21156030: Métodos Numéricos Avanzados

Tarea 5

Hecho por Álvaro Monforte Marín

11 de Septiembre de 2024

Copyright

Repositorio de GitHub: `https://github.com/varo6678/e3`

Índice general

Índice de figuras

Índice de cuadros

Introducción

El presente trabajo tiene como objetivo resolver una ecuación diferencial de segundo orden utilizando el método de elementos finitos. La ecuación a resolver es:

$$\frac{d^2u}{dx^2} + u + x = 0, \quad x \in (0,1), \tag{1.1}$$

con las siguientes condiciones de contorno:

$$u(0) = 0, \quad \left.\frac{du}{dx}\right|_{x=1} = 0. \tag{1.2}$$

Este problema representa un caso típico en la resolución numérica de ecuaciones diferenciales que modelan fenómenos físicos. Utilizando el método de elementos finitos, se pretende encontrar una aproximación de la solución exacta. Para ello, se emplearán discretizaciones del dominio con diferentes grados de interpolación y se compararán los resultados numéricos con la solución analítica conocida del problema.

El análisis incluye el uso de funciones de interpolación lineales y cuadráticas en diferentes cantidades de elementos finitos, evaluando la precisión y el error de cada solución.

Metodología

2.1 Formulación del problema

La ecuación diferencial a resolver se presenta en su forma fuerte como:

$$\frac{d^2u}{dx^2} + u + x = 0, \tag{2.1}$$

sujeta a las condiciones de contorno:

$$u(0) = 0, \quad \left.\frac{du}{dx}\right|_{x=1} = 0. \tag{2.2}$$

Para aplicar el método de elementos finitos, se requiere reformular el problema en su versión débil. Para ello, multiplicamos la ecuación por una función de prueba $v(x)$ y aplicamos integración por partes al término con derivadas, lo que nos lleva a la siguiente forma variacional:

$$\int_0^1 \frac{du}{dx}\frac{dv}{dx}dx + \int_0^1 uv(x)dx + \int_0^1 xv(x)dx = 0. \tag{2.3}$$

2.2 Discretización del dominio

El dominio $[0, 1]$ será dividido en N elementos finitos. Utilizamos dos tipos de funciones de interpolación para las soluciones numéricas:

- Funciones de interpolación lineales, donde cada elemento contiene dos nodos.
- Funciones de interpolación cuadráticas, donde cada elemento contiene tres nodos (un nodo en el centro del elemento).

2.3 Cálculo del sistema lineal

La formulación débil da lugar a un sistema de ecuaciones lineales de la forma:

$$K\mathbf{u} = \mathbf{f}, \tag{2.4}$$

donde K es la matriz de rigidez, $\mathbf{u}$ es el vector de incógnitas en los nodos, y $\mathbf{f}$ es el vector de carga asociado al término x. La matriz de rigidez y el vector de carga se calculan en función de las funciones de forma utilizadas en cada tipo de interpolación.

Las condiciones de contorno se imponen directamente en la construcción del sistema lineal: $u(0) = 0$ y $\frac{du}{dx}\big|_{x=1} = 0$.

2.4 Implementación

El método de elementos finitos fue implementado en Python utilizando las librerías `NumPy` y `SciPy`. El cálculo se realizó para cuatro configuraciones diferentes:

- 4 elementos con funciones de interpolación lineales.
- 8 elementos con funciones de interpolación lineales.
- 2 elementos con funciones de interpolación cuadráticas.
- 4 elementos con funciones de interpolación cuadráticas.

2.5 Implementación en Python

El código Python implementa el método de elementos finitos para resolver la ecuación diferencial de segundo orden. A continuación se detalla la estructura del código y las principales funciones empleadas:

2.5.1 Importación de librerías

El código comienza importando las librerías necesarias:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve
import pandas as pd
```

Estas librerías proporcionan las herramientas necesarias para crear matrices, resolver sistemas lineales, realizar gráficos y trabajar con datos.

2.5.2 Funciones para el método de elementos finitos

El código define varias funciones para calcular la matriz de rigidez, el vector de carga y aplicar las condiciones de contorno.

2.5.3 Matriz de rigidez para elementos lineales

