
Resolución de ecuaciones en GPU
Solving equations in GPU



Trabajo de Fin de Grado
Curso 2023–2024

Autor

Noelia Barranco Godoy

Director

Ana María Carpio Rodríguez

Doble grado en Matemáticas e Informática

Facultad de Ciencias Matemáticas

Universidad Complutense de Madrid

Resolución de ecuaciones en GPU Solving equations in GPU

Trabajo de Fin de Grado en Matemáticas

Autor

Noelia Barranco Godoy

Director

Ana María Carpio Rodríguez

Convocatoria: Septiembre 2024

Doble grado en Matemáticas e Informática

Facultad de Ciencias Matemáticas

Universidad Complutense de Madrid

23 de septiembre de 2024

Resumen

Resolución de ecuaciones en GPU

En este trabajo pretendemos comparar las diferencias respecto a eficacia entre resolver ecuaciones diferenciales mediante algoritmos tradicionales y algoritmos implementados sobre GPU's.

En particular, implementaremos el código utilizando el lenguaje de programación Python, y la api de Nvidia CUDA

Palabras clave

ecuaciones, diferencias finitas, computación, gpu, ecuación del calor, ecuación de ondas, ecuación de Laplace

Abstract

Solving equations in GPU

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Keywords

10 keywords max., separated by commas.

Capítulo 1

Introducción

RESUMEN: En este capítulo pretendemos introducir los objetivos de este trabajo

1.1. Motivación

Desde el inicio de la computación, se han desarrollado métodos numéricos para aproximar soluciones de ecuaciones que no podemos resolver de manera analítica (o cuya solución exacta no se conoce). Con el auge de la computación en GPU, que permite computar los datos en paralelo, se pueden implementar estos mismos métodos de formas más eficientes para lograr mejores resultados.

1.2. Objetivos

En este trabajo pretendemos estudiar la implementación de métodos de diferencias finitas en la GPU y su mejora de eficiencia en las ecuaciones de *Laplace* (en dos dimensiones), del calor (en una y dos dimensiones) y de ondas (en una y dos dimensiones), que son las siguientes:

Laplace:

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

Calor (1D):

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

Calor (2D):

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

Ondas (1D):

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

Ondas (2D):

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

1.3. Nociones generales

Primero de todo necesitaremos hacer un estudio matemático sobre las ecuaciones diferenciales que trataremos. Todos los algoritmos que vamos a hacer están basados

en el método de las diferencias finitas, que consiste en hacer una aproximación de las derivadas por un cociente incremental en puntos cercanos. Para hacer esto, necesitaremos definir sobre todos los problemas una malla discreta de puntos, y serán en estos donde hallemos soluciones aproximadas de las soluciones.

1.3.1. Mallas

Cuando definamos y trabajemos sobre algoritmos numéricos para aproximar las soluciones de los problemas necesitaremos discretizar el dominio, pues necesitamos trabajar con una cantidad de puntos finita para que un ordenador pueda implementar el algoritmo.

Para conseguir esto, definiremos mallas, que son subconjuntos (potencialmente infinitos pero numerables) del dominio (que denotaremos como R) del problema en cuestión. Luego aproximaremos la solución exacta del problema en una cantidad finita de puntos de la malla.

Definición 1.3.1 (Malla bidimensional). *Dado una ecuación diferencial con dominio $R \subseteq \mathbb{R}^2$, sean a, b, c y d números reales tales que $a \leq b$ y $c \leq d$, y sean n_1 y n_2 dos números naturales mayores que 1, definimos la malla bidimensional (o simplemente malla) como*

$$M^2(a, b, c, d, n_1, n_2) := \{(a + i\Delta_1, c + j\Delta_2) \mid i, j \in \mathbb{Z}\} \cap R,$$

siendo $\Delta_1 := \frac{b-a}{n_1-1}$ y $\Delta_2 := \frac{d-c}{n_2-1}$.

