# Resolución de ecuaciones diferenciales en GPU
# Solving differential equations in GPU

Trabajo de Fin de Grado
Curso 2024–2025

Autor
**Noelia Barranco Godoy**

Director
**Ana María Carpio Rodríguez**

**Doble grado en Matemáticas e Informática**
**Facultad de Ciencias Matemáticas**
**Universidad Complutense de Madrid**

**16 de junio de 2025**

# Resolución de ecuaciones diferenciales en GPU
# Solving differential equations in GPU

**Trabajo de Fin de Grado en Matemáticas**

**Autor**
**Noelia Barranco Godoy**

**Director**
**Ana María Carpio Rodríguez**

**Convocatoria:** Junio 2025

**Doble grado en Matemáticas e Informática**
**Facultad de Ciencias Matemáticas**
**Universidad Complutense de Madrid**

**16 de junio de 2025**

# Dedicatoria

*A mi pareja, Ian, por ser mi refugio en cada instante de agobio y recordarme, con su presencia y cariño, que ninguna meta es inalcanzable cuando no se afronta sola.*

# Agradecimientos

Quiero agradecer a la directora de mi TFG, la profesora Ana María Carpio Rodríguez, por su dedicación y el seguimiento realizado durante la elaboración de este trabajo.

También dentro del ámbito académico, no puedo no mencionar a los profesores del departamente do Arquitectura de Computadores y Automática de la Facultad de Informática de nuestra universidad, Joaquín Recas y Luis Piñel, que me han concedido acceso a una máquina de la facultad, y me han ayudado con la solución de problemas técnicos cuando ha sido necesario.

# Resumen


## Resolución de ecuaciones diferenciales en GPU

El estudio y la simulación numérica de fenómenos físicos descritos por ecuaciones en derivadas parciales (EDPs) constituye una herramienta fundamental en múltiples disciplinas científicas y de ingeniería. Sin embargo, la resolución eficiente de estos problemas puede volverse computacionalmente costosa cuando se requieren mallas finas o dominios de gran tamaño. En este contexto, la programación en GPU surge como una alternativa poderosa para acelerar algoritmos numéricos gracias a su arquitectura de paralelismo masivo.

En este trabajo se analizan e implementan métodos numéricos para resolver cinco EDPs clásicas —la ecuación del calor (1D y 2D), la ecuación de onda (1D y 2D) y la ecuación de Laplace (2D)— mediante esquemas de diferencias finitas. Tras establecer el marco teórico, se desarrollan implementaciones tanto en CPU como en GPU utilizando Python y PyCUDA.

El objetivo principal es cuantificar el aumento de rendimiento obtenido al ejecutar estos algoritmos en GPU. Para ello, se construye un banco de pruebas y se comparan los tiempos de ejecución de ambos algoritmos. Los resultados muestran mejoras significativas al utilizar GPU, con aceleraciones de hasta dos órdenes de magnitud en algunos casos. Finalmente, se discuten las limitaciones observadas y se proponen posibles líneas de optimización y trabajo futuro.



## Palabras clave

ecuaciones en derivadas parciales, diferencias finitas, GPU, CUDA, PyCUDA, paralelismo masivo, ecuación del calor, ecuación de onda, ecuación de Laplace

# Abstract

## Solving differential equations in GPU

The study and numerical simulation of physical phenomena described by partial differential equations (PDEs) is a fundamental tool in many scientific and engineering disciplines. However, solving these problems efficiently can become computationally expensive when fine meshes or large domains are required. In this context, GPU programming emerges as a powerful alternative to accelerate numerical algorithms, thanks to its massively parallel architecture.

This work analyzes and implements numerical methods to solve five classical PDEs —the heat equation (1D and 2D), the wave equation (1D and 2D), and the Laplace equation (2D)— using finite difference schemes. After establishing the theoretical framework, we develop implementations both in CPU and GPU using Python and PyCUDA.

The main objective is to quantify the performance improvement achieved by executing these algorithms in a GPU. To this end, a benchmark is built and execution times are compared between both approaches. The results show significant gains when using GPU, with speed-ups of up to two orders of magnitude in some cases. Finally, observed limitations are discussed and possible optimization paths and future work are proposed.

## Keywords

partial differential equations, finite differences, GPU, CUDA, PyCUDA, massive parallelism, heat equation, wave equation, Laplace equation,

# Índice

**1. Introducción** **1**
1.1. CPU vs. GPU . . . . . 1
1.2. Objetivos . . . . . 1
1.3. Plan de trabajo . . . . . 2

**2. Fundamentos teóricos y numéricos comunes** **5**
2.1. Notación y discretización del dominio . . . . . 5
2.1.1. Dominios espaciales y temporales . . . . . 5
2.1.2. Mallas y funciones discretas . . . . . 6
2.2. Cocientes incrementales . . . . . 6
2.3. Errores y análisis de convergencia . . . . . 7
2.3.1. Error de truncamiento y consistencia . . . . . 7
2.3.2. Error global y convergencia . . . . . 9

**3. Ecuación del calor** **11**
3.1. Ecuación del calor en dimensión uno . . . . . 11
3.1.1. Existencia y unicidad . . . . . 11
3.1.2. Formulación discreta del problema . . . . . 12
3.1.3. Análisis de convergencia . . . . . 13
3.2. Ecuación del calor en dos dimensiones . . . . . 15
3.2.1. Existencia y unicidad . . . . . 15
3.2.2. Formulación discreta del problema . . . . . 16
3.2.3. Análisis de convergencia . . . . . 17

**4. Ecuación de onda** **19**
4.1. Ecuación de onda en dimensión uno . . . . . 19
4.1.1. Existencia y unicidad . . . . . 20
4.1.2. Formulación discreta del problema . . . . . 21
4.1.3. Análisis de convergencia . . . . . 23
4.2. Ecuación de onda en dos dimensiones . . . . . 23
4.2.1. Existencia y unicidad . . . . . 24
4.2.2. Formulación discreta del problema . . . . . 24

4.2.3. Análisis de convergencia . . . . . . . . . . 25

**5. Ecuación de Laplace** **27**
5.1. Existencia y unicidad . . . . . . . . . . 27
5.2. Formulación discreta del problema . . . . . . . . . . 27
5.2.1. Formulación matricial del problema . . . . . . . . . . 28
5.3. Análisis de convergencia . . . . . . . . . . 29
5.4. Resolución del sistema $A\mathbf{U} = \mathbf{b}$ . . . . . . . . . . 30
5.4.1. Método de Jacobi . . . . . . . . . . 30
5.4.2. Formulación componente a componente . . . . . . . . . . 31

**6. Introducción a la computación en GPU** **33**
6.1. Motivación y paralelismo masivo . . . . . . . . . . 33
6.2. Fundamentos de CUDA . . . . . . . . . . 33
6.2.1. Hilo, bloque y rejilla . . . . . . . . . . 34
6.2.2. Memoria . . . . . . . . . . 34
6.3. Modelo de ejecución y flujos . . . . . . . . . . 35
6.4. Sincronización entre flujos y con el anfitrión . . . . . . . . . . 36
6.4.1. Eventos para coordinar flujos . . . . . . . . . . 36
6.4.2. Sincronización anfitrión–dispositivo . . . . . . . . . . 37
6.5. Programación en Python con PyCUDA . . . . . . . . . . 37
6.5.1. Flujo de trabajo básico . . . . . . . . . . 37
6.6. Ejemplos ilustrativos . . . . . . . . . . 38
6.6.1. Hola mundo . . . . . . . . . . 38
6.6.2. Suma de vectores . . . . . . . . . . 38

**7. Implementación de los algoritmos** **39**
7.1. Flujo general de los algoritmos en CPU . . . . . . . . . . 39
7.2. Flujo general de los algoritmos en GPU . . . . . . . . . . 40
7.3. Validación . . . . . . . . . . 42

**8. Pruebas y resultados** **43**
8.1. Diseño del banco de pruebas . . . . . . . . . . 43
8.2. Resultados . . . . . . . . . . 44
8.3. Análisis de los resultados . . . . . . . . . . 44

**9. Conclusiones y Trabajo Futuro** **49**
9.1. Conclusiones . . . . . . . . . . 49
9.2. Trabajo futuro . . . . . . . . . . 49
9.3. Principales contribuciones . . . . . . . . . . 50

**Bibliografía** **51**

**A. Código** **53**
A.1. Código auxiliar . . . . . . . . . . 53

A.2. Ejemplos introductorios . . . . . . . . . . . . . . . . . . . . . . . . . . 54
A.2.1. Hola Mundo . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
A.2.2. Suma de vectores . . . . . . . . . . . . . . . . . . . . . . . . . . 54

**B. Hardware y software utilizado** **57**
B.1. Hardware . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
B.2. Librerías utilizadas . . . . . . . . . . . . . . . . . . . . . . . . . . 57

**Lista de acrónimos** **59**

# Índice de figuras

3.1. Representación del esquema para la ecuación del calor 1D . . . . . . . 14
3.2. Representación del esquema para la ecuación del calor 2D . . . . . . . 16

4.1. Representación del esquema para la ecuación de onda 1D . . . . . . . 23
4.2. Representación del esquema para la ecuación de onda 2D . . . . . . . 25

6.1. Rejilla de bloques de hilos . . . . . . . . . . . . . . . . . . . . . . . . 35

8.1. Tiempos de ejecución . . . . . . . . . . . . . . . . . . . . . . . . . . 47
8.2. Aceleración . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

# Índice de tablas

8.1. Parámetros de cada prueba . . . . . . . . . . . . . . . . . . . . . . . . . 45
8.2. Resultados de cada prueba . . . . . . . . . . . . . . . . . . . . . . . . . . 46

Capítulo 1

# Introducción

Los métodos numéricos constituyen una herramienta esencial cuando las ecuaciones diferenciales que describen un fenómeno físico —por ejemplo la difusión del calor o la propagación de ondas— carecen de solución analítica. En este trabajo estudiamos cómo la resolución de tres modelos clásicos, la ecuación del calor, la ecuación de onda y la ecuación de Laplace, puede acelerarse utilizando programación en GPU (*Graphics Proccessing Unit*, Unidad de Procesamiento Gráfico).

La idea de fondo es sencilla: los esquemas basados en el método de las diferencias finitas realizan la misma operación elemental sobre todos los nodos de una malla. Mientras que una CPU (*Central Proccessing Unit*, Unidad Central de Procesamiento) dispone de unos pocos núcleos versátiles, una GPU moderna agrupa miles de núcleos livianos que permiten explotar este paralelismo masivo.

## 1.1. CPU vs. GPU

La mayoría de los programas se ejecutan en la CPU, cuyo paralelismo efectivo está limitado a unas pocas decenas de núcleos versátiles en el mejor de los casos. Una GPU, en cambio, integra miles de núcleos ligeros que pueden aplicar la misma operación sobre grandes conjuntos de datos de manera simultánea.

Algunos algoritmos (como los que trataremos en este trabajo) pueden explotar esa propiedad de las GPUs para acelerar órdenes de magnitud su tiempo de procesamiento. En este trabajo, implementaremos esos algoritmos con métodos clásicos (en CPU) y con métodos que aprovechan el paralelismo de las GPUs y compararemos su eficiencia (para más detalles sobre la implementación de algoritmos en GPU, véase el Capítulo 6). Las especificaciones concretas del hardware utilizado en estas pruebas puede encontrarse en el Apéndice B.

## 1.2. Objetivos

El propósito principal de este trabajo es **cuantificar la mejora de rendimiento** que se obtiene al resolver una serie de ecuaciones en derivadas parciales mediante programación en GPU frente a una implementación del mismo algoritmo en CPU.

En concreto, las ecuaciones en derivadas parciales a resolver son las siguientes:

Calor (1D):

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}$$

Calor (2D):

$$\frac{\partial u}{\partial t} = \kappa \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

Ondas (1D):

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2}$$

Ondas (2D):

$$\frac{\partial^2 u}{\partial t^2} = v^2 \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

Laplace:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

donde $\kappa > 0$ se conoce como el coeficiente de difusividad térmica, $v > 0$ es la velocidad de propagación de la onda en el medio y la función $u$ denota la incógnita a calcular dadas las condiciones consideradas, es decir, de contorno, iniciales o ambas.

## 1.3. Plan de trabajo

Para alcanzar el objetivo general descrito en la sección anterior se seguirá la secuencia que se detalla a continuación, indicando al mismo tiempo el capítulo en el que se desarrolla cada fase:

- **Capítulo 2 - Fundamentos teóricos y numéricos comunes**: Establecer un marco teórico y notacional común para el desarrollo teórico del trabajo.
- **Capítulo 3 - Ecuación del calor**: Formular el problema de la ecuación del calor en una y dos dimensiones, demostrar existencia y unicidad, construir el esquema explícito correspondiente y demostrar su convergencia.
- **Capítulo 4 - Ecuación de onda**: Formular el problema de la ecuación de onda en una y dos dimensiones, demostrar existencia y unicidad, construir el esquema explícito correspondiente y demostrar su convergencia.
- **Capítulo 5 - Ecuación de Laplace**: Formular el problema de la ecuación de Laplace en dos dimensiones, demostrar existencia y unicidad, construir el esquema explícito correspondiente y demostrar su convergencia.
- **Capítulo 6 - Introducción a la computación en GPU**: Presentar el modelo de programación de CUDA, el flujo de trabajo con PyCUDA y los conceptos clave de estos.
- **Capítulo 7 - Implementación de los algoritmos**: Implementar los algoritmos en CPU y GPU.
- **Capítulo 8 - Pruebas y resultados**: Crear un banco de pruebas para hacer una serie de pruebas y medir los tiempos de ejecución y la mejora introducida por la GPU. Además, se analizarán los resultados obtenidos.
- **Capítulo 9 - Conclusiones y Trabajo Futuro**: Señalar las conclusiones derivadas de las pruebas, valorando en qué medida se han cumplido los objetivos planteados. Además, se señalan las principales contribuciones de este trabajo y se proponen líneas de investigación futuras.

Por último, se incluye un apartado de bibliografía, donde se recogen todas las fuentes consultadas a lo largo del documento, así como un anexo con parte del código utilizado[1] y otro con el hardware y las librerías de Python utilizadas para la realización de pruebas.

[1] Todo el código utilizado para este trabajo puede encontrase también en este repositorio de Github: `https://github.com/NotNoe/TFG-Mates`.

Capítulo 2

# Fundamentos teóricos y numéricos comunes

**Resumen:** Este capítulo reúne los fundamentos teóricos y notacionales que se utilizarán de forma común en el análisis y resolución numérica de las distintas ecuaciones tratadas a lo largo del trabajo. Se describen los esquemas de diferencias finitas utilizados, se formalizan conceptos recurrentes como el dominio y la malla, y se presenta una estrategia común para analizar la convergencia de los métodos empleados.

## 2.1. Notación y discretización del dominio

### 2.1.1. Dominios espaciales y temporales

Con el fin de simplificar la notación en algunas de las demostraciones que haremos más adelante, conviene definir el **dominio espacial** de un problema, que denotaremos por $\Omega$:

- **Ecuaciones con una dimensión espacial** tendrán como dominio espacial un intervalo $\Omega := (a, b) \subset \mathbb{R}$.

- **Ecuaciones con dos dimensiones espaciales** tendrán como dominio espacial un rectángulo abierto $\Omega := (a, b) \times (c, d) \subset \mathbb{R}^2$.

Por tanto, el **dominio** del problema (al que denotaremos como $D$) será:

- $\overline{\Omega} \times [0, T]$ para los problemas evolutivos (ecuación del calor y de onda).

- $\overline{\Omega}$ para los problemas estacionarios (ecuación de Laplace).

