
# | 1  4  8 | | z |   | -2 |

A = np.array([
    [1, 2, 3],
    [1, 3, 5],
    [1, 4, 8]
], dtype=float)

b = np.array([1, 0, -2], dtype=float)

print("\nMatriz del sistema A:")
print(A)
print("\nVector independiente b:")
print(b)

# ----------------------------------------------------------------------------
# CÁLCULO DEL INTERVALO DE CONVERGENCIA Y VALORES ÓPTIMOS
```

```python
# ------------------------------------------------------------------------

print("\n" + "=" * 70)
print("CÁLCULO DEL INTERVALO DE CONVERGENCIA SOR")
print("=" * 70)

intervalo, rho_min, w_optimo = sor_interval(A)

print(f"\nResultados obtenidos:")
print(f"  Intervalo de convergencia: {intervalo}")
print(f"  Radio espectral mínimo ( _mín): {rho_min}")
print(f"  Valor óptimo de  : {w_optimo}")

# ------------------------------------------------------------------------
# VERIFICACIÓN CON VALORES ESPERADOS
# ------------------------------------------------------------------------

print("\n" + "=" * 70)
print("VERIFICACIÓN CON VALORES ESPERADOS")
print("=" * 70)

intervalo_esperado = [0.01, 1.91]
rho_min_esperado = 0.6714312623707218
w_optimo_esperado = 1.53

print(f"\nValores esperados:")
print(f"  Intervalo de convergencia: {intervalo_esperado}")
print(f"  Radio espectral mínimo ( _mín): {rho_min_esperado}")
print(f"  Valor óptimo de  : {w_optimo_esperado}")

print("\n" + "-" * 70)
print("COMPARACIÓN DE RESULTADOS")
print("-" * 70)

# Verificar intervalo
error_intervalo_min = abs(intervalo[0] - intervalo_esperado[0])
error_intervalo_max = abs(intervalo[1] - intervalo_esperado[1])

print(f"\nIntervalo:")
print(f"  Obtenido: [{intervalo[0]:.2f}, {intervalo[1]:.2f}]")
print(f"  Esperado: [{intervalo_esperado[0]:.2f}, {intervalo_esperado[1]:.2f}]")
print(f"  Errores:  []{error_intervalo_min:.2e}, {error_intervalo_max:.2e}]")

if error_intervalo_min < 1e-10 and error_intervalo_max < 1e-10:
    print(f"    CORRECTO - Intervalo coincide exactamente")
else:
    print(f"    Diferencia pequeña (tolerancia numérica)")
```

```
# Verificar _mín
error_rho = abs(rho_min - rho_min_esperado)
print(f"\nRadio espectral mínimo ( _mín):")
print(f"  Obtenido: {rho_min:.16f}")
print(f"  Esperado: {rho_min_esperado:.16f}")
print(f"  Error:    {error_rho:.2e}")

if error_rho < 1e-10:
    print(f"   CORRECTO - Radio espectral coincide con alta precisión")
else:
    print(f"   Diferencia: {error_rho:.2e}")

# Verificar  óptimo
print(f"\nValor óptimo de  :")
print(f"  Obtenido: {w_optimo}")
print(f"  Esperado: {w_optimo_esperado}")

if w_optimo == w_optimo_esperado:
    print(f"   CORRECTO -  óptimo coincide exactamente")
else:
    print(f"   INCORRECTO -  óptimo no coincide")

# ------------------------------------------------------------------------
# ANÁLISIS ADICIONAL: GRÁFICA DEL RADIO ESPECTRAL VS
# ------------------------------------------------------------------------

print("\n" + "=" * 70)
print("ANÁLISIS GRÁFICO: RADIO ESPECTRAL vs  ")
print("=" * 70)

import matplotlib.pyplot as plt

# Recalcular para todos los valores de
D = np.diag(np.diag(A))
L = -np.tril(A, -1)
U = -np.triu(A, 1)

W = np.arange(0, 2, 0.01)
radios = []

for w in W:
    try:
        matriz_inv = la.inv(D - w * L)
        M_SOR = np.dot(matriz_inv, (1 - w) * D + w * U)
        autovalores = la.eigvals(M_SOR)
        radio = max(abs(autovalores))
```