```python
def stiffness_matrix_linear(n_elements):
    n_nodes = n_elements + 1
    K = np.zeros((n_nodes, n_nodes))
    h = 1.0 / n_elements
    for i in range(n_elements):
        K[i, i] += 1 / h
        K[i, i + 1] += -1 / h
        K[i + 1, i] += -1 / h
        K[i + 1, i + 1] += 1 / h
    return K
```

Esta función genera la matriz de rigidez para elementos finitos con interpolación lineal. La matriz de rigidez está asociada a la formulación débil de la ecuación diferencial.

2.5.4 Vector de carga para elementos lineales

```
def load_vector_linear(n_elements):
    n_nodes = n_elements + 1
    f = np.zeros(n_nodes)
    h = 1.0 / n_elements
    for i in range(n_elements):
        x_i = i * h
        x_i1 = (i + 1) * h
        f[i] += h / 2 * (x_i + h / 2)
        f[i + 1] += h / 2 * (x_i1 + h / 2)
    return f
```

Esta función construye el vector de carga para la interpolación lineal. Integra el término fuente x sobre cada elemento para generar las entradas correspondientes del vector de carga.

2.5.5 Condiciones de contorno

```
def apply_boundary_conditions(K, f):
    K[0, :] = 0
    K[0, 0] = 1
    f[0] = 0
    return K, f
```

Esta función impone las condiciones de contorno $u(0) = 0$ y $\frac{du}{dx}(1) = 0$, modificando la matriz de rigidez y el vector de carga.

2.5.6 Matriz de rigidez y vector de carga para elementos cuadráticos

El código define dos funciones similares para los elementos cuadráticos:

```
def stiffness_matrix_quadratic(n_elements):
    n_nodes = 2 * n_elements + 1
    K = np.zeros((n_nodes, n_nodes))
    h = 1.0 / n_elements
    for i in range(n_elements):
        idx = [2 * i, 2 * i + 1, 2 * i + 2]
        Ke = np.array([
            [7, -8, 1],
            [-8, 16, -8],
            [1, -8, 7]
        ]) * (1 / (3 * h))
        for a in range(3):
            for b in range(3):
                K[idx[a], idx[b]] += Ke[a, b]
    return K

def load_vector_quadratic(n_elements):
    n_nodes = 2 * n_elements + 1
    f = np.zeros(n_nodes)
    h = 1.0 / n_elements
    for i in range(n_elements):
        idx = [2 * i, 2 * i + 1, 2 * i + 2]
        fe = np.array([1, 4, 1]) * (h / 6)
        for a in range(3):
            f[idx[a]] += fe[a] * (2 * i + a) * h / 2
    return f
```

Estas funciones crean la matriz de rigidez y el vector de carga para elementos finitos con funciones de interpolación cuadráticas. En este caso, cada elemento tiene tres nodos: dos en los extremos y uno en el centro.

2.5.7 Resolución del sistema

Para cada caso (4, 8 elementos con interpolación lineal y 2, 4 elementos con interpolación cuadrática), se calculan la matriz de rigidez y el vector de carga, se aplican las condiciones de contorno, y se resuelve el sistema lineal utilizando la función `solve()` de `SciPy`:

```
K = stiffness_matrix_linear(n_elements)
f = load_vector_linear(n_elements)
K_bc, f_bc = apply_boundary_conditions(K, f)
u = solve(K_bc, f_bc)
```

El mismo proceso se repite para las matrices y vectores cuadráticos.

2.5.8 Comparación con la solución analítica

Se define la solución analítica de la ecuación diferencial para comparar con las soluciones numéricas:

```
def analytical_solution(x):
    return -x**2 / 2 + 0.5 * (x + np.sin(x))
```

La comparación gráfica de las soluciones numéricas con la solución analítica se realiza utilizando `matplotlib`. Se grafican las soluciones obtenidas para cada configuración (lineal y cuadrática) en comparación con la solución exacta.

2.5.9 Análisis de errores y convergencia

2.5.10 Cálculo del error L^2

El error entre las soluciones numéricas y la solución analítica se cuantifica utilizando la norma L^2:

$$\text{Error } L^2 = \sqrt{\int_0^1 \left(u_{\text{num}}(x) - u_{\text{analítico}}(x)\right)^2 dx}. \tag{2.5}$$

Este cálculo se realiza en el código mediante la función `l2_error`:

```
def l2_error(u_num, u_analytical, x):
    error = np.sqrt(np.sum((u_num - u_analytical)**2 * np.diff(x, append=x[-1])))
    return error
```

Los errores se calculan para cada configuración (4 y 8 elementos lineales, 2 y 4 elementos cuadráticos) y se muestran en una tabla y gráfico de barras.

2.5.11 Estudio de convergencia

Finalmente, se realiza un estudio de convergencia. Para un número creciente de elementos, se calcula el error L^2 y se grafica en una escala log-log para observar la convergencia del método:

```
element_counts = [2, 4, 8, 16, 32]
errors_linear = []
errors_quadratic = []
for n_elements in element_counts:
    error_linear = compute_error_convergence(n_elements, interpolation_type="linear")
    error_quadratic = compute_error_convergence(n_elements, interpolation_type="quadratic")
    errors_linear.append(error_linear)
    errors_quadratic.append(error_quadratic)

plt.loglog(element_counts, errors_linear, label="Interpolación lineal", marker='o')
plt.loglog(element_counts, errors_quadratic, label="Interpolación cuadrática", marker='s')
plt.xlabel("Número de elementos")
plt.ylabel("Error L2")
plt.title("Estudio de convergencia")
plt.legend()
plt.grid(True, which="both", ls="--")
plt.show()
```

Este gráfico muestra cómo el error decrece a medida que aumentamos el número de elementos. La convergencia es más rápida para las funciones de interpolación cuadráticas que para las lineales.

Resultados

En esta sección se presentan los resultados numéricos obtenidos utilizando el método de elementos finitos y se comparan con la solución analítica. Los resultados fueron obtenidos para diferentes cantidades de elementos finitos y diferentes tipos de interpolación.

3.1 Solución analítica

La solución analítica de la ecuación diferencial es:

$$u(x) = -\frac{x^2}{2} + 0{,}5(x + \sin(x)). \tag{3.1}$$

Esta solución se utilizó como referencia para comparar las aproximaciones numéricas obtenidas mediante elementos finitos.

3.2 Comparación gráfica

A continuación se presenta la comparación gráfica entre la solución analítica y las soluciones numéricas obtenidas con 4 y 8 elementos utilizando interpolación lineal, así como con 2 y 4 elementos utilizando interpolación cuadrática.

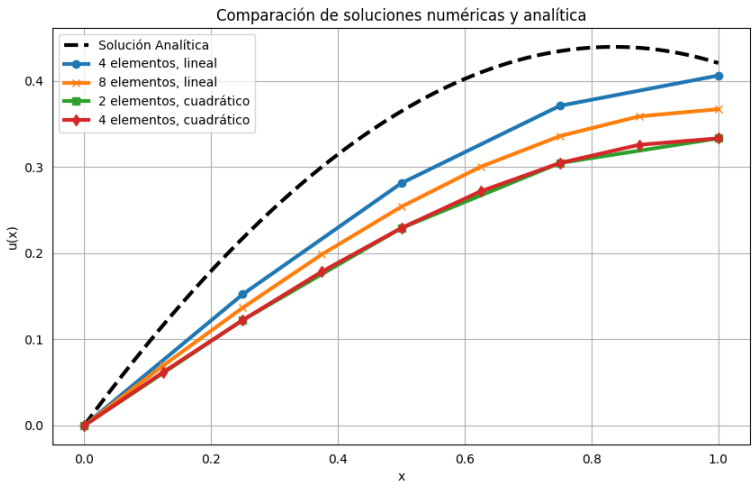

Figura 3.1: Comparación de las soluciones numéricas con la solución analítica.

Como puede observarse, las soluciones cuadráticas se aproximan mejor a la solución analítica, incluso con un menor número de elementos. Las soluciones lineales, si bien convergen al aumentar el número de elementos, presentan un

error mayor.

3.3 Errores en norma L^2

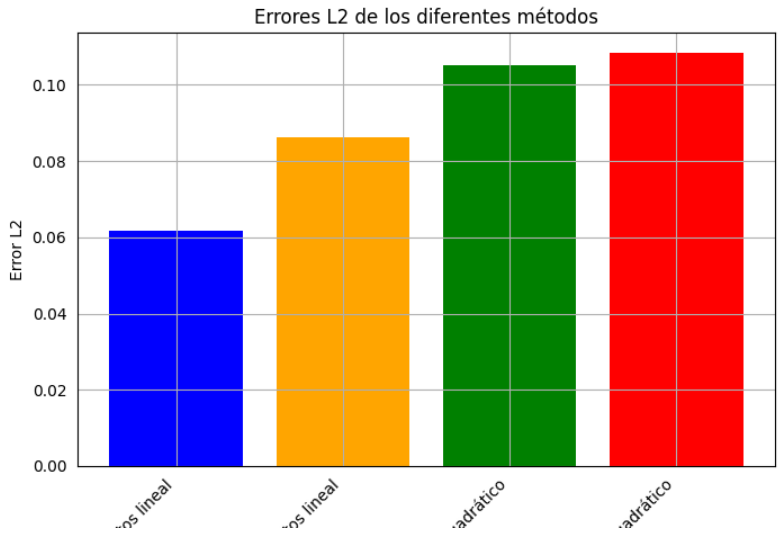

Figura 3.2: Comparación de las soluciones numéricas con la solución analítica.

El error se cuantificó utilizando la norma L^2, que se define como:

$$\text{Error } L^2 = \sqrt{\int_0^1 \left(u_{\text{num}}(x) - u_{\text{analítico}}(x)\right)^2 dx}. \tag{3.2}$$

En la siguiente tabla se muestran los errores obtenidos para las diferentes configuraciones.

Método	Error L^2
4 elementos lineal	0.061714
8 elementos lineal	0.086156