### 2.1.2. Mallas y funciones discretas

La resolución numérica de las ecuaciones planteadas se realizará sobre mallas definidas en dominios de dos o tres dimensiones. Se utilizará la siguiente notación para representar el número total de nodos[1]:

- $n_x$: número total de nodos en la dirección $x$,
- $n_y$: número total de nodos en la dirección $y$ (si aplica),
- $n_t$: número total de nodos temporales (si aplica).

El tamaño de paso asociado a cada dirección será: $\Delta x = \frac{b-a}{n_x-1}$, $\Delta y = \frac{d-c}{n_y-1}$, $\Delta t = \frac{T}{n_t-1}$, y los nodos en cada dimensión serán:

- **Nodos espaciales de la dimensión x**: $x_i = a + i\Delta x,\ \ i = 0, \ldots n_x - 1$.
- **Nodos espaciales de la dimensión y**: $y_j = c + j\Delta y,\ \ j = 0, \ldots n_y - 1$.
- **Nodos temporales**: $t_n = n\Delta t$, $n = 0, \ldots n_t - 1$.

Es importante destacar que, dada la definición que hemos dado de los nodos, se cumple la igualdad $x_{i+k} = x_i + k\Delta x$ (para cualquier dimensión), siempre que los índices resultantes estén dentro de la malla.

A lo largo del trabajo, adoptaremos también la siguiente convención: sea $f$ una función cualquiera definida sobre un dominio que contiene a la malla, denotaremos (dependiendo de las dimensiones sobre las que esté definida la función):

- **Variables x, t**: $f_i^n := f(x_i, t_n)$.
- **Variables x, y, t**: $f_{i,j}^n := f(x_i, y_j, t_n)$.
- **Variables x, y**: $f_{i,j} := f(x_i, y_j)$.

También definimos la aproximación de la solución como $U$, entendiéndola como una función definida sobre la malla. Al ser una función definida sobre la malla, también podemos aplicarle la notación que acabamos de definir, permitiéndonos escribir $U_i^n$ como "la aproximación de la solución en el punto $(x_i, t_n)$".

## 2.2. Cocientes incrementales

Sea $f$ una función cualquiera definida sobre la malla (supongamos, sin pérdida de la generalidad, que está definida sobre la dimensión x, con paso $\Delta x$), y $x_i$ un nodo interior en la malla (es decir, $0 < i < n_x - 1$), definimos, siguiendo [6, p. 445], los siguientes operadores de diferencias finitas (utilizando la notación de índices establecida en la sección anterior):

[1]En algunos textos, como los de análisis numérico clásico, se prefiere utilizar $n$ como número de subintervalos, de modo que haya $n+1$ nodos. Aquí optamos por contar directamente el número de nodos, como es habitual en entornos de programación y computación numérica.

**Definicion 2.2.1** (Cociente incremental progresivo).

$$D_+^x f(x_i) := \frac{f_{i+1} - f_i}{\Delta x}.$$

**Definicion 2.2.2** (Cociente incremental regresivo).

$$D_-^x f(x_i) := \frac{f_{i-1} - f_i}{\Delta x}.$$

**Definicion 2.2.3** (Segundo cociente incremental).

$$D_{xx} f(x_i) := \frac{f_{i+1} - 2f_i + f_{i-1}}{\Delta x^2}.$$

Aunque hemos definido estos para funciones de una sola variable, la extensión a funciones de varias variables es inmediata.

## 2.3. Errores y análisis de convergencia

A lo largo de esta sección definiremos formalmente algunos conceptos y resultados que nos serán de utilidad más adelante.

### 2.3.1. Error de truncamiento y consistencia

Definimos a continuación el error de truncamiento de un esquema numérico.

**Definicion 2.3.1** (Error de truncamiento). *Dada una ecuación diferencial $\mathcal{L}[u] = f$ definida sobre un dominio $D \subseteq \mathbb{R}^d$ (según definimos en la Subsección 2.1.1) y una discretización $\mathcal{L}_h[u_h] = f$, se define el error de truncamiento en un punto $x$ de la malla como el error con el que la solución exacta satisface la aproximación, es decir:*

$$\tau(x) := \mathcal{L}_h[u](x) - f(x) = \mathcal{L}_h[u](x) - \mathcal{L}[u](x).$$

*En los casos tratados en este trabajo, siempre se cumple que $f \equiv 0$, por lo que la definición puede simplificarse a*

$$\tau(x) := L_h[u](x).$$

Este valor representa el error cometido al discretizar una ecuación diferencial. Si este error tiende a cero cuando los pasos de la malla tienden a 0, decimos que el esquema es **consistente**. Si además se verifica que:

$$|\tau(x)| = O\left(\sum_i h_i^{p_i}\right),$$

donde $h_i$ es el tamaño de paso de la dimensión $i$ y la notación $\tau(x) = O(f(x))$ indica que la función de error $\tau(x)$ está acotada asintóticamente por una constante multiplicada por $f(x)$ cuando los pasos en las mallas tienden a cero[2], diremos que el esquema es consistente con orden $p_i$ respecto a la variable $i$.

[2]Para más información sobre la notación $O$, véase [5].

**Lemas sobre el error de los cocientes incrementales.**

En nuestro caso concreto el error de truncamiento será introducido por los cocientes incrementales por los que aproximemos las ecuaciones concretas, por lo que necesitamos conocer el error de truncamiento que introduce cada uno de los cocientes incrementales. Para ello, introducimos tres lemas (basados en las demostraciones que aparecen en [6]). Solo los enunciaremos y demostraremos para $D = [a, b] \times [0, T]$, pero la demostración para los otros casos es completamente análoga.

**Lema 2.3.1.** *Supongamos que $f \in C^2(D)$. El error de truncamiento al aproximar la derivada de $f$ por el cociente incremental progresivo tiene orden $\Delta x$. Más concretamente*

$$\tau(\Delta x) := \left| D_+^x f(x,t) - \frac{\partial f}{\partial x}(x,t) \right| = O(\Delta x).$$

*Demostración.* Sea $t \in [0, T]$ fijo, si consideramos la función $f$ únicamente como una función sobre la variable $x$, podemos hacer su desarrollo de Taylor de primer orden centrado en el punto $x$, lo que nos permite obtener

$$f(x + \Delta x, t) = f(x,t) + \Delta x \frac{\partial f}{\partial x}(x,t) + \frac{\Delta x^2}{2} \frac{\partial^2 f}{\partial x^2}(\xi, t)$$

para $\xi \in (x, x + \Delta x)$. Despejando nos lleva a

$$\frac{f(x+\Delta x, t) - f(x,t)}{\Delta x} - \frac{\partial f}{\partial x}(x,t) = \frac{\Delta x}{2} \frac{\partial^2 f}{\partial x^2}(\xi, t) \Rightarrow D_+^x f(x,t) - \frac{\partial f}{\partial x}(x,t) = \frac{\Delta x}{2} \frac{\partial^2 f}{\partial x^2}(\xi, t).$$

Por lo que, ya que $\frac{\partial^2 f}{\partial x^2}$ es continua (y, por tanto, acotada) en el dominio,

$$\tau(\Delta x) \leq \frac{M}{2} \Delta x = O(\Delta x) \Rightarrow \tau(\Delta x) = O(\Delta x).$$

□

**Lema 2.3.2.** *Supongamos que $f \in C^2(D)$. El error de truncamiento al aproximar la derivada de $f$ por el cociente incremental regresivo tiene orden $\Delta x$. Más concretamente*

$$\tau(\Delta x) := \left| D_-^x f(x,t) - \frac{\partial f}{\partial x}(x,t) \right| = O(\Delta x).$$

*Demostración.* Si hacemos el mismo desarrollo de Taylor que en la demostración del lema anterior, pero lo evaluamos en el punto $x - \Delta x$ en lugar de $x + \Delta x$, la demostración es completamente análoga. □

**Lema 2.3.3.** *Supongamos que $f \in C^4(D)$. El error de truncamiento al aproximar la derivada segunda de $f$ por el segundo cociente incremental tiene orden $\Delta x^2$. Más concretamente*

$$\tau(\Delta x) := \left| D_{xx} f(x,t) - \frac{\partial^2 f}{\partial x^2}(x,t) \right| = O(\Delta x^2).$$

*Demostración.* En los desarrollos de Taylor que siguen, todas las derivadas que aparezcan sin evaluar se entienden evaluadas en el punto $(x, t)$.

Si hacemos el desarrollo de Taylor de orden 3 centrado sobre $x$ para los puntos $x + \Delta x$ y $x - \Delta x$ obtenemos

$$f(x + \Delta x, t) = f(x, t) + \Delta x \frac{\partial f}{\partial x} + \Delta x^2 \frac{1}{2} \frac{\partial^2 f}{\partial x^2} + \Delta x^3 \frac{1}{6} \frac{\partial^3 f}{\partial x^3} + \frac{\Delta x^4}{4!} \frac{\partial^4 f}{\partial x^4}(\xi, t)$$

y

$$f(x - \Delta x, t) = f(x, t) - \Delta x \frac{\partial f}{\partial x} + \Delta x^2 \frac{1}{2} \frac{\partial^2 f}{\partial x^2} - \Delta x^3 \frac{1}{6} \frac{\partial^3 f}{\partial x^3} + \frac{\Delta x^4}{4!} \frac{\partial^4 f}{\partial x^4}(\eta, t)$$

respectivamente, siendo $\xi \in (x, x + \Delta x)$ y $\eta \in (x - \Delta x, x)$. Si sumamos ambas ecuaciones, obtenemos

$$f(x + \Delta x, t) + f(x - \Delta x, t) = 2f(x, t) + \Delta x^2 \frac{\partial^2 f}{\partial x^2} + \frac{\Delta x^4}{4!} \left( \frac{\partial^4 f}{\partial x^4}(\xi, t) + \frac{\partial^4 f}{\partial x^4}(\eta, t) \right),$$

que, al despejar y aplicar la definición de segundo cociente incremental, se nos queda en

$$D_{xx} f(x, t) - \frac{\partial^2 f}{\partial x^2} = \frac{\Delta x^2}{4!} \left( \frac{\partial^4 f}{\partial x^4}(\xi, t) + \frac{\partial^4 f}{\partial x^4}(\eta, t) \right).$$

Al ser $\frac{\partial^4 f}{\partial x^4}$ continua en el dominio, es acotada, por lo que podemos obtener la desigualdad

$$\tau(\Delta x) \leq \frac{M}{4!} \Delta x^2 = O(\Delta x^2) \Rightarrow \tau(\Delta x) = O(\Delta x^2).$$

□

### 2.3.2. Error global y convergencia

**Definicion 2.3.2** (Error global). *Dado un problema con solución exacta $u$ y solución aproximada $U$, definimos el* ***error global*** *de la aproximación en un punto de la malla $x$ como:*

$$e(x) := U(x) - u(x).$$

*Para el caso de problemas evolutivos, definimos también el error global en un instante $t_n$ como:*

$$E^n := \max_{x \in \text{malla espacial}} |e(x, t_n)|.$$

**Definicion 2.3.3** (Convergencia). *Decimos que un esquema numérico es convergente si el error global de la aproximación tiende a 0. Además, si se cumple:*

$$|e(x)| = O\left( \sum_i h_i^{p_i} \right),$$

*donde $h_i$ es el tamaño de paso de la dimensión $i$, diremos que el esquema es consistente con orden $p_i$ respecto a la variable $i$.*

Capítulo 3

# Ecuación del calor

**Resumen:** En este capítulo presentamos la ecuación del calor en una y dos dimensiones espaciales. En cada caso formulamos primero el problema de valores iniciales y de contorno, y damos condiciones de existencia y unicidad de la solución. Tras ello, introduciremos discretizaciones en diferencias finitas y detallaremos los algoritmos resultantes.

## 3.1. Ecuación del calor en dimensión uno

En el caso unidimensional, la ecuación del calor es de la forma[1]

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}.$$

Consideramos el siguiente problema mixto, en el que fijaremos las condiciones iniciales y de contorno de tipo Dirichlet (es decir, fijamos el valor en el contorno del dominio espacial y en el instante inicial).

$$\begin{cases} \frac{\partial u}{\partial t}(x,t) = \frac{\partial^2 u}{\partial x^2}(x,t), & (x,t) \in D^\circ, \\ u(x,0) = f(x), & x \in \overline{\Omega}, \\ u(a,t) = \alpha(t), & t \in (0,T], \\ u(b,t) = \beta(t), & t \in (0,T]. \end{cases} \tag{3.1}$$

### 3.1.1. Existencia y unicidad

Antes de demostrar la existencia y unicidad del Problema (3.1), necesitamos introducir algunas definiciones previas.

[1] En realidad, la ecuación del calor incluye un coeficiente (llamado conductividad térmica) que multiplica al término de la derecha, pero se suele omitir porque el cambio de variable $s = kt$ hace desaparecer este coeficiente cuando es constante.

**Definicion 3.1.1** (Frontera parabólica)**.** *Se denomina **frontera parabólica** al conjunto*

$$B := \partial D \setminus \{(x,T) \mid x \in \Omega\},$$

**Definicion 3.1.2** (Función continua a trozos)**.** *Decimos que una función es **continua a trozos** si es continua salvo en un número finito de puntos, en los cuales existen los límites laterales.*

A continuación presentamos dos resultados clásicos sobre la unicidad de soluciones de la ecuación del calor.

**Teorema 3.1.1** (Unicidad, [4, Th. 1.6.4])**.** *Sean $u$ y $v$ funciones continuas en $D$ que satisfacen el Problema* (3.1)*. Si $u = v$ en la frontera parabólica $B$, entonces $u = v$ en todo $D$.*

**Teorema 3.1.2** (Unicidad extendida, [4, Th. 1.6.6])**.** *Sean $u$ y $v$ funciones continuas a trozos en $D$, con un número finito de discontinuidades acotadas, que satisfacen el Problema* (3.1)*. Si $u = v$ en $B$, excepto posiblemente en los puntos de discontinuidad, entonces $u = v$ en todo $D$.*

Finalmente, enunciamos el resultado principal del teorema, basado en una representación integral explícita de la solución.

**Teorema 3.1.3** (Existencia y unicidad, [4, Secs. 6.1-6.2])**.** *Sean $f : \overline{\Omega} \to \mathbb{R}$ y $\alpha, \beta : [0,T] \to \mathbb{R}$ funciones continuas a trozos compatibles (es decir, que $f(a) = \alpha(0)$ y $f(b) = \beta(0)$). Entonces la función*

$$u(x,t) = \int_a^b \left[\theta(x-\xi,t) - \theta(x+\xi,t)\right] f(\xi) d\xi \\ - 2\int_0^t \frac{\partial \theta}{\partial x}(x,t-\tau)\alpha(\tau)d\tau + 2\int_0^t \frac{\partial \theta}{\partial x}(x-1,t-\tau)\beta(\tau)\, d\tau,$$

*donde*

$$\theta(x,t) = \sum_{m=-\infty}^{\infty} K(x+2m,t) \quad t>0, \qquad K(x,t) = \frac{e^{\frac{-x^2}{4t}}}{\sqrt{4\pi t}} \quad t>0,$$

*es la única solución acotada del Problema* (3.1)*.*

*Demostración.* La expresión anterior satisface formalmente la ecuación del calor y las condiciones de contorno en $D$, como se demuestra en [4, Sec. 6.1]. La unicidad se deduce directamente de los teoremas anteriores, ya que la solución obtenida verifica las condiciones requeridas en la frontera parabólica. □

### 3.1.2. Formulación discreta del problema

Procedemos ahora a construir una aproximación numérica de la solución del Problema (3.1) utilizando un esquema en diferencias finitas.

Para ello, discretizamos el dominio $D$ según la notación introducida en la Subsección 2.1.2, utilizando una malla uniforme en el espacio y tiempo.

