21156030: Métodos Numéricos Avanzados

Tarea 4

Hecho por Álvaro Monforte Marín

11 de Septiembre de 2024

Copyright

Índice general

Índice de figuras

Índice de cuadros

Lista de Abreviaturas

CFL Courant-Friedrichs-Lewy. 2, 10

Introducción al problema

En este informe, abordaremos la resolución de una ecuación diferencial hiperbólica con las siguientes condiciones iniciales y de frontera. Consideramos la ecuación:

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} + (1 - x^2) = 0, \tag{1.1}$$

definida para $x \in (0, 1)$ y $\forall t > 0$, con las siguientes condiciones de contorno y condiciones iniciales:

$$u(x, 0) = x(1 - x), \quad \frac{\partial u}{\partial t}(x, 0) = 0, \quad u(0, t) = 0, \quad u(1, t) = 0. \tag{1.2}$$

La ecuación diferencial hiperbólica que nos ocupa tiene importantes aplicaciones en la física, especialmente en la modelación de ondas. En esta actividad, aplicaremos dos métodos numéricos distintos para su resolución:

- El **método de diferencias finitas**, que aproximará la solución mediante discretización en el tiempo y el espacio.
- El **método de las características**, que es un enfoque analítico para resolver ecuaciones de onda, utilizando las curvas características de la ecuación diferencial.

Finalmente, compararemos los resultados obtenidos por ambos métodos en el mismo punto del dominio y realizaremos un análisis cuantitativo del error y de la convergencia de las soluciones numéricas.

Metodología

2.1 Diferencias Finitas

2.2 Esquema de diferencias finitas

El método de diferencias finitas es una técnica numérica que nos permite aproximar derivadas mediante diferencias entre valores discretos de la función. Para la ecuación diferencial dada, discretizamos tanto la coordenada espacial x como el tiempo t.

En este caso, utilizaremos una malla uniforme de puntos (x_i, t_n) con un paso espacial Δx y un paso temporal Δt. La ecuación diferencial se puede aproximar usando diferencias centradas para las segundas derivadas espaciales y temporales:

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2}, \quad \frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}. \tag{2.1}$$

Sustituyendo estas aproximaciones en la ecuación diferencial, obtenemos el esquema de diferencias finitas explícito:

$$u_i^{n+1} = 2u_i^n - u_i^{n-1} + r\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \Delta t^2(1 - x_i^2), \tag{2.2}$$

donde $r = \left(\frac{\Delta t}{\Delta x}\right)^2$ es el coeficiente de estabilidad de Courant-Friedrichs-Lewy (CFL), que asegura la estabilidad del esquema numérico.

Las condiciones iniciales y de frontera se aplican de la siguiente manera:

$$u(0,t) = 0, \quad u(1,t) = 0, \quad u(x,0) = x(1-x), \quad \frac{\partial u}{\partial t}(x,0) = 0. \tag{2.3}$$

Este esquema se implementará en el código (puede verse en la subseccion 2.2.1) para obtener la solución numérica de la ecuación de onda en un dominio discreto.

2.2.1 Implementación en Python para diferencias finitas

A continuación, se muestra la implementación en Python del esquema de diferencias finitas para resolver la ecuación de onda hiperbólica. El código se encarga de discretizar el dominio espacial y temporal, aplicar las condiciones iniciales y de frontera, y resolver la ecuación diferencial mediante un bucle temporal con criterio de convergencia. Dentro de `main.py` (anexo B) nos encontramos 2.2.1.

La implementación presentada utiliza el método de diferencias finitas para resolver la ecuación de onda hiperbólica en una dimensión. El dominio espacial se discretiza en Nx puntos y el dominio temporal en Nt pasos. A continuación, se describen los pasos clave del algoritmo:

- Se definen los parámetros de discretización: $dx = L/Nx$ y $dt = T/Nt$, donde L es la longitud espacial y T el tiempo total de simulación. La relación de estabilidad, $\sigma = (dt/dx)^2$, es calculada y utilizada para garantizar la estabilidad del esquema.
- La matriz u se inicializa para almacenar las soluciones en todos los puntos espaciales y temporales. La condición inicial para $u(x, 0)$ se define como $u(x, 0) = x(1 - x)$.
- Se calculan los valores para el primer paso temporal ($n = 1$) mediante una aproximación basada en diferencias centradas para la segunda derivada espacial, más un término fuente proporcional a $1 - x^2$.
- Se aplican condiciones de frontera de Dirichlet, donde $u(0, t) = 0$ y $u(L, t) = 0$ para todos los instantes de tiempo.
- El bucle principal recorre cada paso temporal. Para cada instante t_n, la solución en el siguiente instante t_{n+1} se calcula utilizando un esquema explícito de diferencias finitas, basado en los valores en t_n y t_{n-1}.
- Después de cada paso temporal, se verifica el criterio de convergencia calculando el cambio máximo $\Delta u = \max |u^{n+1} - u^n|$. Si el cambio máximo es menor que la tolerancia definida, el bucle se detiene indicando que la solución ha convergido.
- Finalmente, se devuelven los valores de x, el tiempo hasta el paso convergente, y la solución u para todos los pasos espaciales y temporales hasta el punto de convergencia.

De esta forma, el código implementa el método de diferencias finitas para resolver la ecuación de onda con condiciones iniciales y de frontera específicas, proporcionando una aproximación numérica de la solución.

```python
def resolver_onda_hiperbolica(Nx: int, Nt: int, L: float, T: float, tolerancia: float = 1e-6):
    # Parámetros de discretización.
    dx = L / Nx
    dt = T / Nt
    x = np.linspace(0, L, Nx + 1)
    t = np.linspace(0, T, Nt + 1)
    sigma = (dt / dx) ** 2

    # Inicializar la matriz de soluciones.
    u = np.zeros((Nt + 1, Nx + 1))

    # Condición inicial u(x, 0) = x(1 - x).
    u[0, :] = x * (1 - x)

    # Condición inicial de la derivada temporal es cero.
    # Calculamos u[1, :] usando la aproximación:
    for i in range(1, Nx):
        u[1, i] = u[0, i] + 0.5 * sigma * (
            u[0, i+1] - 2*u[0, i] + u[0, i-1] + dx**2 * (1 - x[i]**2)
        )

    # Aplicar condiciones de frontera.
    u[:, 0] = 0  # u(0, t) = 0
    u[:, Nx] = 0  # Suponiendo u(L, t) = 0

    # Bucle de tiempo con criterio de convergencia.
    for n in range(1, Nt):
        u_old = u[n, :].copy()  # Copiar la solución anterior.

        # Actualizar solución en el paso n+1
        for i in range(1, Nx):
            u[n+1, i] = (
                2*u[n, i] - u[n-1, i] + sigma * (
                    u[n, i+1] - 2*u[n, i] + u[n, i-1] + dx**2 * (1 - x[i]**2)
                )
            )

        # Aplicar condiciones de frontera.
        u[n+1, 0] = 0
        u[n+1, Nx] = 0

        # Verificar criterio de convergencia.
        max_delta = np.max(np.abs(u[n+1, :] - u_old))
        if max_delta < tolerancia:
            informer.info(f'Convergencia alcanzada en el paso temporal n={n} con delta máximo={max_delta:.2e}')
            break

    return x, t[:n+2], u[:n+2, :]  # Devolver solo hasta el paso convergente
```