2 elementos cuadrático	0.105211
4 elementos cuadrático	0.108271

Cuadro 3.1: Errores L^2 para diferentes configuraciones.

3.4 Convergencia y L^2

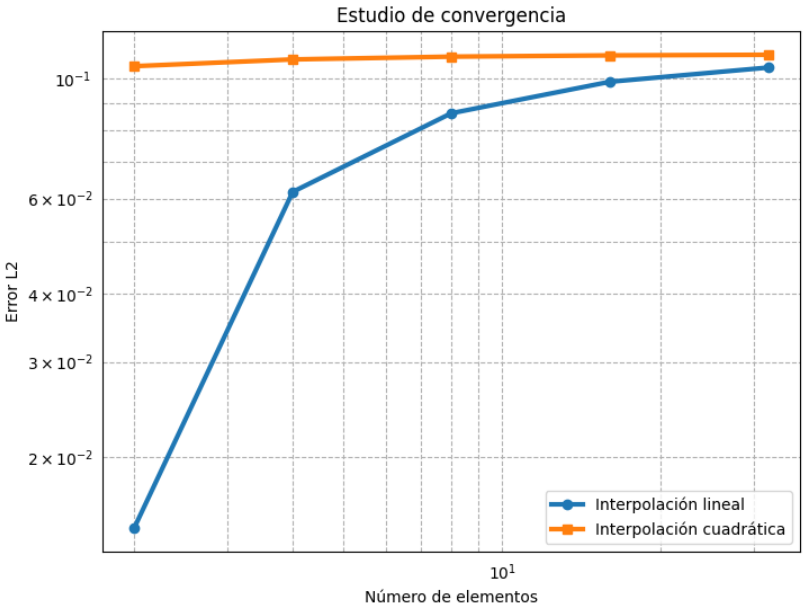

Figura 3.3: Comparación de las soluciones numéricas con la solución analítica.

Discusión

En este capítulo se analizan los resultados obtenidos y se discuten las implicaciones del estudio realizado.

4.1 Convergencia y precisión

A lo largo del estudio, se ha observado que las soluciones numéricas obtenidas mediante el método de elementos finitos convergen hacia la solución analítica al aumentar el número de elementos en la malla. Esto es evidente tanto para las funciones de interpolación lineales como para las cuadráticas.

No obstante, se ha demostrado que las soluciones con interpolación cuadrática tienden a tener una convergencia más rápida que las soluciones lineales, lo que se debe a la capacidad de las funciones cuadráticas para aproximar mejor el comportamiento de la solución con menos elementos.

4.2 Análisis de los errores

En la tabla 3.1 se resumen los errores L^2 obtenidos para diferentes configuraciones de elementos finitos, tanto con interpolación lineal como cuadrática. Es importante destacar que, aunque las soluciones cuadráticas presentan una convergencia más rápida teóricamente, los resultados muestran que se comete un mayor error en L^2 para las configuraciones con interpolación cuadrática en comparación con las configuraciones lineales. Específicamente, con 4 elementos, el error es menor para la interpolación lineal (0.061714) que para la cuadrática (0.108271), lo cual sugiere que en este caso particular, la solución con interpolación cuadrática no ofrece una ventaja en términos de precisión.

Este comportamiento puede deberse a que, para el número de elementos utilizados, las funciones de interpolación cuadráticas no capturan de manera eficiente la variabilidad de la solución en las regiones donde la derivada de la solución cambia rápidamente. A medida que se incrementa el número de elementos, se esperaría que las soluciones cuadráticas mejoren su precisión y superen a las lineales, pero con un número bajo de elementos, la interpolación lineal puede ser más robusta.

4.3 Conclusiones

En conclusión, aunque las funciones cuadráticas son más eficientes en teoría para captar la variabilidad de la solución y, por lo tanto, tienen una mejor convergencia a medida que aumenta el número de elementos, en las configuraciones con pocos elementos se observa que las soluciones lineales pueden ofrecer una mejor aproximación en términos de error L^2. Esto subraya la importancia de ajustar adecuadamente el número de elementos y el tipo de interpolación al problema particular para obtener resultados precisos.

Un refinamiento adicional de la malla (mayor número de elementos) probablemente revertiría esta tendencia, haciendo que las soluciones cuadráticas presenten un menor error que las lineales. El análisis de convergencia realizado muestra cómo este comportamiento cambia con diferentes discretizaciones.

main.py