Queremos definir $U$ como la función que satisface una ecuación en diferencias finitas basada en la ecuación del calor. Para ello, aproximaremos la derivada respecto al tiempo mediante un cociente incremental progresivo (Definición 2.2.1) y la derivada segunda respecto a x mediante un segundo cociente incremental (Definición 2.2.3).

El esquema resultante es el siguiente:

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = \frac{U_{i+1}^n - 2U_i^n + U_{i-1}^n}{\Delta x^2}. \tag{3.2}$$

Reordenando términos, obtenemos la siguiente fórmula explícita:

$$U_i^{n+1} = (1-2\lambda)U_i^n + \lambda(U_{i+1}^n + U_{i-1}^n), \qquad \text{con } \lambda := \frac{\Delta t}{\Delta x^2}. \tag{3.3}$$

Para construir la función $U$, fijamos primero sus valores en el contorno espacial y en el instante inicial, utilizando las condiciones del Problema (3.1):

$$\begin{cases} U_i^0 := f_i, & \text{para } i = 0, \ldots, n_x - 1, \\ U_0^n := \alpha(t_n),\ U_{n_x-1}^n := \beta(t_n), & \text{para } n = 0, \ldots, n_t - 1. \end{cases}$$

Después, para cada $n = 0, \ldots, n_t - 2$ y $i = 1, \ldots, n_x - 2$ definimos $U_i^{n+1}$ mediante la relación de recurrencia (3.3). Este procedimiento nos permite determinar completamente el valor de $U$ en todos los nodos de la malla.

En la Figura 3.1 se muestra de forma visual el mecanismo de recurrencia del método. A partir de los valores conocidos en el tiempo $t_n$, se calcula el valor de cada nodo en $t_{n+1}$ mediante la fórmula (3.3). El proceso se repite para todas las filas temporales de la malla.

### 3.1.3. Análisis de convergencia

Pasamos ahora a demostrar que el esquema propuesto en (3.3) proporciona una aproximación que converge a la solución de (3.1), es decir, que es **convergente**.

#### Consistencia

Recordamos que, siguiendo la Definición 2.3.1, el error de truncamiento en el punto $(x_i, t_n)$ se define como la diferencia entre el esquema (3.2) aplicado a la solución exacta $u$ y la ecuación diferencial original [6, p. 502], es decir:

$$\tau_i^n := \left[D_+^t u_i^n - D_{xx} u_i^n\right] - \left[\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2}\right]. \tag{3.4}$$

Reordenando esa expresión y aplicando el Teorema 2.3.1 y el Teorema 2.3.3 es inmediato que el error cumple $\tau_i^n = O(\Delta t + \Delta x^2)$ [2]. Por tanto, el esquema es **consistente** con orden uno en tiempo y dos en espacio.

[2] Al utilizar la notación $O$, se acota la magnitud total del error, por lo que al restar dos términos de órdenes diferentes se considera la suma de las cotas, es decir, $O(\Delta t) - O(\Delta x^2) = O(\Delta t + \Delta x^2)$.

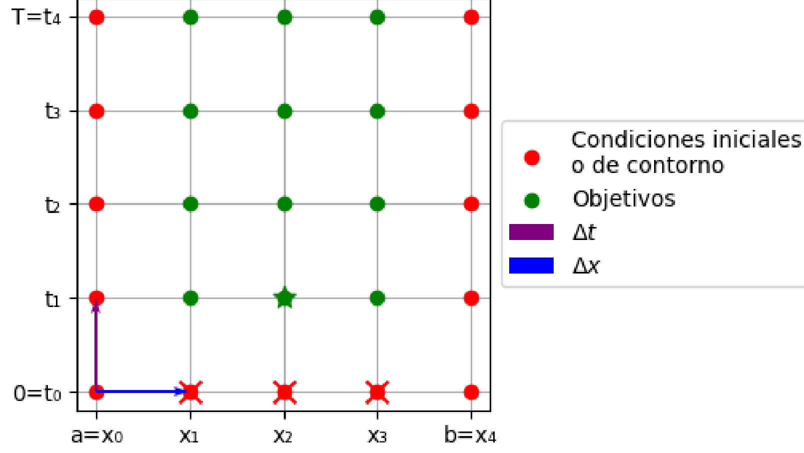


Figura 3.1: Representación en la malla del esquema (3.3). Para calcular el valor de $U$ en el punto marcado con una estrella necesitamos saber el valor de $U$ en los puntos marcados con una equis.

#### Convergencia

Si ahora restamos (3.2) menos (3.4) (habiendo despejado esta última), obtenemos

$$D_+^t U_i^n - D_+^t u_i^n = D_{xx} U_i^n - D_{xx} u_i^n - \tau_i^n.$$

Lo que, utilizando la Definición 2.3.2, resulta en

$$\frac{e_i^{n+1} - e_i^n}{\Delta t} = \frac{e_{i+1}^n - 2e_i^n + e_{i-1}^n}{\Delta x^2} - \tau_i^n.$$

Despejando obtenemos la siguiente fórmula explícita para el error:

$$e_i^{n+1} = (1 - 2\lambda) e_i^n + \lambda(e_{i+1}^n + e_{i-1}^n) - \Delta t \tau_i^n.$$

Si ahora aplicamos valores absolutos, la desigualdad triangular y máximos en ambos lados obtenemos

$$E^{n+1} \leq |1 - 2\lambda| E^n + 2|\lambda| E^n + \Delta t \tau^n.$$

Si $0 \leq \lambda \leq \frac{1}{2}$, entonces $0 \leq 1 - 2\lambda$, por tanto, dado que por definición $\tau^n \leq \tau$, obtenemos:

$$E^{n+1} \leq (1 - 2\lambda) E^n + 2\lambda E^n + \Delta t \tau = E^n + \Delta t \tau \Rightarrow E^{n+1} \leq E^n + \Delta t \tau.$$

Aplicando ahora esta desigualdad de forma recursiva, obtenemos:

$$E^n \leq E^0 + n\Delta t\tau = E^0 + t_n\tau \leq E^0 + T\tau.$$

Dado que $E^0 = 0$ (porque se han impuesto condiciones iniciales exactas) y que el error de truncamiento es de orden $O(\Delta t + \Delta x^2)$, se concluye que el error global también tiende a cero con ese orden. Esto prueba (según la Definición 2.3.3) que el esquema es convergente con orden uno en tiempo y dos en espacio siempre que $0 \leq \lambda \leq \frac{1}{2}$.

## 3.2. Ecuación del calor en dos dimensiones

En el caso bidimensional, la ecuación del calor se formula como

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

donde suponemos, sin pérdida de la generalidad, que el coeficiente de difusión térmica $\kappa$ es igual a uno[3].

Estudiaremos esta ecuación en un dominio espacial $\Omega = (a, b) \times (c, d) \subset \mathbb{R}^2$, con condiciones iniciales y de contorno de tipo Dirichlet. El dominio del problema será entonces $D = \overline{\Omega} \times [0, T]$.

El problema resultante es el siguiente:

$$\begin{cases} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}, & (x, y, t) \in D^\circ, \\ u(x, y, 0) = f(x, y), & (x, y) \in \overline{\Omega}, \\ u(x, y, t) = g(x, y, t), & (x, y, t) \in \partial\Omega \times [0, T]. \end{cases} \tag{3.5}$$

Además, para que el problema esté bien planteado, es necesario que $f(x, y) = g(x, y, 0)$ para todo $(x, y) \in \partial\Omega$.

### 3.2.1. Existencia y unicidad

Nuestro primer objetivo es establecer que el Problema (3.5) está bien planteado, es decir, que existe una única solución que depende de manera continua de los datos.

Para ello, primero enunciamos el siguiente teorema, cuya demostración puede encontrarse en [3, pg. 205][4].

**Teorema 3.2.1.** *Si $g = 0$, el Problema (3.5) tiene una única solución $u \in C^2(D)$.*

Este resultado se puede extender al caso de condiciones de contorno no homogéneas mediante un cambio de variable estándar. Si $g \in C^2$ [5], entonces el Problema (3.5) puede homogeneizarse mediante el cambio de variable $u = v + w$.

[3] Al igual que hicimos para el caso unidimensional, podemos eliminar el parámetro $\kappa$ mediante el cambio de variable $s = \kappa t$

[4] En la fuente original, el resultado se establece para cualquier dimensión espacial. Aquí lo enunciamos únicamente en el caso bidimensional.

[5] Puede afinarse más esta condición para ser menos restrictiva.

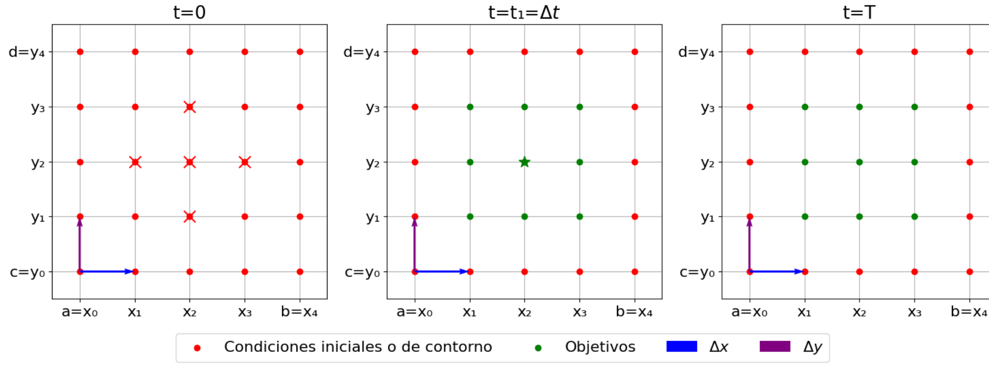


Figura 3.2: Representación en la malla del esquema (3.7) representado en capas temporales. Para calcular el valor de $U$ en el punto marcado con una estrella necesitamos saber el valor de $U$ en los puntos marcados con una equis.

### 3.2.2. Formulación discreta del problema

Al igual que en el caso unidimensional, construiremos una aproximación numérica de la solución del Problema (3.5) mediante un esquema en diferencias finitas.

Para ello, aproximamos la derivada respecto al tiempo mediante un cociente incremental progresivo (Definición 2.2.1) y las derivadas espaciales por segundos cocientes incrementales (Definición 2.2.3). El esquema resultante es:

$$\frac{U_{i,j}^{n+1} - U_{i,j}^{n}}{\Delta t} = \frac{U_{i+1,j}^{n} - 2U_{i,j}^{n} + U_{i-1,j}^{n}}{\Delta x^2} + \frac{U_{i,j+1}^{n} - 2U_{i,j}^{n} + U_{i,j-1}^{n}}{\Delta y^2}. \tag{3.6}$$

Reordenando términos, obtenemos la fórmula explícita:

$$U_{i,j}^{n+1} = (1 - 2\lambda_x - 2\lambda_y)U_{i,j}^{n} + \lambda_x(U_{i+1,j}^{n} + U_{i-1,j}^{n}) + \lambda_y(U_{i,j+1}^{n} + U_{i,j-1}^{n}), \tag{3.7}$$

donde:

$$\lambda_x := \frac{\Delta t}{\Delta x^2}, \qquad \lambda_y := \frac{\Delta t}{\Delta y^2}.$$

Para construir la función $U$, fijamos primero sus valores en el contorno espacial y en el instante inicial, utilizando las condiciones del Problema (3.5):

$$\begin{cases} U_{i,j}^{0} := f_{i,j}, & \text{para } i = 0, \dots, n_x - 1,\ j = 0, \dots, n_y - 1 \\ U_{i,j}^{n} := g(x_i, y_j, t_n), & \text{para } (x_i, y_j) \in \partial\Omega,\ n = 0, \dots, n_t - 1. \end{cases}$$

Después, para cada $n = 0, \dots, n_t - 2$, $i = 1, \dots, n_x - 2$, $j = 1, \dots, n_y - 2$ definimos $U_{i,j}^{n+1}$ mediante la relación de recurrencia (3.7). Este procedimiento nos permite determinar completamente el valor de U en todos los nodos de la malla.

La Figura 3.2 muestra de forma visual el mecanismo de recurrencia del método. A partir de los valores conocidos en el tiempo $t_n$, se calcula el valor de cada nodo en $t_{n+1}$ mediante la fórmula (3.7). El proceso se repite para todas las secciones temporales de la malla.

### 3.2.3. Análisis de convergencia

El análisis de la convergencia del esquema (3.7) sigue exactamente la misma estructura que en el caso unidimensional, desarrollada en la Subsección 3.1.3. A continuación, resumimos los resultados principales, indicando las diferencias específicas del caso bidimensional.

#### Consistencia

Recordamos que, siguiendo la Definición 2.3.1, el error de truncamiento en el punto $(x_i, y_j, t_n)$ se define como la diferencia entre el esquema (3.6) aplicado a la solución exacta $u$ y la ecuación diferencial original, es decir:

$$\tau_i^n := \left[D_+^t u_{i,j}^n - D_{xx} u_{i,j}^n - D_{yy} u_{i,j}^n\right] - \left[\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2}\right].$$

Reordenando esa expresión y aplicando el Teorema 2.3.1 y el Teorema 2.3.3 es inmediato que el error cumple $\tau_{i,j}^n = O(\Delta t + \Delta x^2 + \Delta y^2)$. Por tanto, el esquema es **consistente** con orden uno en tiempo y dos en ambas dimensiones espaciales.

#### Convergencia

Si aplicamos un desarrollo análogo al que hicimos para la ecuación del calor, suponiendo esta vez que $\lambda_x + \lambda_y \leq \frac{1}{2}$, se obtiene una recurrencia de la forma:

$$E^{n+1} \leq E^n + \Delta t \tau,$$

aplicando de manera recursiva esa desigualdad, teniendo en cuenta que $E^0 = 0$, volvemos a obtener que

$$E^n \leq n \Delta t \tau \leq T\tau.$$

Por tanto, concluimos que $E^n = O(\Delta t + \Delta x^2 + \Delta y^2)$, es decir, que el esquema es convergente con orden uno respecto al tiempo y dos respecto a las variables espaciales.

Capítulo 4

# Ecuación de onda

**Resumen:** En este capítulo presentamos la ecuación de onda en una y dos dimensiones espaciales. En cada caso formulamos primero el problema de valores iniciales y de contorno, y damos condiciones de existencia y unicidad de la solución. Tras ello, introduciremos discretizaciones en diferencias finitas y detallaremos los algoritmos resultantes.

## 4.1. Ecuación de onda en dimensión uno

En el caso unidimensional, la ecuación de onda se expresa como[1]:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}.$$

Consideramos el siguiente problema de valor inicial con condiciones iniciales de Cauchy.

$$\begin{cases} \frac{\partial^2 u}{\partial t^2}(x,t) - \frac{\partial^2 u}{\partial x^2}(x,t) = 0, & (x,t) \in D^\circ, \\ u(x,0) = f(x), & x \in \Omega, \\ \frac{\partial u(x,0)}{\partial t} = g(x), & x \in \Omega. \end{cases} \tag{4.1}$$

Aunque solo pedimos que la ecuación diferencial se cumpla en el interior del rectángulo $D$, supondremos que los datos iniciales $f$ y $g$ están definidos sobre todo $\mathbb{R}$, lo cuál nos permitirá construir una solución global y, posteriormente, restringirla a $D$.

Esta formulación se debe a que el esquema numérico explícito que utilizaremos más adelante requiere valores fuera del intervalo $\Omega$ para poder calcular aproximar la solución en $D$. De este modo, resolveremos el Problema (4.4) como una restricción de un problema más general planteado sobre $\mathbb{R} \times [0, T]$, cuyas condiciones iniciales y ecuación diferencial están bien definidas en todo el dominio.

[1] Al igual que en el caso de la ecuación del calor, la ecuación en realidad es de la forma $\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$, pero el cambio de variable $s = ct$ nos permite omitirlo sin pérdida de la generalidad.