Figura 2.1: Implementación en Python del método de diferencias finitas para resolver la ecuación de onda hiperbólica.

2.3 Implementación del método de las características

El método de las características convierte la ecuación diferencial parcial de segundo orden en un conjunto de ecuaciones diferenciales ordinarias que se resuelven a lo largo de las líneas características. Estas líneas son trayectorias en el espacio-temporal en las que las derivadas parciales de la solución se simplifican o se mantienen constantes.

Para la ecuación de onda que estamos resolviendo:

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} + (1 - x^2) = 0, \tag{2.4}$$

las características vienen dadas por las líneas $x - t =$ constante y $x + t =$ constante. Esto implica que, a lo largo de estas líneas, el comportamiento de la solución $u(x, t)$ puede estudiarse de manera simplificada. Específicamente, el método transforma la ecuación original en un conjunto de ecuaciones diferenciales ordinarias que se resuelven mediante integración a lo largo de las características.

En términos prácticos, la aplicación del método de las características consiste en los siguientes pasos:

I. **Reescritura de la ecuación diferencial**: A partir de la ecuación original, se separa en dos partes a lo largo de las curvas características. Esto transforma la ecuación parcial en un conjunto de ecuaciones ordinarias que se pueden resolver con métodos convencionales.

II. **Solución a lo largo de las características**: Se integran estas ecuaciones a lo largo de las líneas $x-t =$ constante y $x + t =$ constante, obteniendo una solución que es válida en esas curvas específicas del espacio-tiempo.

III. **Condiciones iniciales y de frontera**: El método también permite incorporar las condiciones iniciales y de frontera de la ecuación de manera directa, ya que las soluciones a lo largo de las características dependen de estos valores en los puntos de partida de las características.

IV. **Combinación de soluciones**: Una vez obtenidas las soluciones a lo largo de cada característica, se combinan para obtener la solución general en el dominio de interés. Esto se hace utilizando las características de la ecuación y las condiciones impuestas por el problema original.

El método de las características es especialmente útil en este tipo de ecuaciones hiperbólicas, ya que nos permite estudiar cómo las soluciones se propagan a lo largo del tiempo y el espacio, siguiendo trayectorias bien definidas. Además, al resolver a lo largo de estas características, se evita la complejidad de tener que resolver una ecuación en dos variables directamente, reduciendo el problema a uno más manejable.

Finalmente, en la sección de resultados se comparará la solución obtenida por este método con la solución numérica basada en diferencias finitas para validar la efectividad y precisión de ambos métodos.

Resultados y discusión

3.1 Implementación del metodo de las características

3.1.1 Paso para calcular el punto R

- **Curvas características:** Las características de una ecuación de onda hiperbólica son las líneas rectas dadas por:

$$x - t = \text{constante} \quad \text{y} \quad x + t = \text{constante}. \tag{3.1}$$

- **Puntos $P(0{,}2, 0)$ y $Q(0{,}4, 0)$:** Sabemos que las características que pasan por P y Q son:
  - Para $P(0{,}2, 0)$, las curvas características son:

$$x - t = 0{,}2 \quad \text{y} \quad x + t = 0{,}2. \tag{3.2}$$

  - Para $Q(0{,}4, 0)$, las curvas características son:

$$x - t = 0{,}4 \quad \text{y} \quad x + t = 0{,}4. \tag{3.3}$$

- **Intersección en R:** El punto de intersección de estas dos características es el punto R, cuyas coordenadas (x_R, t_R) deben satisfacer:

$$x - t = 0{,}2 \quad \text{y} \quad x + t = 0{,}4. \tag{3.4}$$

- Resolviendo este sistema:

$$x - t = 0{,}2 \tag{3.5}$$

$$x + t = 0{,}4 \tag{3.6}$$

- Sumando 3.5 y 3.6:

$$2x = 0{,}6 \quad \Rightarrow \quad x = 0{,}3 \tag{3.7}$$

- Restando 3.6 de 3.5:

$$2t = 0{,}2 \quad \Rightarrow \quad t = 0{,}1 \tag{3.8}$$

Así, el punto de intersección R tiene las coordenadas $R(0{,}3, 0{,}1)$.

3.2 Método de las características

En esta sección, resolveremos la ecuación diferencial utilizando el método de las características. Consideramos las curvas características que pasan por los puntos $P(0{,}2, 0)$ y $Q(0{,}4, 0)$, las cuales se intersectan en el punto $R(0{,}3, 0{,}1)$.

La ecuación diferencial es la ecuación de onda unidimensional:

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} = 0 \tag{3.9}$$

La solución general de esta ecuación se puede escribir en términos de las características:

$$u(x,t) = f(x-t) + g(x+t) \tag{3.10}$$

Aquí, f y g son funciones que debemos determinar usando las condiciones iniciales.