```
        radios.append(radio)
    except la.LinAlgError:
        radios.append(np.nan)

radios = np.array(radios)

# Crear la gráfica
fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(W, radios, 'b-', linewidth=2, label=' (M_SOR)')
ax.axhline(y=1, color='r', linestyle='--', linewidth=1.5,
           label='  = 1 (límite de convergencia)', alpha=0.7)
ax.axvline(x=w_optimo, color='g', linestyle='--', linewidth=1.5,
           label=f'  óptimo = {w_optimo}', alpha=0.7)
ax.plot(w_optimo, rho_min, 'go', markersize=12,
        label=f' _mín = {rho_min:.4f}', zorder=5)

# Marcar el intervalo de convergencia
if len(intervalo) == 2:
    ax.axvspan(intervalo[0], intervalo[1], alpha=0.2, color='green',
               label=f'Intervalo de convergencia [{intervalo[0]},␣
 ↪{intervalo[1]}]')

ax.grid(True, alpha=0.3)
ax.set_xlabel('  (parámetro de relajación)', fontsize=12)
ax.set_ylabel(' (M_SOR) (radio espectral)', fontsize=12)
ax.set_title('Radio espectral del método SOR en función de  ',
             fontsize=14, fontweight='bold')
ax.legend(fontsize=10, loc='best')
ax.set_ylim([0, max(radios[~np.isnan(radios)]) * 1.1])

plt.tight_layout()
plt.show()

print("\nGráfica generada exitosamente  ")

# ------------------------------------------------------------------------
# RESUMEN FINAL
# ------------------------------------------------------------------------

print("\n" + "=" * 70)
print("RESUMEN DE LA VALIDACIÓN")
print("=" * 70)

print(f"""
SISTEMA DE ECUACIONES:
  | 1  2  3 | | x |   |  1 |
```

```
  | 1  3  5 | | y | = |  0 |
  | 1  4  8 | | z |   | -2 |

RESULTADOS DEL ANÁLISIS SOR:
  • Intervalo de convergencia: [{intervalo[0]:.2f}, {intervalo[1]:.2f}]
  • Radio espectral mínimo:     {rho_min:.16f}
  • Parámetro óptimo  :         {w_optimo}

INTERPRETACIÓN:
  • El método SOR converge para    [{intervalo[0]:.2f}, {intervalo[1]:.2f}]
  • El valor óptimo   = {w_optimo} minimiza el radio espectral
  • Con   óptimo, la tasa de convergencia es   = {rho_min:.4f}
  • Cuanto menor sea  , más rápida es la convergencia