### 4.1.1. Existencia y unicidad

Nuestro primer objetivo es demostrar que el Problema (4.1) admite una única solución suficientemente regular. Para ello, consideraremos primero el siguiente problema extendido, planteado sobre todo $\mathbb{R}$.

$$\begin{cases} \frac{\partial^2 u}{\partial t^2}(x,t) - \frac{\partial^2 u}{\partial x^2}(x,t) = 0, & (x,t) \in \mathbb{R} \times (0,T), \\ u(x,0) = f(x), & x \in \mathbb{R}, \\ \frac{\partial u(x,0)}{\partial t} = g(x), & x \in \mathbb{R}. \end{cases} \tag{4.2}$$

**Teorema 4.1.1** (Existencia y unicidad para el problema extendido, [8])**.** *Sean $f \in C^1(\mathbb{R})$ y $g \in C(\mathbb{R})$. Entonces, el Problema (4.2) admite una única solución $u \in C^2(\mathbb{R} \times [0,T])$, dada por la expresión*

$$u(x,t) = \frac{1}{2}[f(x+t) + f(x-t)] + \frac{1}{2}\int_{x-t}^{x+t} g(\zeta)d\zeta.$$

*Demostración.* Comenzamos aplicando el cambio de variable $\xi := x - t$, $\eta = x + t$. Aplicando la regla de la cadena, obtenemos que

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial \xi} + \frac{\partial u}{\partial \eta}, \quad \frac{\partial u}{\partial t} = -\frac{\partial u}{\partial \xi} + \frac{\partial u}{\partial \eta}.$$

Derivando nuevamente, obtenemos:

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} &= \frac{\partial^2 u}{\partial \xi^2} + 2\frac{\partial^2 u}{\partial \xi \partial \eta} + \frac{\partial^2 u}{\partial \eta^2}, \\ \frac{\partial^2 u}{\partial t^2} &= \frac{\partial^2 u}{\partial \xi^2} - 2\frac{\partial^2 u}{\partial \xi \partial \eta} + \frac{\partial^2}{\partial \eta^2}. \end{aligned}$$

Sustituyendo ambas expresiones en la ecuación de onda, se obtiene:

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} = -4\frac{\partial^2 u}{\partial \xi \partial \eta} = 0,$$

lo que implica que $\frac{\partial^2 u}{\partial \xi \partial \eta} = 0$. Por tanto, la solución general de esta ecuación tiene que ser de la forma:

$$u(x,t) = P(\xi) + Q(\eta) = P(x-t) + Q(x+t),$$

para ciertas funciones $P, Q \in C^2(\mathbb{R})$. Si ahora imponemos las condiciones iniciales, nos queda:

$$u(x,0) = P(x) + Q(x) = f(x), \tag{A}$$

$$\frac{\partial u}{\partial t}(x,0) = -P'(x) + Q'(x) = g(x). \tag{B}$$

Derivando (A) obtenemos $P'(x) + Q'(x) = f'(x)$, y despejando (B) obtenemos $Q'(x) - P'(x) = g(x)$. Sumando y restando ambas expresiones, obtenemos:

$$2Q'(x) = f'(x) + g(x), \quad 2P'(x) = f'(x) - g(x),$$

y por tanto:

$$P'(x) = \frac{1}{2}f'(x) - \frac{1}{2}g(x), \quad Q'(x) = \frac{1}{2}f'(x) + \frac{1}{2}g(x).$$

Integrando ambas expresiones en el intervalo $[0, x]$ obtenemos:

$$P(x) = \frac{1}{2}f(x) - \frac{1}{2}\int_0^x g(\zeta)d\zeta + K_1, \quad Q(x) = \frac{1}{2}f(x) + \frac{1}{2}\int_0^x g(\zeta)d\zeta + K_2,$$

para ciertas constantes $K_1, K_2 \in \mathbb{R}$. Ahora, si evaluamos en $x = 0$ y despejamos las constantes, podemos ver que

$$K_1 = P(0) - \frac{1}{2}f(0), \quad K_2 = Q(0) - \frac{1}{2}f(0). \tag{C}$$

Sumando ambas expresiones obtenemos

$$K_1 + K_2 = P(0) + Q(0) - f(0),$$

pero, si aplicamos (A), tenemos que $K_1 + K_2 = 0$. Si ahora, teniendo en cuenta esto, sustituimos en $u(x,t) = P(x-t) + Q(x+t)$ usando (C) y agrupamos términos, obtenemos

$$u(x,t) = \frac{1}{2}[f(x-t) + f(x+t)] + \frac{1}{2}\int_{x-t}^{x+t} g(\zeta)d\zeta,$$

Por tanto, hemos definido de manera única la solución buscada, por lo que podemos garantizar su existencia y unicidad. □

Dado que la solución $u$ del problema extendido es única y está definida sobre todo $\mathbb{R} \times [0, T]$, su restricción al dominio $D$ también es única y satisface las condiciones del problema original (4.1). Por tanto, la existencia y unicidad del problema extendido implica la del problema restringido.

### 4.1.2. Formulación discreta del problema

Procedemos ahora a construir una aproximación numérica de la solución del Problema (4.1) utilizando un esquema en diferencias finitas.

Al igual que en las secciones anteriores, discretizamos el dominio mediante una malla uniforme en el espacio y el tiempo, según la notación introducida en la Subsección 2.1.2.

Queremos definir $U$ como una función que satisfaga una ecuación en diferencias finitas basada en la ecuación de onda. Para ello, aproximamos las derivadas segundas que aparecen en la ecuación de onda por segundos cocientes incrementales (Definición 2.2.3).

El esquema resultante es el siguiente:

$$\frac{U_i^{n+1} - 2U_i^n + U_i^{n-1}}{\Delta t^2} = \frac{U_{i+1}^n - 2U_i^n + U_{i-1}^n}{\Delta x^2}.$$

Despejando, obtenemos la siguiente fórmula explícita:

$$U_i^{n+1} = 2(1-\lambda^2)U_i^n + \lambda^2(U_{i+1}^n + U_{i-1}^n) - U_i^{n-1}, \quad \text{donde } \lambda := \frac{\Delta t}{\Delta x}. \tag{4.3}$$

Para aplicar este esquema, necesitamos primero fijar los valores de $U_i^0$ y $U_i^1$, para lo que utilizaremos las condiciones iniciales del Problema (4.1):

- El valor en el instante inicial $n = 0$ se fija directamente como:

$$U_i^0 := f_i, \quad \forall i \in \mathbb{N}.$$

- Para el siguiente instante, $n = 1$, utilizaremos el desarrollo de Taylor de orden dos centrado en $t = 0$:

$$u(x, \Delta t) = u(x, 0) + \Delta t \frac{\partial u}{\partial t}(x, 0) + \frac{\Delta t^2}{2}\frac{\partial^2 u}{\partial t^2}(x, 0) + R_2(\Delta t),$$

dónde el término del resto es $O(\Delta t^3)$. Si utilizamos la igualdad $\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}$ y aproximamos esa derivada por el segundo cociente incremental, obtenemos:

$$U_i^1 := f_i + \Delta t g_i + \frac{\lambda^2}{2}(f_{i+1} - 2f_i + f_{i-1}), \quad \forall i \in \mathbb{N}.$$

Más adelante necesitaremos una cota asintótica del error global para $t = \Delta t$, es decir, $e_i^1 = U_i^1 - u_i^1$. Si tenemos en cuenta que $f_i = u(x_i, 0)$ y que $\frac{\Delta u}{\Delta t}(x_i, 0) = g_i$, al restar las dos expresiones anteriores se nos anulan los dos primeros sumandos, por lo que obtenemos que

$$|U_i^1 - u_i^1| = \left|\frac{\Delta t^2}{2}\left[(f_{i+1} - 2f_i + f_{i-1} - \frac{\partial^2 u}{\partial x^2}(x_i, 0))\right] - R_2(\Delta t)\right| \leq$$
$$\frac{\Delta t^2}{2}\left|(D_{xx}f)(x_i, 0) - \frac{\partial^2 u}{\partial x^2}(x_i, 0)\right| + |R_2(\Delta t)|.$$

Teniendo en cuenta el Teorema 2.3.3 y que el resto de Lagrange es orden $O(\Delta t^3)$, el error global $|e_i^1| = O(\Delta t^2 \Delta x^2 + \Delta t^3)$ luego $E^1 = O(\Delta t^2 \Delta x^2 + \Delta t^3)$.

Al igual que en el caso de la ecuación del calor, el cálculo de $U_i^{n+1}$ en un nodo dado requiere conocer los valores de $U$ en nodos vecinos en el instante anterior. Sin embargo, a diferencia del caso del calor, el esquema de la ecuación de onda requiere además conocer el valor en el instante $t_{n-1}$, es decir, dos instantes temporales previos.

Para poder aplicar este esquema y calcular los valores de $U$ dentro del rectángulo $D$, es necesario extender las condiciones iniciales a un intervalo espacial más amplio. Esto se debe a que, a diferencia de como hicimos en la ecuación del calor, no hemos impuesto condiciones de contorno de Dirichlet, lo que obliga a conocer valores de $f$ y $g$ en un intervalo mayor que $(a, b)$, cuyo tamaño depende de $T$. Gracias a que hemos supuesto en la formulación del problema que $f$ y $g$ están definidas en todo $\mathbb{R}$, esto no genera ningún problema.

La Figura 4.1 muestra gráficamente el mecanismo de recurrencia del método. A partir de los valores en los instantes $t_0$ y $t_1$, se puede calcular el valor de cada nodo en $t_2$, y así sucesivamente.

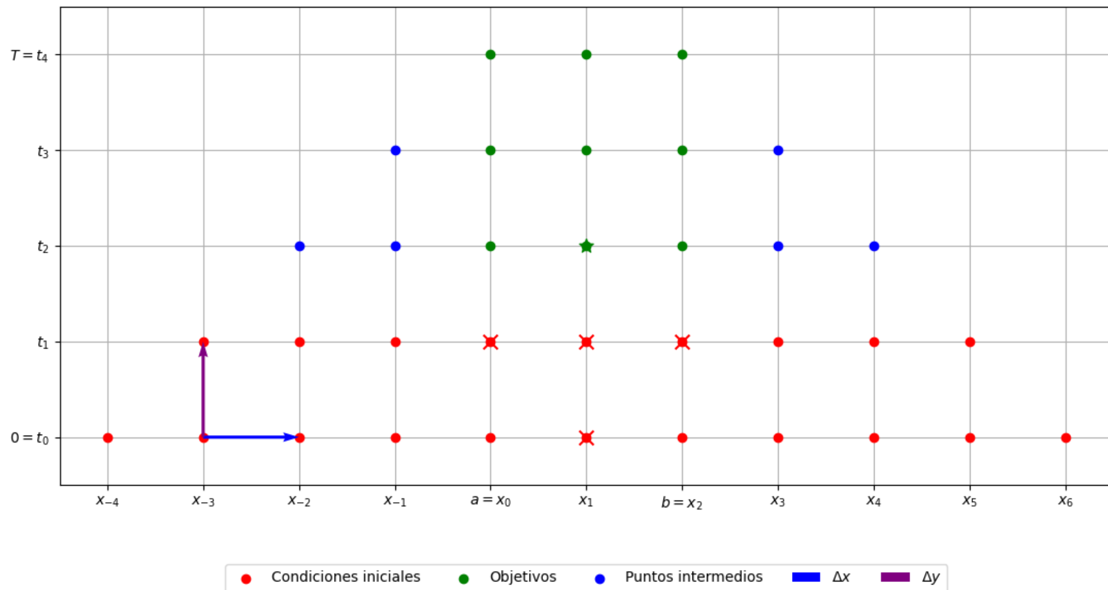


Figura 4.1: Representación en la malla del esquema (4.3). Para calcular el valor $U$ en el punto marcado con una estrella necesitamos saber el valor $U$ en los puPretender que en la sección en la que das los coeficientes para las simulaciones, el tribunal que lo lee, y tiene muchos trabajos que leer se acuerde de qué es cada cosa, o se ponga a buscar en qué parte de la memoria estaba, redunda en que el tribunal te penalize la escritura de la memoria. Si eso es lo que quieres, no tienes más que dejarlo como está.ntos marcados con una equis.

### 4.1.3. Análisis de convergencia

A continuación enunciamos un resultado de convergencia del esquema (4.3) para ciertos valores de $\lambda$.

**Teorema 4.1.2.** *Si $\lambda \leq 1$ y $f, g$ son suficientemente diferenciables, el método numérico dado por* (4.3) *es convergente.*

*Demostración.* Ver [6, Sección 9.3.2]. □

## 4.2. Ecuación de onda en dos dimensiones

En el caso bidimensional, la ecuación de onda se formula como

$$\frac{\partial^2 u}{\partial t^2} - \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = 0,$$

donde suponemos, sin pérdida de la generalidad, que la velocidad de propagación es igual a uno[2].

Consideramos el siguiente problema

$$\begin{cases} \frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, & (x,y,t) \in D^\circ, \\ u(x,y,t) = 0, & (x,y,t) \in \partial\Omega \times [0,T], \\ u(x,y,0) = f(x,y), & (x,y) \in \Omega, \\ \frac{\partial u}{\partial t}(x,y,0) = g(x,y), & (x,y) \in \Omega, \end{cases} \tag{4.4}$$

donde además supondremos que $f$ y $g$ están definidas en todo $\mathbb{R}^2$ y que $f, g \in C^2(\mathbb{R}^2)$.

### 4.2.1. Existencia y unicidad

El Problema (4.4) tiene una única solución. Este resultado puede consultarse en [3, pg. 214].

### 4.2.2. Formulación discreta del problema

Procedemos ahora construir una aproximación numérica de la solución del Problema (4.4) utilizando un esquema en diferencias finitas. Para ello, discretizaremos el dominio $D$ según la notación introducida en la Subsección 2.1.2, utilizando una malla uniforme en el espacio y tiempo.

Queremos definir $U$ como la función que satisface una ecuación en diferencias finitas basada en la ecuación de onda. Para ello, aproximamos las derivadas por cocientes incrementales dobles (Definición 2.2.3).

El esquema resultante es el siguiente:

$$\frac{U_{i,j}^{n+1} - 2U_{i,j}^n + U_{i,j}^{n-1}}{\Delta t^2} = \frac{U_{i+1,j}^n - 2U_{i,j}^n + U_{i-1,j}^n}{\Delta x^2} + \frac{U_{i,j+1}^n - 2U_{i,j}^n + U_{i,j-1}^n}{\Delta y^2}. \tag{4.5}$$

Reordenando, obtenemos un esquema explícito de la forma:

$$U_{i,j}^{n+1} = 2(1 - \lambda_x^2 - \lambda_y^2)U_{i,j}^n + \lambda_x^2(U_{i+1,j}^n + U_{i-1,j}^n) + \lambda_y^2(U_{i,j+1}^n + U_{i,j-1}^n) - U_{i,j}^{n-1}, \tag{4.6}$$

donde:

$$\lambda_x := \frac{\Delta t}{\Delta x}, \quad \lambda_y := \frac{\Delta t}{\Delta y}$$

Para aplicar el esquema, necesitamos fijar algunos valores iniciales y de contorno:

- El valor en el instante inicial $n = 0$ se fija directamente como:

$$U_{i,j}^0 := f_{i,j} \quad \forall i,j \mid (x_i, y_j) \in \Omega.$$

[2] Al igual que hicimos para el caso unidimensional, podemos eliminar el parámetro $c$ mediante el cambio de variable $s := ct$.