3.2.1 Características

Las curvas características para esta ecuación son líneas rectas en el plano $x-t$ con pendientes ± 1, es decir:

- $x - t =$ constante
- $x + t =$ constante

Las características que pasan por los puntos $P(0{,}2, 0)$ y $Q(0{,}4, 0)$ tienen las siguientes ecuaciones:

- Para el punto $P(0{,}2, 0)$:

$$x - t = 0{,}2 \quad \text{(es decir, } x = t + 0{,}2\text{)}, \tag{3.11}$$
$$x + t = 0{,}2 \quad \text{(es decir, } x = -t + 0{,}2\text{)}. \tag{3.12}$$

- Para el punto $Q(0{,}4, 0)$:

$$x - t = 0{,}4 \quad \text{(es decir, } x = t + 0{,}4\text{)}, \tag{3.13}$$
$$x + t = 0{,}4 \quad \text{(es decir, } x = -t + 0{,}4\text{)}. \tag{3.14}$$

El punto $R(0{,}3, 0{,}1)$ es el punto de intersección de estas características.

3.2.2 Solución en $R(0{,}3, 0{,}1)$

Sabemos que la solución de la ecuación de onda está dada por:

$$u(x,t) = f(x-t) + g(x+t) \tag{3.15}$$

Donde f y g son funciones que debemos determinar a partir de las condiciones iniciales:

- $u(x, 0) = x(1-x)$
- $\frac{\partial u}{\partial t}(x, 0) = 0$

Primero, usemos la condición $u(x, 0) = x(1-x)$. Esto implica que en $t = 0$:

$$f(x) + g(x) = x(1-x) \tag{3.16}$$

Luego, usando la condición de velocidad inicial $\frac{\partial u}{\partial t}(x, 0) = 0$, tenemos que:

$$f'(x) - g'(x) = 0 \quad \Rightarrow \quad f'(x) = g'(x) \tag{3.17}$$

Esto implica que $f(x)$ y $g(x)$ difieren solo en una constante, es decir:

$$f(x) = g(x) + C \tag{3.18}$$

Sustituyendo en la ecuación $f(x) + g(x) = x(1-x)$, obtenemos:

$$g(x) + C + g(x) = x(1-x) \quad \Rightarrow \quad 2g(x) + C = x(1-x) \tag{3.19}$$

Por lo tanto:

$$g(x) = \frac{x(1-x) - C}{2} \tag{3.20}$$

Y:

$$f(x) = \frac{x(1-x) + C}{2} \tag{3.21}$$

Para determinar C, se necesita información adicional (por ejemplo, condiciones de frontera).

3.2.3 Evaluación de $u(0{,}3, 0{,}1)$

Ahora que tenemos las expresiones para $f(x)$ y $g(x)$, podemos evaluar $u(0{,}3, 0{,}1)$:

$$u(0{,}3, 0{,}1) = f(0{,}3 - 0{,}1) + g(0{,}3 + 0{,}1) = f(0{,}2) + g(0{,}4) \tag{3.22}$$

Sustituimos los valores de $f(x)$ y $g(x)$:

$$f(0{,}2) = \frac{0{,}2(1 - 0{,}2) + C}{2} = \frac{0{,}2 \times 0{,}8 + C}{2} = \frac{0{,}16 + C}{2}, \tag{3.23}$$

$$g(0{,}4) = \frac{0{,}4(1 - 0{,}4) - C}{2} = \frac{0{,}4 \times 0{,}6 - C}{2} = \frac{0{,}24 - C}{2}. \tag{3.24}$$

Finalmente, sumamos los resultados:

$$u(0{,}3, 0{,}1) = \frac{0{,}16 + C}{2} + \frac{0{,}24 - C}{2} = \frac{0{,}16 + 0{,}24}{2} = \frac{0{,}4}{2} = 0{,}2 \tag{3.25}$$

3.3 Metodo de diferencias finitas

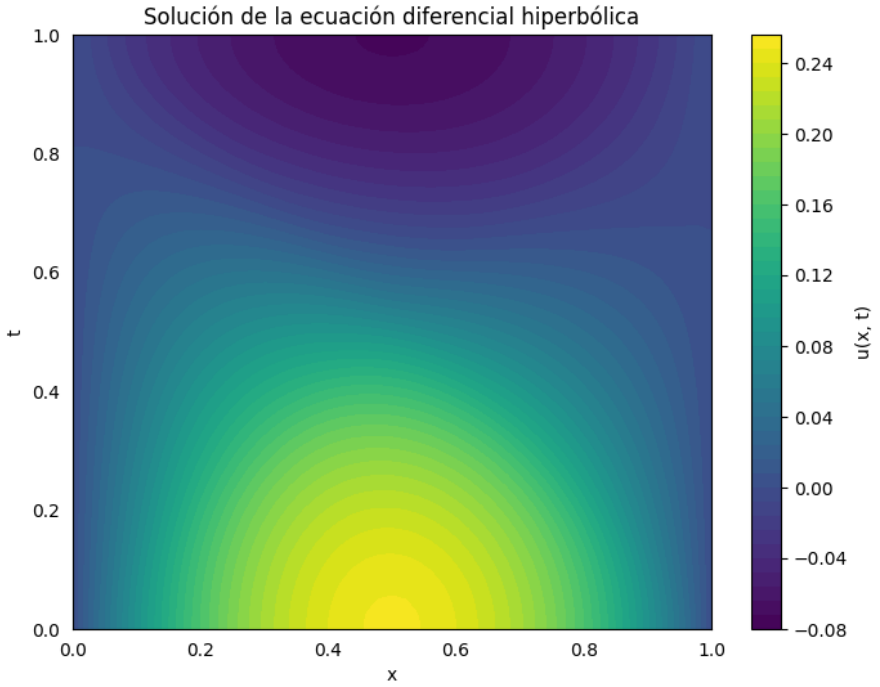

Figura 3.1: Solución numérica de la ecuación de onda en el dominio $x \in (0, 1)$ y $t \in (0, 1)$.

Punto	Valor Analítico	Valor Numérico	Error Absoluto	Error Relativo
P(0.2, 0)	0.160000	0.160000	0.000000	0.000000
Q(0.4, 0)	0.240000	0.240000	0.000000	0.000000
R(0.3, 0.1)	0.200000	0.204542	0.004542	2.270875

Cuadro 3.1: Comparación de los valores analíticos y numéricos en los puntos P, Q y R, con errores absoluto y relativo.

3.4 Comparación de los métodos

La figura 3.1 muestra la solución numérica obtenida a partir del método de diferencias finitas para la ecuación de onda hiperbólica en el dominio $x \in (0,1)$ y $t \in (0,1)$. En este método, el dominio espacial y temporal se discretiza utilizando un número de puntos N_x y N_t respectivamente, y se implementa un esquema explícito que relaciona la evolución temporal de la solución en términos de los valores presentes y pasados.