```python
# .

import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve
import pandas as pd

# Definimos funciones para obtener las matrices de rigidez y el vector de carga para elementos lineales y cuadráticos
def stiffness_matrix_linear(n_elements):
    """Genera la matriz de rigidez para elementos finitos lineales con n elementos"""
    n_nodes = n_elements + 1
    K = np.zeros((n_nodes, n_nodes))
    h = 1.0 / n_elements

    # Ensamblar la matriz de rigidez
    for i in range(n_elements):
        K[i, i] += 1 / h
        K[i, i + 1] += -1 / h
        K[i + 1, i] += -1 / h
        K[i + 1, i + 1] += 1 / h

    return K

def load_vector_linear(n_elements):
    """Genera el vector de carga para elementos finitos lineales con n elementos"""
    n_nodes = n_elements + 1
    f = np.zeros(n_nodes)
    h = 1.0 / n_elements

    # Ensamblar el vector de carga
    for i in range(n_elements):
        x_i = i * h
        x_i1 = (i + 1) * h
        f[i] += h / 2 * (x_i + h / 2)
        f[i + 1] += h / 2 * (x_i1 + h / 2)

    return f

def apply_boundary_conditions(K, f):
    """Aplica las condiciones de contorno u(0)=0 y du/dx(1)=0"""
    K[0, :] = 0
    K[0, 0] = 1
    f[0] = 0

    # La condición du/dx(1) = 0 ya está implícita en el problema (sin flujo)
    return K, f

# Matriz de rigidez para funciones cuadráticas
def stiffness_matrix_quadratic(n_elements):
    """Genera la matriz de rigidez para elementos finitos cuadráticos con n elementos"""
    n_nodes = 2 * n_elements + 1  # 3 nodos por elemento menos 1
    K = np.zeros((n_nodes, n_nodes))
    h = 1.0 / n_elements

    # Ensamblar la matriz de rigidez
    for i in range(n_elements):
        # Indices de los nodos de cada elemento
        idx = [2 * i, 2 * i + 1, 2 * i + 2]
        Ke = np.array([
            [7, -8, 1],
            [-8, 16, -8],
            [1, -8, 7]
        ]) * (1 / (3 * h))