- Para el siguiente instante, $n = 1$, utilizamos el desarrollo de Taylor de orden dos centrado en $t = 0$:

$$u(x, y, \Delta t) = u(x, y, 0) + \Delta t \frac{\partial u}{\partial t}(x, y, 0) + \frac{\Delta t^2}{2}\frac{\partial^2 u}{\partial t^2}(x, y, 0) + O(\Delta t^3).$$

  Si ahora sustituimos la segunda derivada en el tiempo por la suma de las segundas derivadas del espacio, y aplicamos la Definición 2.2.3, obtenemos para cualquier $(i, j) \in \Omega$

$$U^1_{i,j} := f_{i,j} + \Delta t g_{i,j} + \frac{\lambda_x^2}{2}(f_{i+1,j} - 2f_{i,j} + f_{i-1,j}) + \frac{\lambda_y}{2}(f_{i,j+1} - 2f_{i,j} + f_{i,j-1}).$$

  De manera similar al caso unidimensional, el error cometido en esta aproximación es $O(\Delta t^2 \Delta x^2 + \Delta t^2 \Delta y^2 + \Delta t^3)$.

- Dado que tenemos condiciones de contorno de Dirichlet homogéneas, definimos:

$$U^n_{i,j} := 0 \quad \forall i, j \mid (x_i, y_j) \in \partial\Omega$$

La Figura 4.2 ilustra cómo se aplica el esquema en cada nodo interior de la malla. Cada nuevo valor se calcula a partir de su valor en los dos instantes anteriores y de los vecinos espaciales en la capa temporal anterior.

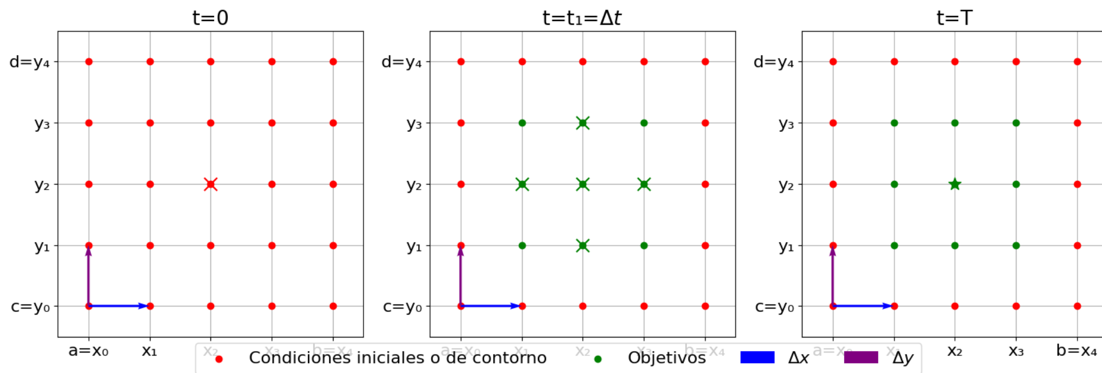


Figura 4.2: Representación en la malla del esquema (4.6). Para calcular el valor $U$ en el punto marcado con una estrella necesitamos saber el valor $U$ en los puntos marcados con una equis.

### 4.2.3. Análisis de convergencia

Para demostrar la convergencia del esquema (4.5) se requieren demostraciones matriciales más complejas que para la versión unidimensional que se escapan a los objetivos de este trabajo.

Capítulo 5

# Ecuación de Laplace

**Resumen:** En este capítulo presentamos la ecuación de Laplace en dos dimensiones. Formulamos el problema con condiciones de contorno de Dirichlet en todo el borde del dominio, establecemos condiciones que aseguran la existencia y unicidad de la solución, y finalmente introducimos una discretización por el método de diferencias finitas basada en el esquema de cinco puntos.

La ecuación de Laplace en dos dimensiones es:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0.$$

Consideramos el siguiente problema de contorno, con condiciones de Dirichlet:

$$\begin{cases} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, & (x, y) \in \Omega, \\ u(x, y) = g(x, y), & (x, y) \in \partial\Omega. \end{cases} \tag{5.1}$$

## 5.1. Existencia y unicidad

Es bien sabido que el problema de Dirichlet para la ecuación de Laplace en un dominio rectangular tiene una única solución siempre que la función $g$ sea suficientemente regular.

## 5.2. Formulación discreta del problema

Procedemos ahora a construir una aproximación numérica de la solución del Problema (5.1) utilizando un esquema en diferencias finitas.

Discretizamos el dominio mediante una malla uniforme en el espacio, según la notación introducida en la Subsección 2.1.2.

Queremos definir $U$ como una función que satisfaga una ecuación en diferencias finitas basada en la ecuación de Laplace. Para ello, aproximamos las derivadas que aparecen en la ecuación por segundos cocientes incrementales (Definición 2.2.3).

El esquema resultante es:

$$\frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{\Delta x^2} + \frac{U_{i,j+1} - 2U_{i,j} + U_{i,j-1}}{\Delta y^2} = 0. \tag{5.2}$$

Además, pediremos que $U$ cumpla las condiciones de contorno, es decir:

$$U_{i,j} = g_{i,j}, \qquad \forall i, j \mid (x_i, y_j) \in \partial\Omega.$$

En los capítulos anteriores, cuando construíamos el problema discreto (es decir, encontrar una función $U$ que cumpla la versión discretizada de la ecuación diferencial y respete los valores iniciales y/o de contorno), la existencia y unicidad de la solución del problema era inmediata, ya que podíamos despejar el esquema y obtener una fórmula explícita de $U$, que junto con los valores iniciales nos daba una construcción única de esta.

Debido a la naturaleza estacionaria de la ecuación de Laplace, esto no puede hacerse. En este caso, al construir el problema discreto obtenemos un sistema de ecuaciones lineales, por lo que tendremos que probar su existencia y unicidad.

### 5.2.1. Formulación matricial del problema

Para pasar del esquema (5.2) a un único sistema lineal, seguiremos un razonamiento basado en [6] para simplificar los cocientes:

- El paso efectivo $\lambda^2 := \frac{\Delta x^2 \Delta y^2}{2(\Delta x^2 + \Delta y^2)}$.
- $\lambda_x := \frac{\lambda^2}{\Delta x^2} = \frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)}$.
- $\lambda_y := \frac{\lambda^2}{\Delta y^2} = \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)}$.

Es sencillo comprobar que $2\lambda_x + 2\lambda_y = 1$, por lo que, si multiplicamos (5.2) por $-\lambda^2$ obtenemos el esquema normalizado

$$U_{i,j} = \lambda_x(U_{i-1,j} + U_{i+1,j}) + \lambda_y(U_{i,j-1} + U_{i,j+1}), \quad 1 \le i \le n_x - 2,\ 1 \le j \le n_y - 2. \tag{5.3}$$

En esta expresión, cada término corresponde al aporte de un vecino; cuando alguno de ellos pertenece a la frontera (es decir, cuando $i + 1 = 0, n_x - 1$ o $j + 1 = 0, n_y - 1$) su valor ya no es incógnita sino $g_{i,j}$. Para recoger esas contribuciones definimos el término siguiente vector:

$$b_{i,j} := \begin{cases} \lambda_x\, g_{0,j}, & \text{si } i = 1, \\ \lambda_x\, g_{n_x-1,j}, & \text{si } i = n_x - 2, \\ 0, & \text{en otro caso,} \end{cases} + \begin{cases} \lambda_y\, g_{i,0}, & \text{si } j = 1, \\ \lambda_y\, g_{i,n_y-1}, & \text{si } j = n_y - 2, \\ 0, & \text{en otro caso.} \end{cases}$$

Para llevar el problema esto a la forma estándar $A\,\mathbf{U} = \mathbf{b}$, necesitamos "aplanar" la malla 2D en un vector unidimensional. Usamos numeración lexicográfica por filas:

$$k := (j-1)(n_x - 2) \;+\; (i-1), \qquad \mathbf{U}_k = u_{i,j}, \quad k = 0, \ldots, (n_x-2)(n_y-2) - 1.$$

De este modo, el vector $\mathbf{U} \in \mathbb{R}^N$ (con $N = (n_x - 2)(n_y - 2)$) recoge en su entrada $k$ la incógnita de la posición $(i, j)$. Análogamente apilamos en $\mathbf{b}$ las cantidades $b_{i,j}$.

Con todo lo que hemos definido, podemos reescribir el Problema (5.3) con las condiciones de contorno como $\mathbf{U} = M\mathbf{U} + \mathbf{b}$, donde la matriz $M \in \mathbb{R}^{N \times N}$ tiene únicamente cuatro diagonales no nulas, cada una correspondiente a uno de los sumandos de la expresión (5.3):

- La primera superdiagonal (es decir, los puntos $(k, k+1)$ representa al vecino *este*, con valor $\lambda_x$.
- La primera subdiagonal (es decir, los puntos $(k, k-1)$) representa al vecino *oeste*, con valor $\lambda_x$.
- La $(n_x - 2)$-ésima superdiagonal representa al vecino *norte*, con valor $\lambda_y$.
- La $(n_x - 2)$-ésima subdiagonal representa al vecino *sur*, con valor $\lambda_y$.

Reordenando, obtenemos el sistema

$$A\mathbf{U} = \mathbf{b}, \qquad \text{con } A := (I - M). \tag{5.4}$$

Aunque nuestros razonamientos emplean una notación y unos pasos algo distintos a [6, Ch.9], puede comprobarse que el sistema matricial que aquí presentamos coincide con el suyo cuándo $f \equiv 0$. En ese mismo libro podemos encontrar la demostración de que el sistema lineal obtenido tiene solución única.

## 5.3. Análisis de convergencia

### Consistencia

Aplicando el Teorema 2.3.3 y la Definición 2.3.1, obtenemos de manera sencilla que

$$\tau_{i,j} = O(\Delta x^2 + \Delta y^2),$$

lo que implica que el esquema (5.2) es **consistente** con orden dos en ambas dimensiones espaciales.

### Convergencia

En [6, p. 450] se demuestra la siguiente cota para el error global de aproximación:

$$e_{i,j} \leq \frac{b^2}{2}\tau.$$

Se deduce de esta cota y del hecho de que el esquema es consistente con orden dos en ambas dimensiones espaciales que el esquema es **convergente** con el mismo orden.

## 5.4. Resolución del sistema $A\mathbf{U} = \mathbf{b}$

En las secciones anteriores hemos concluido que si aproximamos la solución exacta $u$ del Problema (5.1) por la solución $U$ del sistema (5.4) obtenemos una aproximación que converge a la solución. No obstante, a diferencia de la ecuación del calor y onda, no es inmediato encontrar la función $U$. Aunque el sistema tiene solución única, su tamaño es de $N \times N$. En lugar de invertir $A$, usaremos un método iterativo para aproximar la función $U$.

Existen muchos métodos bien conocidos para resolver sistemas de ecuaciones, en concreto, en [6] se consideran los siguientes:

- El método de Jacobi.
- El método de Gauss-Seidel
- El método de Gauss-Seidel acelerado.
- El método de Gauss-Seidel aplicado a bloques.
- El método de Peaceman-Rachford para alternar direcciones.

De estos, vamos a elegir el método de Jacobi. Este es un método iterativo sencillo (especialmente para nuestro caso, donde la diagonal de $A$ es la identidad)[1].

### 5.4.1. Método de Jacobi

**Inicialización** $\mathbf{U}^{(0)} = 0$.

**Iteración** $\mathbf{U}^{(n+1)} := M\mathbf{U}^{(n)} + \mathbf{b}$.

**Criterio de parada** Para poder asegurar que el error del método de Jacobi sea menor que el introducido por la aproximación basada en diferencias finitas, implementamos simultáneamente dos criterios de parada que dependen de un parámetro definido como $h = \text{máx}(\Delta x, \Delta y)$:

- **Mínimo de mejora**: Definimos la mejora en el paso $n$ como

$$\delta^{(n)} := ||U^{(n)} - U^{(n-1)}||_\infty = \max_{i,j} |U_{i,j}^{(n)} - U_{i,j}^{(n-1)}|.$$

La condición de parada es que la mejora obtenida en el último paso sea *demasiado pequeña*, lo que indica que el método se está estabilizando. En nuestro caso, es razonable definir *demasiado pequeña* como el error que induce la aproximación de $u$ por $U$, ya que no obtenemos ninguna

[1] A pesar de que el método de Jacobi es el que peor convergencia tiene de los que propone [6], hemos optado por éste ya que es el más sencillo de implementar, y el objetivo de nuestro trabajo no es desarrollar los métodos más eficientes, sino comparar el tiempo que tarda un método en computación clásica y en computación paralela en la GPU.

mejora real por mejorar la aproximación de $U$ por encima de este error. Por tanto, la condición de parada resulta en

$$\delta^{(n)} \leq Ch^2$$

para algún $C < 1$. En nuestro caso concreto hemos elegido $C = 0{,}01$.

- **Cota máxima de iteraciones**: Es habitual poner un máximo de iteraciones para evitar bucles excesivamente largos. En nuestro caso tomaremos
$$k_{\text{máx}} := \frac{1}{Ch^2}$$
para el $C$ definido en el punto anterior. Dada la naturaleza del método de Jacobi y nuestro problema concreto, nos garantiza un número de iteraciones del orden necesario para que el error iterativo quede al nivel del truncamiento $O(\Delta x^2 + \Delta y^2)$.

### 5.4.2. Formulación componente a componente

En las secciones anteriores hemos formulado un método iterativo basado en la multiplicación de matrices. Sin embargo, para propósitos de implementación numérica, resulta más conveniente expresar el método componente a componente. De este modo, la fórmula nos quedaría

$$U_{i,j}^{(k+1)} = \lambda_x(U_{i+1,j}^{(k)} + U_{i-1,j}^{(k)}) + \lambda_y(U_{i,j+1}^{(k)} + U_{i,j-1}^{(k)}) \quad \forall (x_i, y_j) \in \Omega. \tag{5.5}$$

Para introducir las condiciones de contorno en esta fórmula, simplemente definimos $U_{i,j}^{(k)} := g_{i,j}$ para los puntos $(x_i, y_j) \in \partial\Omega$.

Capítulo 6

# Introducción a la computación en GPU

**Resumen:** En este capítulo se presentan los conceptos mínimos necesarios para programar en la GPU con CUDA y lanzar los cálculos desde Python mediante la librería PyCUDA. Se explica cómo se organizan los hilos en bloques y rejillas, y por qué esta estructura permite acelerar los esquemas explícitos de diferencias finitas al asignar, en cada iteración, un hilo a cada nodo de la malla.

## 6.1. Motivación y paralelismo masivo

Los esquemas que hemos presentado aplican una misma operación de forma repetida en todos los nodos de una malla.

En la CPU (con pocos núcleos versátiles) estas operaciones se ejecutan de manera completamente secuencial; la GPU, en cambio, dispone de miles de núcleos ligeros que pueden aplicar dicha operación de forma **simultánea** sobre un gran número de nodos, reduciendo drásticamente el tiempo total de cálculo.

En cada iteración del algoritmo (un paso temporal o un barrido Jacobi) lanzaremos un núcleo que asigna un hilo a cada nodo de la malla.

## 6.2. Fundamentos de CUDA

CUDA es una extensión de C/C++ propuesta por NVIDIA para programar la GPU como un coprocesador masivamente paralelo [1]. En lugar de ejecutar todo el algoritmo en la CPU, se identifican aquellas operaciones susceptibles de paralelización y se agrupan en una función especial denominada núcleo.

Cada invocación del núcleo crea miles de hilos que ejecutan, de manera simultánea, el mismo cuerpo de código sobre datos distintos. El programa que se ejecuta en la CPU (el cual, en este trabajo, ha sido implementado en Python) se denomina anfitrión, y la GPU que está ejecutando los núcleos se denomina dispositivo.