La distribución que se observa en la figura refleja la evolución de la solución $u(x,t)$, la cual depende de las condiciones iniciales y de frontera impuestas. En particular, en esta simulación se ha supuesto que $u(x,0) = x(1-x)$, lo que proporciona un perfil parabólico inicial. A medida que avanza el tiempo, este perfil evoluciona según las características de la ecuación, y la figura captura cómo la solución varía en función del tiempo y el espacio.

Además, se observa que las condiciones de frontera en $x = 0$ y $x = 1$ fuerzan a la solución a ser cero en estos puntos durante todo el tiempo simulado, lo que genera una simetría en el comportamiento de $u(x,t)$ alrededor de $x = 0{,}5$.

La comparación de los métodos numérico (diferencias finitas) y analítico (método de las características) es esencial para validar la precisión del método numérico. En la Tabla 3.3, se presenta una comparación entre los valores numéricos obtenidos en puntos específicos del dominio y los valores analíticos calculados utilizando el método de las características.

Los puntos seleccionados son $P(0{,}2, 0)$, $Q(0{,}4, 0)$ y $R(0{,}3, 0{,}1)$. A continuación, se detalla la interpretación de estos resultados:

- **Punto** $P(0{,}2, 0)$: El valor numérico coincide exactamente con el valor analítico, lo que sugiere que el método de diferencias finitas proporciona una excelente aproximación en este punto. El error absoluto y el error relativo son ambos cero, lo que es un buen indicativo de la precisión en los puntos iniciales.
- **Punto** $Q(0{,}4, 0)$: Similar al punto anterior, el valor numérico coincide perfectamente con el valor analítico. Esto refuerza la idea de que el método numérico reproduce de manera correcta los valores en el tiempo inicial.
- **Punto** $R(0{,}3, 0{,}1)$: A medida que avanzamos en el tiempo, comienzan a aparecer pequeñas discrepancias entre el valor analítico y el numérico. En este caso, el error absoluto es de 0.004542, lo cual es relativamente pequeño en términos absolutos, pero en términos relativos representa un error del 2.27

En general, la tabla demuestra que el método de diferencias finitas es capaz de reproducir los resultados analíticos con gran precisión en las etapas iniciales y en ciertas regiones del dominio, aunque se observa una ligera pérdida de precisión a medida que se avanza en el tiempo.

3.4.1 Posibles mejoras en el método numérico

Aunque los resultados obtenidos mediante el método de diferencias finitas son muy precisos en gran parte del dominio, hay algunas mejoras que podrían implementarse para aumentar la precisión, especialmente en puntos como $R(0{,}3, 0{,}1)$, donde el error relativo es más significativo:

- **Aumento de la resolución temporal y espacial**: Una posible solución para reducir el error es incrementar el número de puntos espaciales N_x y temporales N_t, lo que llevaría a una mayor precisión en la discretización del dominio. Esto ayudaría a disminuir la acumulación de errores en la evolución temporal.
- **Refinamiento adaptativo de la malla**: En lugar de usar una malla uniforme, se podría emplear un método adaptativo que refine la malla en las regiones donde la solución varía más rápidamente. Esto permitiría una mejor aproximación en las zonas críticas sin aumentar innecesariamente el costo computacional en otras áreas.
- **Métodos de diferencias finitas de mayor orden**: El esquema utilizado es de segundo orden de precisión. Si bien esto es suficiente para muchas aplicaciones, se podría implementar un esquema de mayor orden para reducir aún más los errores numéricos.
- **Estudio de la estabilidad**: Aunque el método utilizado es condicionalmente estable, sería interesante estudiar más a fondo el criterio de estabilidad en función del tamaño de paso temporal y espacial para asegurarse de que no hay inestabilidades en la evolución de la solución.

Al comparar el método de diferencias finitas con el método de las características, es evidente que ambos enfoques tienen fortalezas y limitaciones. A continuación se analiza en detalle:

- **Precisión del método de las características**: Este método es analíticamente exacto dentro de sus limitaciones, ya que transforma la ecuación diferencial parcial en un conjunto de ecuaciones diferenciales ordinarias que se pueden resolver de manera exacta. Sin embargo, en situaciones más complejas donde las características se cruzan o el problema involucra condiciones más complicadas, su implementación puede volverse menos práctica.
- **Flexibilidad del método de diferencias finitas**: Aunque el método de diferencias finitas introduce errores numéricos, es extremadamente flexible y puede manejar geometrías complejas, condiciones de frontera arbitrarias y problemas no lineales. Además, su implementación es más sencilla en problemas multidimensionales y no requiere del conocimiento exacto de las características del sistema.
- **Estabilidad y convergencia**: El método de diferencias finitas está sujeto a restricciones de estabilidad (por ejemplo, la relación de CFL) y es necesario asegurar que el tamaño del paso temporal y espacial cumplen con estas condiciones para evitar inestabilidades. A diferencia del método de las características, que es inherentemente estable, el método numérico necesita ser ajustado cuidadosamente para garantizar convergencia.
- **Costo computacional**: El método de diferencias finitas tiende a ser más costoso computacionalmente que el método de las características, ya que requiere resolver la ecuación en una malla de puntos en el espacio y el tiempo. El método de las características, por otro lado, puede ser más eficiente, especialmente cuando el problema puede resolverse de manera analítica a lo largo de las características.
- **Errores y validación**: La tabla de comparación de errores muestra que el método de diferencias finitas proporciona una buena aproximación en la mayoría de los puntos estudiados, con errores absolutos y relativos muy bajos en general. Sin embargo, se observa una mayor discrepancia en puntos más alejados del tiempo inicial, lo que indica que la precisión numérica se degrada a medida que la simulación avanza.