Definición 1.3.2 (Malla tridimensional). *Dado una ecuación diferencial con dominio $R \subseteq \mathbb{R}^3$, sean a, b, c, d, e y f números reales tales que $a \leq b$ y $c \leq d$, y sean n_1, n_2 y n_3 tres números naturales mayores que 1, definimos la malla tridimensional (o simplemente malla) como*

$$M^3(a, b, c, d, e, f, n_1, n_2, n_3) := \{(a + i\Delta_1, c + j\Delta_2, e + k\Delta_3) \mid i, j, k \in \mathbb{Z}\} \cap R,$$

siendo $\Delta_1 := \frac{b-a}{n_1-1}$, $\Delta_2 := \frac{d-c}{n_2-1}$ y $\Delta_3 := \frac{f-e}{n_3-1}$.

Normalmente los parámetros de la construcción de la malla están fijos en cada problema, por lo que normalmente nos referiremos a la malla (ya sea bidimensional o tridimensional) de cada problema simplemente como M , omitiendo los parámetros y la dimensión.

Además, cada una de las dimensiones de la malla van a corresponder siempre a x, y o t , por lo que si la dimensión i del problema es x , diremos Δx y n_x en lugar de Δ_i y n_i para que la notación sea más intuitiva.

Supongamos que tenemos un problema con un dominio R y una malla M (ya sea bidimensional o tridimensional), sin pérdida de la generalidad podemos decir que x está asociado a la primera dimensión, y a la segunda y t a la tercera. Denotaremos pues $x_i := a + i\Delta x$, $y_j := c + j\Delta y$ y $t_k := e + k\Delta t$. Es importante apreciar que con esta notación, los puntos de la malla serán todos de la forma $(x_i, y_j, t_k)^1$ y que

¹Nótese que todos los puntos de la malla serán de esa forma para alguna terna $(i, j, k) \in \mathbb{Z}^3$, pero el recíproco no es necesariamente cierto. Dependiendo del dominio R , puede pasar que haya ternas $(i, j, k) \in \mathbb{Z}^3$ tales que $(x_i, y_j, t_k) \notin R$ y por tanto el punto no pertenezca a la malla.

se cumple la relación $x_{n_x-1} = a + (n_x - 1) * \Delta x = a + (n_x - 1) * \frac{b-a}{n_x-1} = b$ para cualquiera de las tres dimensiones (cambiando x por y o t y a, b por c, d o e, f).

Una última observación necesaria es que, si el dominio R es convexo (lo cuál se cumplirá en todos los problemas en los que trabajamos), todos los puntos (x_i, y_j, t_k) tales que $0 \leq i < n_x$, $0 \leq j < n_y$ y $0 \leq k < n_t$ estarán dentro de la malla.

Significado de las mallas

La idea intuitiva tras la construcción de las mallas es que el objetivo de los algoritmos será aproximar el valor de las funciones sobre una región del espacio acotada durante un tiempo concreto T . Como necesitamos trabajar con una cantidad finita de puntos, hacemos mallas con una densidad concreta (que afectará al rendimiento y la precisión).

Los puntos que tienen todos sus índices dentro de los límites $[0, n_i)$ son los puntos que están dentro de esta superficie finita (que tiene forma de rectángulo en dos o tres dimensiones) que queremos calcular, pero puede ser que necesitemos aproximar la función en otros puntos para poder calcularlo.

1.3.2. Notación

Haremos ahora algunas definiciones sobre la notación con el objetivo de hacer las demostraciones futuras más legibles.