### 6.2.1. Hilo, bloque y rejilla

- **Hilo**: Unidad mínima de ejecución.
- **Bloque**: Conjunto de hilos. Cada hilo dentro del bloque puede identificarse mediante las variables `threadIdx.(x,y,z)`. Además, el tamaño del bloque está almacenado en la variable `blockDim.(x,y,z)`.
- **Rejilla**: Rejilla rectangular de bloques que abarca todos los hilos lanzados en una llamada al núcleo. Cada bloque puede identificarse dentro del rejilla mediante las variables `blockIdx.(x,y,z)`.

En nuestro caso solamente necesitaremos bloques y rejillas de una o dos dimensiones, por lo que ignoraremos la dimensión z de las coordenadas. Es posible asignar de forma unívoca un par de coordenadas $(i, j)$ a cada hilo del rejilla de la siguiente forma

$$i = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x},$$
$$j = \texttt{blockIdx.y} \times \texttt{blockDim.y} + \texttt{threadIdx.y},$$

lo que permite asignar de forma natural cada hilo a un nodo de la malla numérica.

En todas las pruebas emplearemos bloques de 256 hilos en los problemas unidimensionales y $32 \times 8$ hilos en los bidimensionales, una elección habitual en programación CUDA, ya que ofrece un equilibrio adecuado entre uso de recursos y eficiencia computacional, tal como se ha observado en evaluaciones de banco de pruebas paralelos [2][1].

### 6.2.2. Memoria

La GPU y la CPU tienen memorias independientes, por esto, para ejecutar un núcleo sobre ciertos datos de entrada es necesario:

- Copiar los datos de entrada desde el anfitrión al dispositivo.
- Ejecutar uno o varios núcleos.
- Copiar los resultados de vuelta desde el dispositivo hasta el anfitrión.

Estas transferencias presentan una latencia fija, del orden de milisegundos [1]. Debido a esa latencia, la GPU puede tardar más tiempo que la CPU en resolver el problema si este es muy pequeño.

La librería PyCUDA, que presentamos en la siguiente sección, se encarga de gestionar las copias entre las memorias con funciones simples como `memcpy_htod` (del anfitrión al dispositivo) y `memcpy_dtoh` (del dispositivo al anfitrión).

[1] Un análisis exhaustivo para cada problema podría permitirnos crear algoritmos incluso más eficaces, no obstante, en este trabajo hemos decidido no implementar esas mejoras porque supondrían un estudio en profundidad del funcionamiento concreto de CUDA para obtener una mejora marginal comparada con los ordenes de mejora que obtendremos al pasar de implementación en CPU a implementación en GPU.

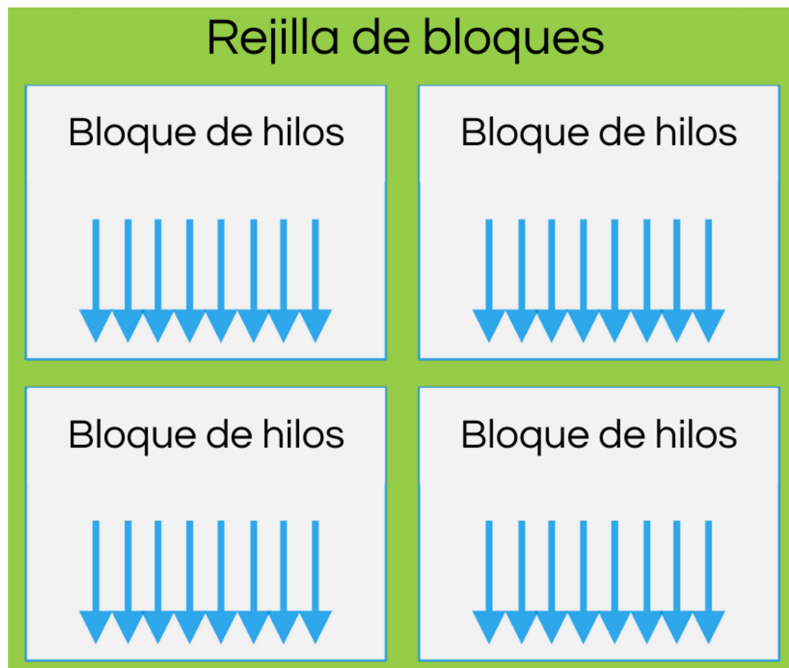


Figura 6.1: Rejilla de bloques de hilos

**Llamadas bloqueantes versus no bloqueantes**

En PyCUDA, las funciones de copia por defecto son bloqueantes, de manera que el anfitrión espera hasta que la transferencia finalice antes de continuar. Para beneficiarse realmente del paralelismo de los flujos (concepto que introduciremos en la siguiente sección), es necesario utilizar las variantes asíncronas (`memcpy_htod_async` y `memcpy_dtoh_async`), que permiten encolar la operación y devolver el control al programa inmediatamente.

## 6.3. Modelo de ejecución y flujos

En CUDA, todas las solicitudes que el anfitrión envía al dispositivo (ya sean transferencias de memoria o lanzamientos de núcleos) se organizan en una o varias colas de ejecución, denominadas flujos. Conceptualmente, un flujo es una lista FIFO[2] de operaciones que la GPU procesará en el orden en que fueron encoladas, garantizando que cada instrucción dentro de un mismo flujo respete la secuencia programada.

De forma predeterminada, todas las operaciones se encolan en un único flujo:

- Las transferencias de memoria (`cudaMemcpy` o `memcpy_htod`) y los lanzamientos de núcleos se ejecutan de forma secuencial y bloqueante: la operación siguiente no comenzará hasta que la anterior haya finalizado.
- Este modelo es sencillo y garantiza determinismo en la orden de ejecución, pero

[2] *First in First out*, es decir, que las peticiones se procesan en orden de llegada.

puede desaprovechar la capacidad de la GPU de solapar cómputo y movimiento de datos.

Para extraer un mayor rendimiento, CUDA permite crear múltiples flujos. Cada flujo mantiene su propia cola de operaciones, de modo que la GPU puede intercalar tareas de distintas colas siempre que no existan dependencias explícitas. En particular, es posible solapar:

1. Copias de memoria en un flujo (`memcpy_dtoh_async` o `memcpy_htod_async`),
2. Lanzamientos de núcleos no bloqueantes en otro flujo.

De esta forma, mientras un núcleo se ejecuta sobre un bloque de datos, la GPU puede simultáneamente transferir el bloque anterior al anfitrión, reduciendo tiempos muertos y mejorando el rendimiento global en los casos en los que, por la naturaleza del algoritmo, podamos copiar datos y calcular resultados de manera independiente.

## 6.4. Sincronización entre flujos y con el anfitrión

En nuestras implementaciones emplearemos únicamente tres mecanismos de sincronización:

### 6.4.1. Eventos para coordinar flujos

Para garantizar que una operación en un flujo no comience hasta que haya finalizado una tarea dependiente en otro, utilizamos *eventos* como marcadores de finalización. Desde un punto de vista conceptual, un evento marca un punto en la cola FIFO de un flujo y permite a otros flujos sincronizar su ejecución con respecto a dicha marca.

Pongamos un ejemplo muy simplificado con dos flujos, A y B, cada uno con una tarea que realizar y la dependencia $A \to B$[3]:

1. Se instancia un objeto de evento que luego registraremos en un flujo.
2. Encolar la operación del flujo A.
3. Encolar la señalización del evento en el flujo A. Cuando la operación A finalice, el evento se marcará como ocurrido.
4. En B, encolar primero la espera al evento. La GPU no procesará ninguna operación en B hasta que reciba esta señal.
5. Encolar la operación del flujo B. Como la espera está antes en la cola, la operación B aguardará hasta que A y la señalización del evento hayan concluido.

De este modo, aunque A y B residan en flujos distintos, la dependencia $A \to B$ queda garantizada sin bloquear otros flujos de ejecución.

[3] Aquí forzamos la dependencia usando dos flujos aunque bastaría con uno; el objetivo es ilustrar el uso de eventos.

### 6.4.2. Sincronización anfitrión–dispositivo

#### Synchronize

Para poder asegurarnos desde el anfitrión que se han realizado todas las operaciones encoladas hasta ese momento, podemos utilizar la función `synchronize` (que ofrecen tanto CUDA como PyCUDA), que bloquea la ejecución del anfitrión hasta que todas las operaciones de un flujo concreto hayan sido finalizadas.

Con esto, podemos asegurarnos de que la GPU ha terminado sus operaciones antes de liberar memoria o de utilizar los resultados en la CPU.

#### Eventos

En ocasiones resulta más eficiente sincronizar el hilo de Python con puntos concretos de la ejecución en la GPU, sin bloquear todas las operaciones en un flujo. Para ello, podemos hacer que el anfitrión espere a un evento concreto en lugar de a la finalización de todo un flujo.

## 6.5. Programación en Python con PyCUDA

PyCUDA permite compilar y lanzar núcleos CUDA directamente desde Python [7], de modo que sólo se necesita escribir el núcleo en C/C++ mientras que el resto del programa (generación de datos, post-proceso y cronometraje) permanece en Python.

### 6.5.1. Flujo de trabajo básico

Aunque cada proyecto concreto puede requerir un flujo ligeramente distinto, el uso general de PyCUDA para ejecutar cierta tarea en GPU podría ser:

1. **Compilación en tiempo de ejecución**: Un archivo `.cu` con la definición del núcleo se pasa a `SourceModule`[4]; la librería invoca al compilador `nvcc` y carga el código resultante en la GPU.

2. **Reserva de memoria**: `mem_alloc(nbytes)` crea un búfer en la memoria de la GPU.

3. **Encolado de operaciones y copias**: En uno o varios de los flujos creados se ponen en cola de manera asíncrona:

    - Transferencias de datos desde el anfitrión a la GPU o viceversa.
    - Lanzamiento de núcleos.
    - Señalizacioines y esperas de eventos para sincronizar adecuadamente las transferencias y los núcleos.

[4] Esto es un módulo propio con el objetivo de simplificar el uso de los núcleos. Su código puede encontrarse en A.1

4. **Sincronización final**: Una vez encoladas todas las tareas necesarias, utilizamos la función de sincronización para esperar hasta que todas hayan terminado.

## 6.6. Ejemplos ilustrativos

Con los conceptos anteriores hemos creado dos pequeños programas que ejemplifican cómo crear un programa en Python que, mediante PyCUDA, ejecute un núcleo en GPU; el código completo de los ejemplos se puede encontrar en el Apéndice A.

### 6.6.1. Hola mundo

Siguiendo la convención habitual en programación, el primer ejemplo consiste en un programa que imprime en consola la frase "Hola mundo".

- En A.2 vemos como el anfitrión utiliza `generateMod.py` para compilar y cargar el núcleo A.3, así como reservar memoria en el dispositivo y ejecutar el núcleo (especificando el tamaño de bloque y del rejilla).

- En A.3 vemos como nuestro núcleo en CUDA define una función (con la etiqueta `__global__`, que le indica al compilador que esta será ejecutada en GPU) que imprime la frase `"Hello World, my id is ... and my number is ...`, utilizando tanto el id del hilo como el número que le pasó el anfitrión al dispositivo.

- En A.4 podemos ver la salida del programa, que coincide con lo esperado.

### 6.6.2. Suma de vectores

A continuación se presenta un ejemplo que pone de manifiesto la mejora de rendimiento que conseguimos con el uso de GPU. En este caso nuestro programa anfitrión implementa una función (en CPU) para sumar dos vectores de tamaños arbitrarios. Luego, utilizando el núcleo A.6, realiza la suma de los mismos vectores, esta vez en GPU.

El programa A.2 define estas dos funciones, genera vectores de tamaños cada vez más grandes, y ejecuta las dos funciones anteriores, midiendo el tiempo que tarda cada uno y ofreciéndonos como salida los resultados y la mejora de eficiencia.

Como podemos observar en A.7, la mejora que introduce la GPU es bastante significativa, y va aumentando junto con el tamaño de los vectores.

Capítulo 7

# Implementación de los algoritmos

**Resumen:** En este capítulo implementamos en Python los esquemas numéricos desarrollados en los capítulos anteriores para resolver las ecuaciones del calor, de onda y de Laplace. Se presentan tanto las versiones clásicas en CPU, construidas de forma directa a partir de las fórmulas explícitas, como sus adaptaciones a la arquitectura paralela de GPU mediante CUDA y la librería PyCUDA. Estas implementaciones servirán de base para el estudio comparativo de eficiencia que se abordará en el siguiente capítulo.

## 7.1. Flujo general de los algoritmos en CPU

Todos los métodos implementados en CPU[1] siguen una estructura común, compuesta por los siguientes pasos:

1. **Cálculo de parámetros.**
   A partir de los datos de entrada (condiciones iniciales y de contorno, y dimensiones del dominio) determinamos varios parámetros necesarios para el algoritmo, como por ejemplo

   $$n_x,\ n_y,\ n_t, \quad \Delta x,\ \Delta y,\ \Delta t, \quad \text{o} \quad \lambda,\ \lambda_x,\ \lambda_y,$$

   según corresponda al esquema explícito de calor, onda o al método de Jacobi.

2. **Reserva e inicialización de memoria.**
   Asignamos en memoria principal un arreglo de dos (o tres) dimensiones que almacenará la solución completa. Seguidamente, volcamos en él los valores iniciales y de contorno. En el caso de la ecuación de Laplace, se utilizan dos matrices (denotadas `u_old` y `u_new`) que permiten alternar entre los valores de la iteración anterior y los de la actual.

[1] El código puede encontrarse en el repositorio de Github `https://github.com/NotNoe/TFG-Mates`.

3. **Bucle de iteración.**

   - **Ecuaciones evolutivas (calor y onda).** Recorremos temporalmente los $n_t$ pasos y, para cada instancia $n$, actualizamos todos los nodos de la malla según la fórmula de diferencias finitas explícitas correspondiente.
   - **Ecuación de Laplace.** Aplicamos iterativamente la fórmula recursiva (5.5) hasta que se cumple alguna de las condiciones de parada:

     (I) El número de iteraciones alcanza el máximo .

     (II) La norma infinita del error relativo $\text{máx}_{i,j}\left|u_{i,j}^{(k+1)} - u_{i,j}^{(k)}\right|$ cae por debajo de la tolerancia **tol**.

4. **Recogida de resultados.**
   Concluido el bucle de iteración, la solución numérica queda contenida en el arreglo activo (o en `u_new` para Laplace) y se retorna directamente como salida de la función.

## 7.2. Flujo general de los algoritmos en GPU

Todos los métodos implementados en GPU[2] siguen esencialmente la misma secuencia de pasos que en CPU, con la diferencia de que aprovechan la concurrencia entre cómputo y transferencia mediante flujos y eventos:

1. **Cálculo de parámetros.**
   Igual que en CPU, calculamos varios parámetros necesarios para el esquema correspondiente.

2. **Reserva de memoria.**

   *a*) **Anfitrión:** Asignamos los arreglos completos en memoria fija[3].

   *b*) **Dispositivo:** Dado que la memoria disponible en la GPU es inferior a la de la memoria principal, sólo reservamos $p$ búferes ($p = 2$ para calor y Laplace, $p = 3$ para onda). Cada iteración $n$ escribe en el búfer `buf`$[n \bmod p]$, de modo que se sobrescriben únicamente datos ya usados.

   En el caso de Laplace además reservamos dos búferes (en GPU y en CPU) para alamcenar el valor $\delta^{(n)}$.

3. **Configuración de flujos y eventos.**
   Creamos dos flujos:

   - `kernel_stream` para encolar los `núcleos`.
   - `mem_stream` para las transferencias de memoria.

[2] El código puede encontrarse en el repositorio de Github `https://github.com/NotNoe/TFG-Mates`.

[3] La memoria fija mejora la tasa de transferencia de memoria entre anfitrión y dispositivo.