En resumen, el método de diferencias finitas ofrece una solución numérica robusta y flexible, mientras que el método de las características proporciona una solución analítica exacta en ciertos casos específicos

main.py

```
# .

# Imports | external.
from pathlib import Path

# Imports | internal.
from e.src.lib.logger import (
    define_logger,
    dict_log_level,
    dict_level_log
)
from e.src.lib.placeholders import (
    ParametrosFisicos,
    ParametrosGeometricos,
    ParametrosComputacionales,
    NpuntosDireccion
)
from e import np, plt, pd
from e.src.lib.constants import Rutas
from e.src.lib.parsers import define_parser
from e.src.core._typing import (
    Callable
)

informer = define_logger(logger_name='mna', logger_level='INFO')

def resolver_onda_hiperbolica(Nx: int, Nt: int, L: float, T: float, tolerancia: float = 1e-6):
    # Parámetros de discretización.
    dx = L / Nx
    dt = T / Nt
    x = np.linspace(0, L, Nx + 1)
    t = np.linspace(0, T, Nt + 1)
    sigma = (dt / dx) ** 2

    # Inicializar la matriz de soluciones.
    u = np.zeros((Nt + 1, Nx + 1))

    # Condición inicial u(x, 0) = x(1 - x).
    u[0, :] = x * (1 - x)

    # Condición inicial de la derivada temporal es cero.
    # Calculamos u[1, :] usando la aproximación:
    for i in range(1, Nx):
        u[1, i] = u[0, i] + 0.5 * sigma * (
            u[0, i+1] - 2*u[0, i] + u[0, i-1] + dx**2 * (1 - x[i]**2)
        )

    # Aplicar condiciones de frontera.
    u[:, 0] = 0  # u(0, t) = 0
    u[:, Nx] = 0  # Suponiendo u(L, t) = 0

    # Bucle de tiempo con criterio de convergencia.
    for n in range(1, Nt):
        u_old = u[n, :].copy()  # Copiar la solución anterior.

        # Actualizar solución en el paso n+1
        for i in range(1, Nx):
            u[n+1, i] = (
                2*u[n, i] - u[n-1, i] + sigma * (
                    u[n, i+1] - 2*u[n, i] + u[n, i-1] + dx**2 * (1 - x[i]**2)
                )
            )

        # Aplicar condiciones de frontera.
        u[n+1, 0] = 0
        u[n+1, Nx] = 0

        # Verificar criterio de convergencia.
        max_delta = np.max(np.abs(u[n+1, :] - u_old))
        if max_delta < tolerancia:
            informer.info(f'Convergencia alcanzada en el paso temporal n={n} con delta máximo={max_delta:.2e}')
            break

    return x, t[:n+2], u[:n+2, :]  # Devolver solo hasta el paso convergente


if __name__ == '__main__':

    # Definir los parámetros físicos, geométricos y computacionales.
    parser = define_parser()
    argumentos_parseados = parser.parse_args()
    informer.setLevel(argumentos_parseados.verbosity)
```

```python
# Ruta de Outputs:
TEMATICA: str = 'ecuacion_ondas'
FORMATO_GRAFICAS: str = '.png'
OUTPUTS: str = 'OUTPUTS'
RUTA_OUTPUTS: str = f"{Rutas.RUTA_PAQUETE}/../{OUTPUTS}"
Path(RUTA_OUTPUTS).mkdir(parents=True, exist_ok=True)

gparams = ParametrosGeometricos(T=1.0, L=1.0, Nx=50, Nt=100, X_R=0.3, T_R=0.1)
cparams = ParametrosComputacionales(cfl=0.15)

# 1. Soluciones analíticas en los puntos P, Q y R.
u_analitico_P = 0.2 * (1 - 0.2)  # u(0.2, 0) = 0.16
u_analitico_Q = 0.4 * (1 - 0.4)  # u(0.4, 0) = 0.24
u_analitico_R = 0.2  # Resultado ya obtenido para R(0.3, 0.1)

informer.info(f"Solución analítica en P(0.2, 0): {u_analitico_P}")
informer.info(f"Solución analítica en Q(0.4, 0): {u_analitico_Q}")
informer.info(f"Solución analítica en R(0.3, 0.1): {u_analitico_R}")

# 2. Solución numérica utilizando el esquema de diferencias finitas.
tolerancia = 1e-6
x, t, u = resolver_onda_hiperbolica(gparams.Nx, gparams.Nt, gparams.L, gparams.T, tolerancia=tolerancia)

# 3. Obtener los valores numéricos en los puntos P(0.2, 0), Q(0.4, 0), y R(0.3, 0.1).
x_idx_P = np.argmin(np.abs(x - 0.2))
x_idx_Q = np.argmin(np.abs(x - 0.4))
x_idx_R = np.argmin(np.abs(x - gparams.X_R))

t_idx_0 = 0  # t = 0 para los puntos P y Q.
t_idx_R = np.argmin(np.abs(t - gparams.T_R))  # t = 0.1 para el punto R.

u_numerico_P = u[t_idx_0, x_idx_P]  # Solución numérica en P(0.2, 0).
u_numerico_Q = u[t_idx_0, x_idx_Q]  # Solución numérica en Q(0.4, 0).
u_numerico_R = u[t_idx_R, x_idx_R]  # Solución numérica en R(0.3, 0.1).

informer.info(f"Solución numérica en P(0.2, 0): {u_numerico_P}")
informer.info(f"Solución numérica en Q(0.4, 0): {u_numerico_Q}")
informer.info(f"Solución numérica en R(0.3, 0.1): {u_numerico_R}")

# 4. Comparación cuantitativa en los puntos P, Q y R.
error_absoluto_P = np.abs(u_numerico_P - u_analitico_P)
error_relativo_P = (error_absoluto_P / np.abs(u_analitico_P)) * 100

error_absoluto_Q = np.abs(u_numerico_Q - u_analitico_Q)
error_relativo_Q = (error_absoluto_Q / np.abs(u_analitico_Q)) * 100

error_absoluto_R = np.abs(u_numerico_R - u_analitico_R)
error_relativo_R = (error_absoluto_R / np.abs(u_analitico_R)) * 100

informer.info(f"Error absoluto en P(0.2, 0): {error_absoluto_P}, Error relativo: {error_relativo_P:.2f}%")
informer.info(f"Error absoluto en Q(0.4, 0): {error_absoluto_Q}, Error relativo: {error_relativo_Q:.2f}%")
informer.info(f"Error absoluto en R(0.3, 0.1): {error_absoluto_R}, Error relativo: {error_relativo_R:.2f}%")

# 5. Volcado de resultados en una tabla de pandas.
tabla_errores = pd.DataFrame(
    data=[
        ['P(0.2, 0)', u_analitico_P, u_numerico_P, error_absoluto_P, error_relativo_P],
        ['Q(0.4, 0)', u_analitico_Q, u_numerico_Q, error_absoluto_Q, error_relativo_Q],
        ['R(0.3, 0.1)', u_analitico_R, u_numerico_R, error_absoluto_R, error_relativo_R]
    ],
    columns=['Punto', 'Valor Analítico', 'Valor Numérico', 'Error Absoluto', 'Error Relativo (%)']
)

# Guardar la tabla en un archivo CSV.
tabla_errores.to_csv(f"{RUTA_OUTPUTS}/tabla_errores.csv", index=False)
informer.info(f"Resultados volcados en {RUTA_OUTPUTS}/tabla_errores.csv")

# 6. Función para convertir a formato LaTeX.
def convertir_tabla_a_latex(df: pd.DataFrame, ruta_salida: str):
    latex_code = df.to_latex(index=False)
    with open(ruta_salida, 'w') as f:
        f.write(latex_code)
    informer.info(f"Tabla en formato LaTeX guardada en {ruta_salida}")

# Convertir la tabla a LaTeX.
convertir_tabla_a_latex(tabla_errores, f"{RUTA_OUTPUTS}/{TEMATICA}_tabla_errores.tex")

# 7. Visualización de la solución numérica.
plt.figure(figsize=(8, 6))
X, T_grid = np.meshgrid(x, t)
plt.contourf(X, T_grid, u, levels=50, cmap='viridis')
plt.colorbar(label='u(x, t)')
plt.xlabel('x')
plt.ylabel('t')
plt.title('Solución de la ecuación diferencial hiperbólica')
plt.savefig(f"{RUTA_OUTPUTS}/{TEMATICA}{FORMATO_GRAFICAS}")
```