- Sea f una función cualquiera definida en R , denotaremos $f_{i,j,k} := f(x_i, y_j, t_k)$.
- La solución del problema que estemos trabajando la denotaremos por u .
- La aproximación de la solución, que es una función definida solo sobre la malla M , la llamaremos U .
- Por último, definimos las diferencias finitas progresiva, regresiva y centrada sobre la variable x (o cualquiera de las otras variables de manera análoga) de una función f definida sobre R como:

$$f^x(x, y, t) := \frac{f(x + \Delta x, y, t) - f(x, y, t)}{\Delta x} \Rightarrow f_{i,j,k}^x = \frac{f_{i+1,j,k} - f_{i,j,k}}{\Delta x}, \quad (1.1)$$

$$f^{\bar{x}}(x, y, t) := \frac{f(x, y, t) - f(x - \Delta x, y, t)}{\Delta x} \Rightarrow f_{i,j,k}^{\bar{x}} = \frac{f_{i,j,k} - f_{i-1,j,k}}{\Delta x}, \quad (1.2)$$

$$f^{\hat{x}}(x, y, t) := \frac{1}{2}[f^x(x, y, t) + f^{\bar{x}}(x, y, t)] \Rightarrow f_{i,j,k}^{\hat{x}} = \frac{1}{2}[f_{i,j,k}^x + f_{i,j,k}^{\bar{x}}],$$

Donde la parte derecha de las implicaciones resulta de reescribirlas utilizando la notación descrita en el primer apartado de esta sección (suponiendo que el punto (x, y, t) pertenece a la malla y es de la forma (x_i, y_j, t_k)).

Un caso de especial interés es si aplicamos (se puede comprobar que el orden no importa) primero (1.1) a f y luego (1.2) al resultado, que nos da la siguiente igualdad:

$$\begin{aligned}
f^{x\bar{x}}(x, y, t) &= f^{\bar{x}x}(x, y, t) = \\
\frac{1}{\Delta x^2} [f(x + \Delta x, y, t) - 2f(x, y, t) + f(x - \Delta x, y, t)] &\Rightarrow \\
f_{i,j,k}^{x\bar{x}} = f_{i,j,k}^{\bar{x}x} &= \frac{1}{\Delta x^2} [f_{x+1,j,k} - 2f_{i,j,k} + f_{i-1,j,k}].
\end{aligned} \tag{1.3}$$

1.4. Plan de trabajo

Para realizar el estudio, los lenguajes de programación que utilizaremos serán CUDA (una extensión de C++ que permite la ejecución de funciones -llamadas *kernels*- en la GPU) y Python, que tiene una librería llamada *pycuda* para ejecutar código CUDA. Además, todos los programas aquí mostrados y los resultados obtenidos serán ejecutados en la misma máquina, con las siguientes especificaciones, no obstante, los programas están pensados para poder ejecutarse en cualquier máquina²

Procesador Intel© Core™ i5-10400F CPU @ 2.90GHz × 6

RAM 15.5 GiB

GPU NVIDIA Corporation GA104 [GeForce RTX 3070]

SO Linux Mint 21.3 Cinnamon

En el Capítulo 2 explicaremos cómo ejecutar programas en la GPU usando la librería *pycuda* (para lo que primero debemos de entender como programar en CUDA), seguido de un par de programas para familiarizarnos con las librerías y ver que, en efecto, pueden acelerar los tiempos de ejecución.

En los Capítulos 3, 4 y 5 demostraremos los resultados necesarios para asegurar la existencia y unicidad de las ecuaciones del calor, ondas y Laplace respectivamente. Una vez asegurada la existencia y unicidad, propondremos un esquema numérico para aproximar la solución en los puntos de la malla y demostraremos su convergencia (que generalmente dependerán de los parámetros con los que construyamos la maya).

Después de haber propuesto los métodos numéricos, en el Capítulo 6 los implementaremos en Python utilizando computación clásica, y analizaremos la eficiencia de estos

Por último, en el Capítulo 7 implementaremos los algoritmos en CUDA aprovechándonos de la computación paralela para hacerlos más rápidos que sus equivalentes en Python, y tras esto analizaremos el incremento de eficiencia que nos ha proporcionado la GPU.