Para evitar condiciones de carrera (acceso concurrente de lectura y escritura sobre un mismo búfer) definimos $2p$ eventos: $\{$`ev_ke`$[i]$, `ev_mem`$[i] \mid i = 0, \ldots, p-1\}$, donde `ev_ke`$[i]$ señala la finalización del núcleo que escribe en el búfer $i$ y `ev_mem`$[i]$ fin de copia del mismo búfer.

4. **Bucle de iteración.**

   - **Evolutivos (calor y onda).**
     En cada paso $n$:

     *a*) Si $n \geq p$, hacer que `kernel_stream` espere a *ev_mem*$[n \bmod p]$, garantizando que el búfer está libre.

     *b*) En `kernel_stream` lanzar el `núcleo` que computa la nueva capa en `buf`$[n \bmod p]$, y registrar `ev_ke`$[n \bmod p]$.

     *c*) En `mem_stream` esperar a `ev_ke`$[n \bmod p]$, luego encolar la copia de resultados hacia el anfitrión, y finalmente registrar `ev_mem`$[n \bmod p]$.

   - **Laplace.**
     En cada paso $k$:

     *a*) Ponemos a cero el búfer `diff_gpu`$[n \bmod p]$.

     *b*) Si $n \geq p$, hacer que `kernel_stream` espere a `ev_mem`$[n \bmod p]$, garantizando que el búfer donde almacenamos diff está libre.

     *c*) En `kernel_stream` lanzamos el núcleo que computa la nueva capa en `buf`$[n \bmod p]$, y registramos `ev_ke`$[n \bmod p]$. Este `núcleo`, además, calcula el valor $\delta^{(n)}$ y lo deja en la memoria de la GPU.

     *d*) Esperamos al evento `ev_ke`$[n \bmod p]$ en `mem_stream` y copiamos el valor $\delta^{(k)}$ en `diff_bits`$[n \bmod p]$

     *e*) Si $n > 0$:

       1) En el anfitrión, esperamos a que ocurra el evento `diff_bits`$[(n-1) \bmod p]$, que nos indica que el búfer con el valor $\delta^{(n-1)}$ está listo.

       2) Convertimos este valor a un float, y si se cumple la condición de tolerancia, salimos del bucle.

5. **Recogida de resultados.**

   - **Evolutivos (calor y onda).**
     Al completar todas las iteraciones, sincronizamos con los flujos mediante `synchronize()`. De este modo garantizamos que tanto las copias como los núcleos han finalizado, tras lo cual se recupera la solución completa almacenada en el búfer activo.

   - **Laplace.**
     En este caso, no hemos llegado a sacar el resultado de la GPU durante el bucle, por lo que después de sincronizar tendremos que copiar el resultado de la GPU a la CPU. Tras ello, liberamos recursos y devolvemos la solución.

## 7.3. Validación

Todos los esquemas numéricos desarrollados en este capítulo han sido validados experimentalmente comparando la solución aproximada obtenida con una solución exacta conocida en al menos un caso concreto. En particular, se ha construido, para cada tipo de ecuación, un ejemplo en el que la solución analítica se conoce explícitamente y satisface las condiciones del problema. Se considera que la implementación numérica es válida cuando el error máximo entre la solución numérica y la exacta es inferior a $10^{-2}$.

Capítulo 8

# Pruebas y resultados

**Resumen:** En este capítulo describimos las pruebas realizadas para comparar los algoritmos en CPU y GPU, presentamos los tiempos de cómputo medidos para cada caso de prueba, y el aumento de velocidad en cada caso.

## 8.1. Diseño del banco de pruebas

Para cada uno de los cinco métodos numéricos implementados:

1. Se fija un problema de valor inicial y/o de contorno. En concreto:

   *a*) **Ecuación del calor 1D**
   - $(a, b) = (0, 1)$.
   - $T = 0{,}1$ [1].
   - $f(x) = \sin(\pi x)$.
   - $g(t) = h(t) = 0$.

   *b*) **Ecuación del calor 2D**
   - $(a, b) = (c, d) = (0, 1)$.
   - $T = 0{,}1$ [1].
   - $f(x, y) = \sin(\pi x)\sin(\pi y)$.
   - $g(t) = h(t) = 0$.

   *c*) **Ecuación de onda 1D**
   - $(a, b) = (0, 1)$.
   - $T = 1$.
   - $f(x) = \sin(\pi x)$.

[1] Esta elección se debe a la naturaleza de la condición de convergencia del método, que obliga a que el tamaño de paso la malla temporal crezca cuadráticamente con el tamaño de paso de la malla espacial. Al elegir un $T$ más pequeño, contrarrestamos un poco este crecimiento.

   - $g(x) = 0$.

   d) **Ecuación de onda 2D**
   - $(a, b) = (c, d) = (0, 1)$.
   - $T = 1$.
   - $f(x, y) = \sin(\pi x)\sin(\pi y)$.
   - $g(x) = 0$.

   e) **Ecuación de Laplace**
   - $(a, b) = (c, d) = (0, 1)$.
   - $g(x, y) = \begin{cases} 0, & x = 0, 1 \text{ o } y = 0, \\ \sin(\pi x)\sinh(\pi), & y = 1. \end{cases}$

2. Se fijan seis pruebas con tamaños de malla crecientes de manera que se cumplan las condiciones de convergencia (los tamaños concretos se recogen en la Tabla 8.1). Las condiciones concretas de convergencia son:

   a) **Ecuación del calor 1D**: $\lambda \leq \frac{1}{2}$.

   b) **Ecuación del calor 2D**: $\lambda_x + \lambda_y \leq \frac{1}{2}$.

   c) **Ecuación de onda 1D**: $\lambda \leq 1$.

   d) **Ecuación de onda 2D**: $\lambda_x + \lambda_y \leq 1$.

   e) **Ecuación de Laplace**: El método para esta ecuación siempre converge.

3. Se resuelve el problema de valor inicial y/o contorno con el algoritmo implementado en CPU y en GPU, midiendo los tiempos de cada uno de estos.

## 8.2. Resultados

En la Figura 8.1 y la Figura 8.2 podemos ver los tiempos obtenidos y el aceleración que introduce la GPU respectivamente (los valores exactos se presentan en la Tabla 8.2). La aceleración de un problema concreto la definimos como el tiempo que tarda el algoritmo en CPU en resolver ese problema entre el tiempo que tarda el algoritmo en GPU. Además, entendemos el número de nodos de la malla de un problema como el número total de nodos de la malla espacial, es decir, $n_x$ para problemas de una dimensión espacial y $n_x \cdot n_y$ para problemas de dos dimensiones espaciales.

Como cabía esperar, el uso de GPU permite acelerar la ejecución de los algoritmos entre uno y dos órdenes de magnitud.

## 8.3. Análisis de los resultados

Analizando la Figura 8.1 y la Figura 8.2 pueden extraerse las siguientes conclusiones:

| Ecuación | Prueba | $n_x$ | $n_y$ | $n_t$ | Cond. Conv. |
|---|---|---|---|---|---|
| Heat 1D | 1 | 100 | – | 2179 | 0,45 |
| | 2 | 180 | – | 7121 | 0,45 |
| | 3 | 330 | – | 24054 | 0,45 |
| | 4 | 600 | – | 79734 | 0,45 |
| | 5 | 900 | – | 178200 | 0,4535 |
| | 6 | 1100 | – | 268401 | 0,45 |
| Heat 2D | 1 | 50 | 50 | 1068 | $0{,}225 + 0{,}225$ |
| | 2 | 75 | 75 | 2434 | $0{,}2251 + 0{,}2251$ |
| | 3 | 100 | 100 | 4357 | $0{,}225 + 0{,}225$ |
| | 4 | 125 | 125 | 6875 | $0{,}2237 + 0{,}2237$ |
| | 5 | 150 | 150 | 9868 | $0{,}225 + 0{,}225$ |
| | 6 | 210 | 210 | 19414 | $0{,}225 + 0{,}225$ |
| Wave 1D | 1 | 100 | – | 111 | 0,9 |
| | 2 | 250 | – | 277 | 0,9022 |
| | 3 | 630 | – | 699 | 0,9011 |
| | 4 | 1580 | – | 1755 | 0,9002 |
| | 5 | 4000 | – | 4444 | 0,9001 |
| | 6 | 10000 | – | 11111 | 0,9 |
| Wave 2D | 1 | 50 | 50 | 77 | $0{,}4157 + 0{,}4157$ |
| | 2 | 75 | 75 | 117 | $0{,}4070 + 0{,}4070$ |
| | 3 | 115 | 115 | 180 | $0{,}4056 + 0{,}4056$ |
| | 4 | 175 | 175 | 274 | $0{,}4062 + 0{,}4062$ |
| | 5 | 265 | 265 | 415 | $0{,}4066 + 0{,}4066$ |
| | 6 | 400 | 400 | 627 | $0{,}4063 + 0{,}4063$ |
| Laplace | 1 | 20 | 20 | – | – |
| | 2 | 30 | 30 | – | – |
| | 3 | 40 | 40 | – | – |
| | 4 | 60 | 60 | – | – |
| | 5 | 85 | 85 | – | – |
| | 6 | 120 | 120 | – | – |

Tabla 8.1: Parámetros de entrada y condición de convergencia (es decir, los valores de $\lambda$ o $\lambda_x$ y $\lambda_y$, dependiendo del problema) para cada prueba realizada

- **Ventaja de la GPU respecto a la CPU**
  Excepto en los casos con mallas más pequeñas, los tiempos de ejecución en GPU resultan claramente inferiores a los tiempos obtenidos en CPU. Como comentamos en secciones anteriores, esto era previsible, dado que las operaciones de reserva y copia de memoria en la GPU introducen un coste fijo que, en mallas muy pequeñas, no se ve compensado por el aumento de velocidad de cálculo en GPU.

- **Monotonía creciente del aceleración**
  Al usar la GPU no solo obtenemos algoritmos más rápidos (para mallas no muy pequeñas), sino que observamos que la mejora introducida por la GPU crece junto con el número de nodos de la malla. Esto también es esperable, debido a que a mayor tamaño de malla, mayor es el número de operaciones que pueden ejecutarse en paralelo en la GPU.

- **Disminución del crecimiento de la aceleración para mallas muy grandes**

| Ecuación | Prueba | $t_{cpu}$ | $t_{gpu}$ | aceleración ($t_{cpu}/t_{gpu}$) |
|---|---|---|---|---|
| Heat 1D | 1 | 0.2728 | 0.1268 | 2.15 |
| | 2 | 1.147 | 0.3216 | 4.578 |
| | 3 | 8.772 | 0.9397 | 9.335 |
| | 4 | 56.7 | 3.08 | 18.40 |
| | 5 | 780.7 | 6.695 | 26.99 |
| | 6 | 332.3 | 10.55 | 31.49 |
| Heat 2D | 1 | 4.849 | 7.784 | 0.623 |
| | 2 | 23.86 | 0.2509 | 95.07 |
| | 3 | 78.64 | 0.4853 | 162 |
| | 4 | 189.3 | 0.9736 | 194.5 |
| | 5 | 398.3 | 1.609 | 247.4 |
| | 6 | 1574 | 4.876 | 322.8 |
| Wave 1D | 1 | 0.04279 | 1.039 | 0.0411 |
| | 2 | 0.2358 | 0.05416 | 4.355 |
| | 3 | 1.376 | 0.05832 | 23.6 |
| | 4 | 7.941 | 0.1709 | 46.45 |
| | 5 | 53.41 | 0.3901 | 136.9 |
| | 6 | 324.7 | 1.651 | 196.6 |
| Wave 2D | 1 | 0.4137 | 0.9467 | 0.437 |
| | 2 | 1.253 | 0.03904 | 32.09 |
| | 3 | 4.884 | 0.04185 | 116.6 |
| | 4 | 19.09 | 0.06846 | 278.8 |
| | 5 | 61.54 | 0.1124 | 547.2 |
| | 6 | 206.4 | 0.3003 | 687.5 |
| Laplace | 1 | 0.3224 | 1.0719 | 0.3008 |
| | 2 | 1.639 | 0.143 | 11.45 |
| | 3 | 5.713 | 0.1853 | 30.82 |
| | 4 | 27.54 | 0.3546 | 77.65 |
| | 5 | 124.3 | 0.7171 | 169.2 |
| | 6 | 551.3 | 1.247 | 441.8 |

Tabla 8.2: Tiempos y aceleración de prueba realizada

Puede apreciarse en la Figura 8.2 que el aceleración crece rápidamente, pero en algunas de las gráficas, la tasa de crecimiento disminuye en los tamaños de malla más grandes. Esto genera tres zonas distintas en el rendimiento relativo entre CPU y GPU:

1. **Zona inicial**: Para mallas muy pequeñas, el coste de incialización de la GPU (la compilación del núcleo, la reserva de memoria y las transferencias iniciales de los datos) no se compensa con el trabajo en paralelo que realiza la GPU. En esta zona, la CPU es igual o incluso más eficiente que la GPU.
2. **Zona de rendimiento óptimo**: Al aumentar el tamaño de la malla, el número de operaciones que realiza la GPU en paralelo se incrementa sustancialmente. La GPU puede realizar una gran cantidad de cálculos en paralelo, lo que produce un aumento rápido del aceleración.
3. **Zona de saturación**: Para tamaños de malla muy grandes, el cálculo deja de constituir el principal factor limitante. El rendimiento se ve limitado por el ancho de banda de la memoria global y el coste de las transferencias

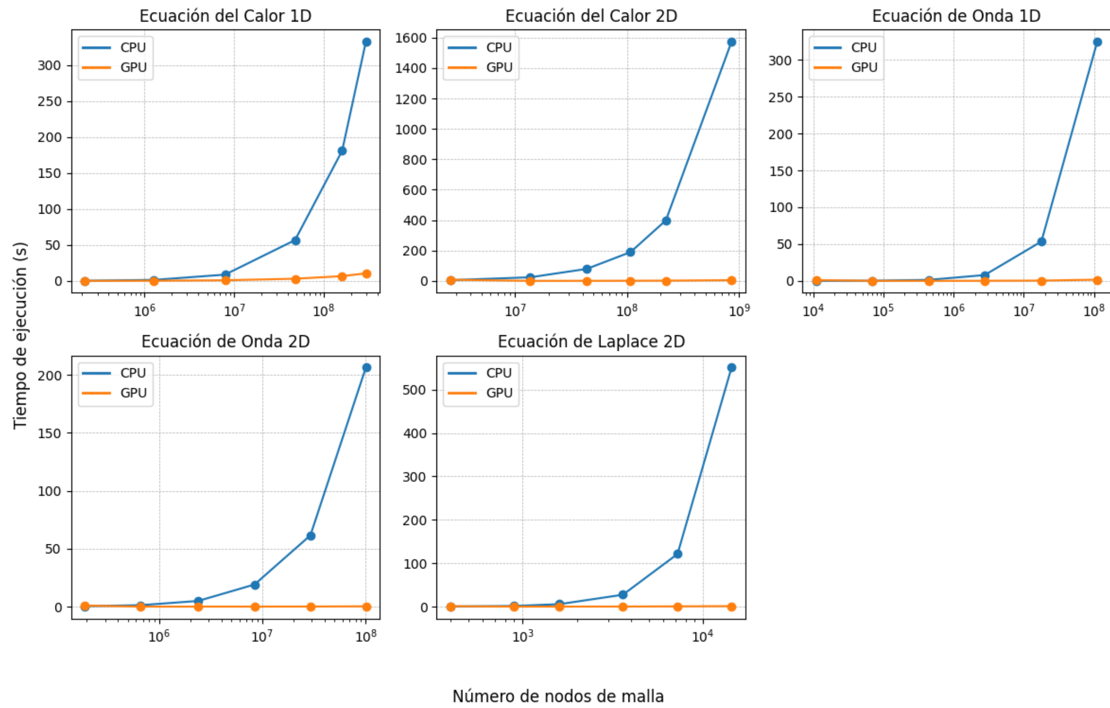


Figura 8.1: Tiempos de ejecución (eje Y) en función del tamaño de la malla (eje X).

entre el dispositivo y el anfitrión. La GPU sigue siendo más rápida, pero el aceleración crece de forma más lenta, ya que la velocidad del acceso a memoria de la GPU y de la CPU no es muy diferente.