e

B.1 __init__.py

```python
import numpy as np
from .src.lib.matplotlib_settings import plt
import pandas as pd
```

B.2 src

B.2.1 core

B.2.1.1 __init__.py

B.2.1.2 _abstractas.py

```python
# /e/src/core/_abstractas.py

from abc import ABC, abstractmethod
from .._typing import (
    ListaStringsLike,
    DictParametrosLike
)


class Parametros(ABC):

    """
    # Explicacion
    Esta clase pretende facilitar el uso de guardado de parametros.

    ## Example
    >>> class SDEModelParameters(Parameters):
    >>>     mu = 2
    >>>     sigma = 1
    >>>     X0 = 1
    >>>
    >>> params = SDEModelParameters()
    >>> print(params)  # Salida esperada: "Parameters: SDEModelParameters"
    >>> print(params.nombre) # Salida esperada: "SDEModelParameters"
    >>> print(params.parametros_de_la_clase())  # {'mu': 2, 'sigma': 1, 'X0': 1}
    """

    def __init__(self, **kwargs):
        for key, value in kwargs.items():
            setattr(self, key, value)

    def __repr__(self) -> str:
        base_class_name = self.__class__.__bases__[0].__name__
        return f"{base_class_name}: {self.__class__.__name__}"

    @property
    def nombre(self) -> str:
        return self.__repr__().split(':')[1].strip()

    @classmethod
    def lista_de_funciones_prop_de_una_clase(cls) -> ListaStringsLike:
        return [p for p in dir(cls) if isinstance(getattr(cls, p), property)]

    @classmethod
    def parametros_de_la_clase(cls) -> DictParametrosLike:
        # Obtener las propiedades de la clase
        property_names = cls.lista_de_funciones_prop_de_una_clase()

        # Obtener todos los atributos que no sean métodos ni propiedades internas
```

```
        internal_variables_dict = {k: v for k, v in vars(cls).items() if not k.startswith("__")}

        # Excluir las propiedades y los atributos internos que comienzan con "_"
        store_keys = [] + property_names
        for key in internal_variables_dict.keys():
            if key.startswith('_'):
                store_keys.append(key)
        for key in store_keys:
            if key in internal_variables_dict:
                del internal_variables_dict[key]

        return internal_variables_dict
```

B.2.1.3 _typing.py

```
# /e/src/core/_typing.py

from ... import np

from typing import (
    Dict,
    Any,
    Union,
    Tuple,
    List,
    Optional,
    TypeVar,
    Generic,
    Callable
)

__all__ = [
    'Dict',
    'Any',
    'Union',
    'Tuple',
    'List',
    'Optional',
    'TypeVar',
    'Generic',
    'Callable',
    'HiperparametrosLike',
    'ModuloLike',
    'CoordenadaLike',
    'CoordenadasLike',
    'DictOptionsLIke',
    'InputLike',
    'ResultadosLike',
    'ListaStringsLike',
    'DictParametrosLike',
    'ListaIntLike'
]

type ResultadosLike = Dict[str, Any]
type DictOptionsLIke = Dict[str, Any]
type DictParametrosLike = Dict[str, Any]
type HiperparametrosLike = Dict[str, int | float]
type ModuloLike = float
type CoordenadaLike = float | np.ndarray | int
type CoordenadasLike = Tuple[CoordenadaLike, ...]
type InputLike = Tuple[int | float]
type ListaStringsLike = List[str]
type ListaIntLike = List[int]
```

B.2.2 lib

B.2.2.1 __init__.py

B.2.2.2 constants.py

```
# /e/src/lib/constants.py

import os
import sys
from pathlib import Path
from dataclasses import dataclass

__all__ = [
    'Rutas'
]

@dataclass(frozen=True)
class Rutas:
    RUTA_PAQUETE: str = str(Path(__file__).resolve().parents[2])
```

B.2.2.3 general.py

```python
# /e/lib/clases.py

from ..core._abstractas import Parametros
from ..core._typing import (
    InputsLike
)

class ParametrosProblema(Parametros):

    """
    Ejemplo
    ---
    >>> Temperatura.T0 = 0
    >>> Temperatura.T1 = 50
    >>> NpuntosDireccion.Nx = 100
    >>> NpuntosDireccion.Ny = 100
    >>> SemiCirculoParametros.R = 1

    >>> inputs = {
    >>>     'T0' : Temperatura.T0,
    >>>     'T1' : Temperatura.T1,
    >>>     'Nx' : NpuntosDireccion.Nx,
    >>>     'Ny' : NpuntosDireccion.Ny,
    >>>     'R' : SemiCirculoParametros.R
    >>> }

    >>> params = ParametrosProblema(dict_parametros=inputs)
    >>> params.print_parametros
    """

    def __init__(self, dict_parametros: InputsLike) -> None:
        self.inputs = dict_parametros

    def __repr__(self) -> str:
        return f"ParametrosProblema({list(self.inputs.keys())})"

    @property
    def print_parametros(self):
        for parametro, valor in self.inputs.items():
            print(f"{parametro:3} | {valor:6}")
```