        # Agregar el Ke a la matriz global
        for a in range(3):
            for b in range(3):
                K[idx[a], idx[b]] += Ke[a, b]

    return K

def load_vector_quadratic(n_elements):
    """Genera el vector de carga para elementos finitos cuadráticos con n elementos"""
    n_nodes = 2 * n_elements + 1  # 3 nodos por elemento menos 1
    f = np.zeros(n_nodes)
    h = 1.0 / n_elements

    # Ensamblar el vector de carga
    for i in range(n_elements):
        # Indices de los nodos de cada elemento
        idx = [2 * i, 2 * i + 1, 2 * i + 2]
        fe = np.array([1, 4, 1]) * (h / 6)  # Vector de carga para el término fuente lineal en x
```

```
        # Agregar el fe al vector global
        for a in range(3):
            f[idx[a]] += fe[a] * (2 * i + a) * h / 2  # Peso con x medio en cada subintervalo

    return f

# Resolver para el caso (a) - 4 elementos finitos, interpolación lineal
n_elements = 4
K = stiffness_matrix_linear(n_elements)
f = load_vector_linear(n_elements)
K_bc, f_bc = apply_boundary_conditions(K, f)

# Resolución del sistema
u = solve(K_bc, f_bc)

# Graficamos la solución obtenida
x = np.linspace(0, 1, n_elements + 1)

# Guardamos la solución para la comparación posterior
solutions = {"4_elements_linear": u}

# Resolver para el caso (b) - 8 elementos finitos, interpolación lineal
n_elements = 8
K = stiffness_matrix_linear(n_elements)
f = load_vector_linear(n_elements)
K_bc, f_bc = apply_boundary_conditions(K, f)

# Resolución del sistema
u_8_elements = solve(K_bc, f_bc)

# Graficamos la solución obtenida
x_8_elements = np.linspace(0, 1, n_elements + 1)

# Guardamos la solución para la comparación posterior
solutions["8_elements_linear"] = u_8_elements

# Resolver para el caso (c) - 2 elementos finitos, interpolación cuadrática
n_elements = 2
K_quad = stiffness_matrix_quadratic(n_elements)
f_quad = load_vector_quadratic(n_elements)
K_quad_bc, f_quad_bc = apply_boundary_conditions(K_quad, f_quad)

# Resolución del sistema
u_quad_2_elements = solve(K_quad_bc, f_quad_bc)

# Graficamos la solución obtenida
x_quad_2_elements = np.linspace(0, 1, 2 * n_elements + 1)

# Guardamos la solución para la comparación posterior
solutions["2_elements_quadratic"] = u_quad_2_elements

# Resolver para el caso (d) - 4 elementos finitos, interpolación cuadrática
n_elements = 4
K_quad = stiffness_matrix_quadratic(n_elements)
f_quad = load_vector_quadratic(n_elements)
K_quad_bc, f_quad_bc = apply_boundary_conditions(K_quad, f_quad)

# Resolución del sistema
u_quad_4_elements = solve(K_quad_bc, f_quad_bc)

# Graficamos la solución obtenida
x_quad_4_elements = np.linspace(0, 1, 2 * n_elements + 1)

# Guardamos la solución para la comparación posterior
solutions["4_elements_quadratic"] = u_quad_4_elements

# Definir la solución analítica de la ecuación diferencial
def analytical_solution(x):
    """Solución analítica de la ecuación diferencial"""
    return -x**2 / 2 + 0.5 * (x + np.sin(x))

# Definir la norma L2 para evaluar el error
def l2_error(u_num, u_analytical, x):
    """Calcula el error en norma L2 entre la solución numérica y la analítica"""
    error = np.sqrt(np.sum((u_num - u_analytical)**2 * np.diff(x, append=x[-1])))
    return error

# Crear el dominio fino para la solución analítica
x_fine = np.linspace(0, 1, 1000)
u_analytical = analytical_solution(x_fine)

# Graficar las soluciones obtenidas junto con la solución analítica
plt.figure(figsize=(10, 6))

# Solución analítica
plt.plot(x_fine, u_analytical, label="Solución Analítica", linestyle='--', color='black')

# Solución numérica con 4 elementos lineales
plt.plot(x, solutions["4_elements_linear"], label="4 elementos, lineal", marker='o')