Este efecto se ve ampliado por el hecho de que la mayoría de algoritmos implementados copian al anfitrión la parte de la solución calculada en cada paso temporal. En caso de haberse optado por devolver únicamente la solución final $u(\cdot, T)$, el volumen de datos transferidos habría sido considerablemente menor y la zona de saturación aparecería para mallas significativamente más grandes.

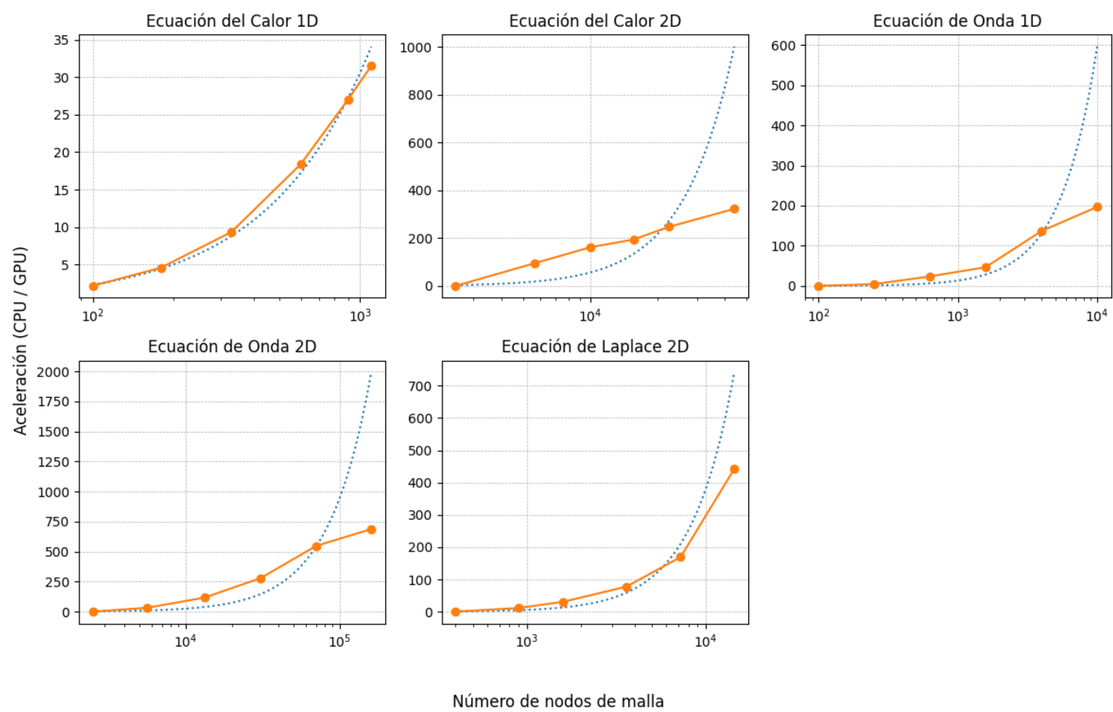


Figura 8.2: Aceleración (eje Y) en función del tamaño de la malla (eje X).

Capítulo 9

# Conclusiones y Trabajo Futuro

## 9.1. Conclusiones

Gracias a la comparativa de los resultados obtenidos a lo largo del trabajo, hemos podido obtener las siguientes conclusiones:

- Los métodos numéricos basados en diferencias finitas resultan adecuados para la obtención de aproximaciones numéricas de EDPs, en especial en casos evolutivos, donde se puede obtener en muchos casos un método explícito.

- Estos métodos tienden a ser más precisos cuando las derivadas implicadas son de orden mayor, ya que el error de truncamiento introducido por las diferencias finitas es asintóticamente menor cuánto mayor es el orden de derivación.

- La mejora de rendimiento obtenida mediante el uso de la GPU resulta evidente, salvo en casos con mallas de tamaño muy reducido. No obstante, esta mejora acaba viéndose mermada para mallas muy grandes debido al cuello de botella que genera el acceso y copia de memoria.

## 9.2. Trabajo futuro

Existen múltiples líneas en las que este trabajo podría ampliarse o generalizarse. A continuación se enumeran algunas de las más relevantes:

- **Implementar versiones ligeras de los algoritmos**
  Desarrollar versiones alternativas de los algoritmos implementados que, en lugar de almacenar todas las iteraciones intermedias, devuelvan únicamente la solución final $u(\cdot, T)$. Aunque en la mayoría de contextos es necesario disponer de la evolución completa de la solución, contar con versiones optimizadas que omitan dicho almacenamiento permite reducir significativamente las transferencias de memoria y acelerar los algoritmos en los casos en los que solo se requiera el estado final.

- **Análisis en profundidad de la malla espacial y temporal**
  En las pruebas realizadas durante este trabajo, ambos parámetros crecían de forma correlada para satisfacer la condición de convergencia de cada método. Sin embargo, en GPU:

  - El incremento de la *malla espacial* aumenta simultáneamente el grado de paralelismo y el tamaño de los datos trasladados entre CPU y GPU.
  - El aumento de la *malla temporal* extiende el número de iteraciones (lanzamientos de núcleos y copias), sin cambiar el tamaño de cada dato.

  Resultaría de interés cuantificar, o bien por separado, o bien en gráficas tridimensionales, el impacto de cada tamaño sobre el aceleración y al patrón de cuellos de botella.

- **Optimización avanzada de la GPU**
  Profundizar en detalles de la arquitectura CUDA (memoria compartida, coalescencia de accesos, etc.) podría aumentar aún más el rendimiento, especialmente en problemas 2D de gran tamaño.

- **Estudio avanzado de métricas**
  Emplear herramientas de perfilado para desglosar tiempos de ejecución, transferencia de datos y otros servicios de la GPU nos permitiría estudiar con precisión los cuellos de botella y guiaría optimizaciones específicas.

## 9.3. Principales contribuciones

Las principales contribuciones al campo del análisis numérico que hemos aportado son las siguientes:

- Implementación de cinco esquemas de diferencias finitas para las ecuaciones del calor (1D y 2D), de onda (1D y 2D) y de Laplace (2D).

- Desarrollo de versiones paralelas en GPU mediante CUDA (a través de PyCUDA), incluyendo la gestión de flujos, eventos y búferes, sin depender de bibliotecas de alto nivel.

- Creación de un banco de pruebas sistemático que permite medir y comparar tiempos de CPU y GPU para unos tamaños de malla cualesquiera.

- Análisis cualitativo de los resultados experimentales, identificando las distintas zonas de rendimiento y los factores que influyen en el aceleración.

# Bibliografía


[1] *CUDA C++ Programming Guide.*

[2] G. Araujo, D. Griebler, D. A. Rockenbach, M. Danelutto, and L. G. Fernandes. Nas parallel benchmarks with cuda and beyond. *Software: Practice and Experience*, 53(1):53–80, 2023.

[3] H. Brezis and J. R. Esteban. *Análisis funcional, teoria y aplicaciones.* Alianza, 1984.

[4] J. R. Cannon. *The One-dimensional heat equation.* Reading, Massachusetts [etc] : Addison-Wesley, 1984.

[5] J. A. Infante del Río and J. M. Rey Cabezas. *Métodos Numéricos: Teoría, Problemas y Prácticas con MATLAB.* Pirámide, Madrid, 6 edition, 2022.

[6] E. Isaacson and H. B. Keller. *Analysis of Numerical Methods.* John Wiley and Sons, 1966.

[7] A. Kloeckner. Pycuda documentation. `https://documen.tician.de/pycuda/`, 2025.

[8] E. Weisstein. d´Alembert's Solution. – MathWorld. A Wolfram Web Resource, 2024. Disponible online en: `https://mathworld.wolfram.com/dAlembertsSolution.html`, Último acceso: 28/08/2024.

Apéndice A

# Código

En este apéndice se incluye el código completo de todos los algoritmos desarrollados en el trabajo, clasificados por ecuación y tipo de implementación (CPU o GPU). Se han omitido elementos auxiliares como generadores de gráficos o visualizaciones, que no son relevantes para el desarrollo teórico ni la validación.

## A.1. Código auxiliar

```
import pycuda.autoinit
from pycuda.compiler import SourceModule
import os, sys

CUDA_FOLDER = "./CUDA"

def search_dir(path, files):
    files += [f for f in os.listdir(path) if os.path.isfile(path + os.sep + f) and f[-3:]==".cu"]
    for d in [path + os.sep + d for d in os.listdir(path) if os.path.isdir(path+os.sep+d) and d != "__pycache__"]:
        search_dir(d, files)


def init(files=None):
    """
   Generates SourceModule with the code and autoinits pycuda.

   :param [str] files: The files from which the code is generated (if none, all the files in the directory (recursive) are considered)
   :return: The SourceModule
   """
    if files == None:
        files = []
        search_dir(os.getcwd(), files)
    code = ""
    for file in files:
        code += open(os.path.join(CUDA_FOLDER, file), 'r').read() + '\n'
```

```
    return SourceModule(code)




if __name__ == "__main__":
    files = []
    search_dir(os.getcwd(), files)
    print(files)
```

Código A.1: Módulo para generar `SourceModule` a partir de los archivos

## A.2. Ejemplos introductorios

### A.2.1. Hola Mundo

```
import utils.generateMod as generateMod
import pycuda.driver as cuda
import numpy as np


mod = generateMod.init(["demos/HelloWorld.cu"])
hello_world = mod.get_function("hello_world")
a = [1.0, 2.0]
a = np.asarray(a).astype("float32") #Esta es la memoria que
    gestionaremos en host, arrays de numpy
a_gpu = cuda.mem_alloc(a.nbytes) #Reservamos la memoria en GPU
cuda.memcpy_htod(a_gpu, a)
hello_world(a_gpu, block = (2,1,1))
```

Código A.2: Hola Mundo (Código Python)

```

__global__ void hello_world(float* a){
    int id = threadIdx.x;
    printf("Hello World, my id is %i and my number is \"%f\".\n",
    id, a[id]);
}
```

Código A.3: Hola Mundo (Código CUDA)

```
Hello World, my id is 0 and my number is "1.000000".
Hello World, my id is 1 and my number is "2.000000".
```

Código A.4: Hola Mundo (Salida)

### A.2.2. Suma de vectores

```
import numpy as np
import utils.generateMod as generateMod
import pycuda.driver as cuda
from timeit import default_timer as timer

mod = generateMod.init(["demos/vectorAdd.cu"])
vectorAdd = mod.get_function("vectorAdd")
```

```
8 def add_random_vects(n, out):
9     #Tiempo en CPU (Contando reservar memoria)
10    start = timer()
11    a = np.random.randn(n).astype(np.float32)
12    b = np.random.rand(n).astype(np.float32)
13    c2 = np.zeros(n).astype(np.float32)
14    np.add(a,b,out=c2)
15    CPUt = timer()-start
16    #Tiempo en GPU (Contando reservar y copiar los datos al
    dispositivo)
17    start = timer()
18    c1 = np.zeros(n).astype(np.float32)
19    a_gpu = cuda.mem_alloc(a.nbytes)
20    b_gpu = cuda.mem_alloc(b.nbytes)
21    c_gpu = cuda.mem_alloc(c1.nbytes)
22    n_gpu = cuda.mem_alloc(np.int32(n).nbytes)
23    cuda.memcpy_htod(a_gpu, a)
24    cuda.memcpy_htod(b_gpu, b)
25    b = n if n < 1024 else 1024 #el bloque va a ser de 1024 (
    siempre que n > 1024)
26    g = n // 1024 + 1 #Tantos grids como haga falta
27    vectorAdd(a_gpu, b_gpu, c_gpu, np.int32(n), grid=(g,1,1), block
    = (b,1,1))
28    cuda.memcpy_dtoh(c1, c_gpu)
29    GPUt = timer() - start
30
31    if (c1 == c2).all():
32        text = f"""------------------N = {n}------------------
33        Los resultados coinciden:
34        Tiempo en GPU {GPUt}
35        Tiempo en CPU: {CPUt}
36        La GPU es un {round(CPUt/GPUt*100,1)}% mas rapida"""
37        print(text, file=out)
38        print(text)
39    else:
40        print("Ha habido un error en el computo en GPU.\n", file=
    out)
41        print("Ha habido un error en el computo en GPU.\n")
```

Código A.5: Suma de vectores (Código Python)

```
1 __global__ void vectorAdd(float* a, float* b, float* c, int n){
2     int i = blockIdx.x * 1024 + threadIdx.x; //Calculamos el indice
      teniendo en cuenta que cada bloque tiene 1024 elementos
3     if(i<n){
4         c[i] = a[i] + b[i];
5     }
6 }
```

Código A.6: Suma de vectores (Código CUDA)

```
1 ------------------N = 100------------------
2         Los resultados coinciden:
3         Tiempo en GPU 0.00045944999874336645
4         Tiempo en CPU: 0.004859157997998409
5         La GPU es un 1057.6% mas rapida
```

```
-------------------N = 10000-------------------
        Los resultados coinciden:
        Tiempo en GPU 0.00018776399883790873
        Tiempo en CPU: 0.00044756100032827817
        La GPU es un 238.4% mas rapida
-------------------N = 1000000-------------------
        Los resultados coinciden:
        Tiempo en GPU 0.0065122929991048295
        Tiempo en CPU: 0.04882143099894165
        La GPU es un 749.7% mas rapida
-------------------N = 100000000-------------------
        Los resultados coinciden:
        Tiempo en GPU 0.5896225029973721
        Tiempo en CPU: 4.510856007000257
        La GPU es un 765.0% mas rapida
```

Código A.7: Suma de vectores (Salida)

# Apéndice B

# Hardware y software utilizado

## B.1. Hardware

Todas las pruebas se han realizado en *bujaruelo*, una máquina del departamento de arquitectura de computadores y automática de la facultad de informática de la universidad.

- **Procesador**: Intel(R) Xeon(R) CPU E5-2695 v3 @2.3Gz
- **GPUs**: El equipo tiene en total cuatro gráficas, dos de cada uno de los modelos siguientes:
    - NVIDIA GeForce GTX 1080
    - NVIDIA GeForce GTX 980
- **Discos**: El equipo cuenta con dos discos de 1TB, además del disco donde está montado el sistema operativo, de 74.5GB.

## B.2. Librerías utilizadas

El proyecto ha sido desarrollado en **Python 3.11.10**, utilizando las siguientes librerías:

| **Librería** | **Versión** | **Función principal** |
|---|---|---|
| pycuda | 2024.1.2 | Compilación y ejecución de núcleos CUDA desde Python |
| numpy | 2.1.3 | Cálculo numérico y manipulación de vectores |
| pandas | 2.2.3 | Procesamiento y exportación de resultados del banco de pruebas |
| tqdm | 4.67.1 | Visualización del progreso durante la ejecución de pruebas |

Además, para compilar y ejecutar los programas CUDA en la GPU se utilizó el **CUDA Toolkit 12.4.1**, necesario para el correcto funcionamiento de PyCUDA.

## Lista de acrónimos

**EDP**........................ Ecuación en Derivadas Parciales

**CPU**..... Unidad Central de Procesamiento (*Central Proccesing Unit*)

**CUDA**........................ *Compute Unified Device Architecture*

**PyCUDA**........................ *Python for CUDA*