B.2.2.4 logger.py

```python
# /e/src/lib/logger.py

import logging
from logging import _nameToLevel, _levelToName
from e.src.core._typing import (
    DictParametrosLike,
)

__all__ = [
    'dict_log_level',
    'dict_level_log',
    'define_logger'
]

dict_log_level: DictParametrosLike = _nameToLevel
dict_level_log: DictParametrosLike = _levelToName

def define_logger(logger_name: str, logger_level: str = 'DEBUG'):

    # Configurar logger.
    logger = logging.getLogger(logger_name)
    logger.setLevel(dict_log_level[logger_level])

    console_handler = logging.StreamHandler()

    # Añadir un formato básico para los mensajes de log.
    formatter = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
    console_handler.setFormatter(formatter)

    # Añadir el handler al logger.
    logger.addHandler(console_handler)

    return logger
```

B.2.2.5 matplotlib_settings.py

```python
# /e/src/lib/matplotlib_settings.py

import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = (9,6)
plt.rcParams['lines.linewidth'] = 3
plt.rcParams['xtick.bottom'] = False
plt.rcParams['ytick.left'] = False
pal = ["#FBB4AE","#B3CDE3", "#CCEBC5","#CFCCC4"]
```

B.2.2.6 metodos_numericos.py

```python
# /e/src/lib/metodos_numericos.py

from ... import np
from .logger import define_logger
from ..core._typing import Callable

__all__ = [
    'solve_wave_eq',
    'jacobi',
    'gauss_seidel',
    'gauss_seidel_sor'
]

informer = define_logger(logger_name='mna', logger_level='INFO')

def DiferenciasFinitas2D(
    u: np.ndarray,
    Nx: int,
    Ny: int,
    update_rule: Callable[[np.ndarray, int, int], float],  # Función para actualizar u[i,j]
    tol: float = 1e-6,
    max_iter: int = int(1e4)
    ) -> np.ndarray:

    for iteracion in range(max_iter):
        u_old = np.copy(u)

        # Iterar sobre los puntos internos.
        for i in range(1, Nx-1):
            for j in range(1, Ny-1):
                # Llamada a la regla de actualización que depende de la ecuación.
                u[i, j] = update_rule(u_old, i, j)

        # Criterio de convergencia
        error = np.max(np.abs(u - u_old))
        if error < tol:
            print(f"Convergencia alcanzada después de {iteracion} iteraciones.")
            return u, iteracion

    print(f"No se alcanzó la convergencia después de {max_iter} iteraciones.")
    return u, max_iter

def solve_wave_eq(Nx, Nt, L, T, cfl):
    """
    Resuelve la ecuación de onda hiperbólica con el esquema explícito en diferencias finitas.

    Args:
    - Nx: Número de puntos en la dirección espacial (x).
    - Nt: Número de puntos en la dirección temporal (t).
    - L: Longitud del dominio espacial.
    - T: Tiempo total a simular.
    - cfl: Número de Courant (CFL), define la relación entre dt y dx.

    Returns:
    - u: Matriz con las soluciones aproximadas.
    - x: Vector de posiciones espaciales.
    - t: Vector de tiempos.
    """
    # Discretización espacial y temporal
    dx = L / (Nx - 1)
    dt = cfl * dx  # Para mantener la estabilidad, dt <= dx/c
    x = np.linspace(0, L, Nx)
    t = np.linspace(0, T, Nt)

    # Coeficiente de estabilidad CFL
    r = (dt / dx)**2

    # Inicialización de la solución
    u = np.zeros((Nt, Nx))
    informer.debug(u)

    # Condiciones iniciales
    u[0, :] = x * (1 - x)  # u(x, 0) = x(1 - x)

    # Primera iteración: derivada temporal cero
    u[1, :] = u[0, :]  # u_t(x, 0) = 0 implica que u[1, :] = u[0, :]

    # Aplicar condiciones de frontera
    u[:, 0] = 0  # u(0, t) = 0
    u[:, -1] = 0  # u(1, t) = 0

    # Iteraciones en el tiempo
    for n in range(1, Nt-1):
        for i in range(1, Nx-1):
            u[n+1, i] = (2 * u[n, i] - u[n-1, i] +
                         r * (u[n, i+1] - 2 * u[n, i] + u[n, i-1]) +
                         dt**2 * (1 - x[i]**2))

    return u, x, t


def solve_wave_eq(Nx, Nt, L, T, cfl):
    """
    Resuelve la ecuación de onda hiperbólica con el esquema explícito en diferencias finitas.

    Args:
    - Nx: Número de puntos en la dirección espacial (x).
    - Nt: Número de puntos en la dirección temporal (t).
    - L: Longitud del dominio espacial.
    - T: Tiempo total a simular.
    - cfl: Número de Courant (CFL), define la relación entre dt y dx.

    Returns:
    - u: Matriz con las soluciones aproximadas.
    - x: Vector de posiciones espaciales.
    - t: Vector de tiempos.
    """
    # Discretización espacial y temporal
```

```
    dx = L / (Nx - 1)
    dt = cfl * dx  # Para mantener la estabilidad, dt <= dx/c
    x = np.linspace(0, L, Nx)
    t = np.linspace(0, T, Nt)

    # Coeficiente de estabilidad CFL
    r = (dt / dx)**2

    # Inicialización de la solución
    u = np.zeros((Nt, Nx))
    informer.debug(u)

    # Condiciones iniciales
    u[0, :] = x * (1 - x)  # u(x, 0) = x(1 - x)

    # Primera iteración: derivada temporal cero
    u[1, :] = u[0, :]  # u_t(x, 0) = 0 implica que u[1, :] = u[0, :]

    # Aplicar condiciones de frontera
    u[:, 0] = 0  # u(0, t) = 0
    u[:, -1] = 0  # u(1, t) = 0

    # Iteraciones en el tiempo
    for n in range(1, Nt-1):
        for i in range(1, Nx-1):
            u[n+1, i] = (2 * u[n, i] - u[n-1, i] +
                         r * (u[n, i+1] - 2 * u[n, i] + u[n, i-1]) +
                         dt**2 * (1 - x[i]**2))

    return u, x, t

def jacobi(u, Nx, Ny, tol=1e-6, max_iter=10000):

    for iteracion in range(max_iter):
        u_old = np.copy(u)

        # Iterar sobre los puntos internos.
        for i in range(1, Nx-1):