  Todos los resultados coinciden con los valores esperados
""")
```

1.13 Referencias y Recursos Adicionales

1.13.1 Bibliografía Recomendada

Libros Fundamentales

1. **Burden, R. L., & Faires, J. D.** (2010). *Numerical Analysis* (9th ed.). Brooks/Cole.
   - Capítulo 3: Interpolación y Aproximación Polinomial
   - Capítulo 4: Diferenciación e Integración Numérica
   - Capítulo 5: Solución Numérica de Ecuaciones Diferenciales
2. **Quarteroni, A., Sacco, R., & Saleri, F.** (2007). *Numerical Mathematics* (2nd ed.). Springer.
   - Sección 4.2: Interpolación de Lagrange y Tchebishev
   - Sección 8: Integración Numérica
   - Sección 9: Métodos de Extrapolación
3. **Atkinson, K. E.** (1989). *An Introduction to Numerical Analysis* (2nd ed.). Wiley.
   - Capítulo 3: Interpolación Polinomial
   - Capítulo 4: Aproximación de Funciones
   - Capítulo 5: Diferenciación e Integración Numérica
4. **Heath, M. T.** (2018). *Scientific Computing: An Introductory Survey* (3rd ed.). SIAM.
   - Capítulo 7: Interpolación
   - Capítulo 8: Integración Numérica
   - Tratamiento moderno con enfoque computacional
5. **Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P.** (2007). *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). Cambridge University Press.
   - Capítulo 3: Interpolación y Extrapolación
   - Capítulo 4: Integración de Funciones
   - Código práctico y explicaciones detalladas

Recursos Especializados

6. **Trefethen, L. N.** (2013). *Approximation Theory and Approximation Practice*. SIAM.
    - Teoría avanzada de aproximación con polinomios de Tchebishev
    - Enfoque en análisis numérico moderno
7. **Stoer, J., & Bulirsch, R.** (2002). *Introduction to Numerical Analysis* (3rd ed.). Springer.
    - Tratamiento riguroso de extrapolación de Richardson
    - Método de Romberg y técnicas avanzadas

1.13.2 Recursos en Línea

Documentación de Bibliotecas

- **NumPy Documentation** - Documentación oficial de NumPy
    - Tutorial de polinomios: `numpy.polynomial`
    - Operaciones vectoriales y matrices
- **Matplotlib Gallery** - Galería de ejemplos de visualización
    - Gráficas científicas y personalizadas
    - Ejemplos de uso avanzado
- **SciPy Interpolation** - Módulo de interpolación de SciPy
    - `scipy.interpolate.lagrange`: Interpolación de Lagrange
    - `scipy.interpolate.CubicSpline`: Splines cúbicas
    - `scipy.integrate.romberg`: Integración de Romberg
- **SymPy Documentation** - Biblioteca de matemáticas simbólicas
    - Cálculo simbólico de derivadas e integrales
    - Verificación analítica de resultados

Tutoriales y Cursos

- **Real Python - NumPy Tutorial** - Tutorial completo de NumPy
- **Scipy Lecture Notes** - Curso completo de Python científico
- **Python Data Science Handbook** - Libro online gratuito

Recursos Académicos

- **MIT OpenCourseWare - Numerical Methods** - Cursos gratuitos del MIT
- **Wolfram MathWorld** - Enciclopedia matemática
    - Artículos sobre interpolación, diferenciación e integración numérica

1.13.3 Conceptos Clave

1.13.4 Aplicaciones Prácticas

Interpolación

- Reconstrucción de señales en procesamiento digital
- Animación y gráficos por computadora (curvas suaves)
- Modelado de trayectorias en física y robótica

Derivación Numérica

- Cálculo de velocidades y aceleraciones a partir de datos de posición
- Análisis de tasas de cambio en experimentos

- Gradientes para optimización y aprendizaje automático

Integración Numérica

- Cálculo de áreas, volúmenes y centroides
- Evaluación de integrales sin solución analítica
- Física computacional (trabajo, energía, flujos)
- Estadística (funciones de distribución)

1.13.5 Herramientas de Software

1.13.6 Notas sobre Implementación

Buenas prácticas: - Validar siempre con casos conocidos antes de usar en problemas reales - Comparar diferentes métodos para verificar consistencia - Analizar la convergencia y estimar errores - Documentar claramente las hipótesis y limitaciones - Usar vectorización de NumPy para mejor rendimiento

Advertencias: - No usar nodos equiespaciados para interpolación de alto grado - Cuidado con la cancelación numérica en derivación con h muy pequeño - Verificar que las funciones sean suficientemente suaves - El método de Romberg puede fallar con funciones oscilatorias - Siempre verificar la convergencia en integración adaptativa

---

1.13.7 Desarrollado con

Métodos Numéricos y Análisis Funcional 2025

Última actualización: Noviembre 2025