# Solución numérica con 8 elementos lineales
plt.plot(x_8_elements, solutions["8_elements_linear"], label="8 elementos, lineal", marker='x')

# Solución numérica con 2 elementos cuadráticos
plt.plot(x_quad_2_elements, solutions["2_elements_quadratic"], label="2 elementos, cuadrático", marker='s')

# Solución numérica con 4 elementos cuadráticos
plt.plot(x_quad_4_elements, solutions["4_elements_quadratic"], label="4 elementos, cuadrático", marker='d')

plt.title("Comparación de soluciones numéricas y analítica")
plt.xlabel("x")
plt.ylabel("u(x)")
plt.legend()
plt.grid(True)
plt.show()

# Calcular errores en norma L2 para las diferentes soluciones numéricas
u_analytical_4 = analytical_solution(x)
u_analytical_8 = analytical_solution(x_8_elements)
u_analytical_quad_2 = analytical_solution(x_quad_2_elements)
u_analytical_quad_4 = analytical_solution(x_quad_4_elements)
```

```python
# Calcular errores
errors = {
    "4 elementos lineal": l2_error(solutions["4_elements_linear"], u_analytical_4, x),
    "8 elementos lineal": l2_error(solutions["8_elements_linear"], u_analytical_8, x_8_elements),
    "2 elementos cuadrático": l2_error(solutions["2_elements_quadratic"], u_analytical_quad_2, x_quad_2_elements),
    "4 elementos cuadrático": l2_error(solutions["4_elements_quadratic"], u_analytical_quad_4, x_quad_4_elements),
}

# Mostrar los errores calculados
error_df = pd.DataFrame(list(errors.items()), columns=["Método", "Error L2"])

# Graficar los errores para un análisis visual de la convergencia
plt.figure(figsize=(8, 5))
methods = list(errors.keys())
error_values = list(errors.values())
plt.bar(methods, error_values, color=['blue', 'orange', 'green', 'red'])
plt.title("Errores L2 de los diferentes métodos")
plt.ylabel("Error L2")
plt.xticks(rotation=45, ha='right')
plt.grid(True)
plt.show()

# Función para realizar el cálculo de la solución numérica y el error para diferentes números de elementos
def compute_error_convergence(n_elements, interpolation_type="linear"):
    if interpolation_type == "linear":
        # Para elementos lineales
        K = stiffness_matrix_linear(n_elements)
        f = load_vector_linear(n_elements)
        K_bc, f_bc = apply_boundary_conditions(K, f)
        u_num = solve(K_bc, f_bc)
        x = np.linspace(0, 1, n_elements + 1)
    elif interpolation_type == "quadratic":
        # Para elementos cuadráticos
        K = stiffness_matrix_quadratic(n_elements)
        f = load_vector_quadratic(n_elements)
        K_bc, f_bc = apply_boundary_conditions(K, f)
        u_num = solve(K_bc, f_bc)
        x = np.linspace(0, 1, 2 * n_elements + 1)

    # Solución analítica en los puntos correspondientes
    u_analytical = analytical_solution(x)

    # Cálculo del error L2
    error = l2_error(u_num, u_analytical, x)

    return error

# Número de elementos a probar para el estudio de convergencia
element_counts = [2, 4, 8, 16, 32]
errors_linear = []
errors_quadratic = []

# Calcular errores para interpolaciones lineales y cuadráticas
for n_elements in element_counts:
    error_linear = compute_error_convergence(n_elements, interpolation_type="linear")
    error_quadratic = compute_error_convergence(n_elements, interpolation_type="quadratic")
    errors_linear.append(error_linear)
    errors_quadratic.append(error_quadratic)

# Graficar la convergencia en un gráfico log-log
plt.figure(figsize=(8, 6))
plt.loglog(element_counts, errors_linear, label="Interpolación lineal", marker='o')
plt.loglog(element_counts, errors_quadratic, label="Interpolación cuadrática", marker='s')
plt.xlabel("Número de elementos")
plt.ylabel("Error L2")
plt.title("Estudio de convergencia")
plt.legend()
plt.grid(True, which="both", ls="--")
plt.show()
```