            for j in range(1, Ny-1):
                # Esquema para la ecuacion de Laplace.
                u[i, j] = 0.25 * (u[i+1, j] + u[i-1, j] + u[i, j+1] + u[i, j-1])

        # Criterio de convergencia
        error = np.max(np.abs(u - u_old))
        if error < tol:
            informer.info(f"Convergencia alcanzada después de {iteracion} iteraciones.")
            return u, iteracion

    informer.info(f"No se alcanzó la convergencia después de {max_iter} iteraciones.")
    return u, max_iter


def gauss_seidel(u, Nx, Ny, tol=1e-6, max_iter=10000):
    """
    Método de Gauss-Seidel para resolver el sistema de ecuaciones discretizado
    de la ecuación de Laplace.

    Args:
    - u: Matriz con las condiciones iniciales de temperatura.
    - Nx: Número de puntos en la dirección x.
    - Ny: Número de puntos en la dirección y.
    - tol: Tolerancia para la convergencia.
    - max_iter: Máximo número de iteraciones.

    Returns:
    - u: Matriz con las soluciones aproximadas.
    - iteraciones: Número de iteraciones realizadas.
    """
    for iteracion in range(max_iter):
        max_error = 0.0

        # Iterar sobre los puntos internos
        for i in range(1, Nx-1):
            for j in range(1, Ny-1):
                # Guardar el valor anterior
                u_old = u[i, j]

                # Esquema para la ecuación de Laplace
                u[i, j] = 0.25 * (u[i+1, j] + u[i-1, j] + u[i, j+1] + u[i, j-1])

                # Calcular el error máximo
                max_error = max(max_error, abs(u[i, j] - u_old))


        # Criterio de convergencia
        if max_error < tol:
            informer.info(f"Convergencia alcanzada después de {iteracion} iteraciones.")
            return u, iteracion

    informer.info(f"No se alcanzó la convergencia después de {max_iter} iteraciones.")
    return u, max_iter


def gauss_seidel_matriz(A, b, tol=1e-6, max_iter=10000):
    """
    Método de Gauss-Seidel para resolver un sistema lineal Ax = b.

    Args:
    - A: Matriz de coeficientes.
    - b: Vector de términos independientes.
    - tol: Tolerancia para la convergencia.
    - max_iter: Máximo número de iteraciones.

    Returns:
    - x: Vector solución.
    - iteraciones: Número de iteraciones realizadas.
    """
    n = len(b)
    x = np.zeros_like(b, dtype=np.double)  # Vector inicial de solución

    for iteracion in range(max_iter):
```

```
        x_old = np.copy(x)

        # Iterar sobre cada ecuación del sistema
        for i in range(n):
            sigma = 0
            for j in range(n):
                if i != j:
                    sigma += A[i, j] * x[j]

            # Actualizar la solución usando los valores más recientes
            x[i] = (b[i] - sigma) / A[i, i]

        # Criterio de convergencia
        error = np.linalg.norm(x - x_old, ord=np.inf)
        if error < tol:
            informer.info(f"Convergencia alcanzada después de {iteracion} iteraciones.")
            return x, iteracion

    informer.info(f"No se alcanzó la convergencia después de {max_iter} iteraciones.")
    return x, max_iter


def gauss_seidel_sor(u, Nx, Ny, omega=1.5, tol=1e-6, max_iter=10000):

    for iteracion in range(max_iter):
        max_error = 0.0

        # Iterar sobre los puntos internos
        for i in range(1, Nx-1):
            for j in range(1, Ny-1):
                # Guardar el valor anterior
                u_old = u[i, j]

                # Esquema para la ecuación de Laplace (Gauss-Seidel + Sobrerrelajación)
                u_new = 0.25 * (u[i+1, j] + u[i-1, j] + u[i, j+1] + u[i, j-1])

                # Actualización con Sobrerrelajación
                u[i, j] = (1 - omega) * u_old + omega * u_new

                # Calcular el error máximo
                max_error = max(max_error, abs(u[i, j] - u_old))

        # Criterio de convergencia
        if max_error < tol:
            informer.info(f"Convergencia alcanzada después de {iteracion} iteraciones.")
            return u, iteracion

    informer.info(f"No se alcanzó la convergencia después de {max_iter} iteraciones.")
    return u, max_iter
```

B.2.2.7 parsers.py

```
# /e/src/lib/parsers.py

import argparse
from .logger import (
    dict_log_level,
    dict_level_log
)
from ..core._typing import Any

def define_parser(mensaje_descripcion: str = "Este script procesa datos para MNA.") -> Any:

    parser = argparse.ArgumentParser(description=mensaje_descripcion)

    parser.add_argument(
        "-vsy", "--verbosity",
        type=int,
        choices=[level for level in dict_log_level.values()],
        default='INFO',
        help=f"Nivel de verbosidad {list(dict_log_level.items())}"
    )

    parser.add_argument(
        "-sp", "--show_plots",
        action="store_true",
        help="Muestra los plots del script."
    )

    parser.add_argument(
        "-pl", "--parallel",
        action="store_true",
        help="Hace los calculos (los que procedan) en paralelo."
    )

    return parser
```

B.2.2.8 placeholders.py

```
# /e/src/lib/placeholders.py

from ..core._typing import Optional
from ..core._abstractas import Parametros

__all__ = [
    'ParametrosFisicos',
    'NpuntosDireccion',
    'ParametrosGeometricos',
    'ParametrosComputacionales'
]
```

```python
class ParametrosFisicos(Parametros):
    pass

class NpuntosDireccion(Parametros):
    pass

class ParametrosGeometricos(Parametros):
    pass

class ParametrosComputacionales(Parametros):
    pass
```