²Si se desea utilizar el script *generateMod.py* en un sistema basado en Windows o MAC, podría ser necesario hacer unos pocos cambios para adaptarse a las distintas formas de nombrar las rutas en estos sistemas.

Capítulo 2

Computación en GPU con CUDA y pycuda

RESUMEN: En este capítulo se pretenden introducir los conceptos básicos de la programación en GPU, en particular en el lenguaje CUDA C++ utilizado desde la librería de Python pycuda.

2.1. Bases de la programación en GPU

Las GPUs son componentes hardware del ordenador pensados para acelerar el procesamiento de los gráficos. Esto se hace aprovechando que la mayoría de los cálculos que se requieren para procesar estos se puede ejecutar de manera simultánea en paralelo.

Dada la naturaleza de las tarjetas gráficas, resulta común utilizarlas para implementar algoritmos que puedan beneficiarse de este paralelismo. Con ese fin, NVIDIA desarrolló el lenguaje de programación CUDA¹, una extensión de C/C++ que permite la definición de *kernels*.

Los *kernels* son funciones de C/C++ diseñadas para ejecutarse en varios hilos al mismo tiempo en la GPU (a la que llamaremos *dispositivo* en este contexto). Estos kernels serán luego ejecutados desde la CPU, que en este contexto llamaremos *host*.

2.2. Programación en pycuda

Como adelantamos en la Sección 1, utilizaremos pycuda para poder ejecutar código CUDA en la GPU desde un programa convencional en Python. Este será el proceso para hacer un programa:

1. Generar archivos con la extensión .cu que contengan la definición de las funciones que vayamos a ejecutar en la GPU, escritas en CUDA. Este será el único

¹Para ver el manual completo de CUDA, consultar el siguiente enlace: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

código que escribamos en este lenguaje de programación, todo lo demás estará escrito en Python.

2. Crear un programa en Python, que será el que ejecutemos. Éste será el que se encargue de compilar y llamar (mediante las funciones de pycuda) a las funciones definidas en el apartado anterior. Para hacer esto, utilizaremos el programa `generateMod.py` (ver A.1). Además del programa como tal, tendremos que utilizar unas funciones de pycuda para reservar memoria y llevar los datos al dispositivo (para ver más sobre la gestión de memoria en dispositivo, ver 2.3.1).

2.3. Programación en CUDA

Aunque no vayamos a escribir mucho código en este lenguaje propiamente, necesitamos entender bien cómo funciona para evitar errores. La única diferencia que nos encontramos con C/C++ en la sintaxis, es que todos los *kernels* que definamos tienen que tener uno de los siguientes identificadores, que describe desde dónde se van a llamar y dónde se van a ejecutar:

device Funciones que van a ser llamadas y ejecutadas desde el dispositivo (la GPU)

global Funciones que van a ser llamadas desde la CPU pero ejecutadas en el dispositivo. Cuando llamemos a estas funciones les pasaremos como meta-parámetro el número de veces que se van a ejecutar.

host Son funciones habituales de C++. Dado que vamos a utilizar CUDA a través de pycuda, no utilizaremos este tipo de funciones, ya que toda la programación en CPU la haremos en Python.

2.3.1. Gestión de memoria

Es importante comprender que el dispositivo y el host son unidades de computación diferentes. Esto significa que no comparten espacio de direcciones de memoria y, por tanto, tenemos que proceder de manera distinta dependiendo si estamos usando variables del dispositivo o del host.

A pesar de la necesidad de entender el funcionamiento de la memoria, no usaremos las funciones de gestión de memoria de CUDA, ya que la librería pycuda tiene sus propias funciones para hacer esto de manera un poco más cómoda.

Memoria en el host

Para trabajar con la memoria en la CPU se utilizarían las funciones habituales de C/C++ como *malloc* (para reservar memoria), pero como estamos trabajando sobre Python, no tenemos que preocuparnos por eso.

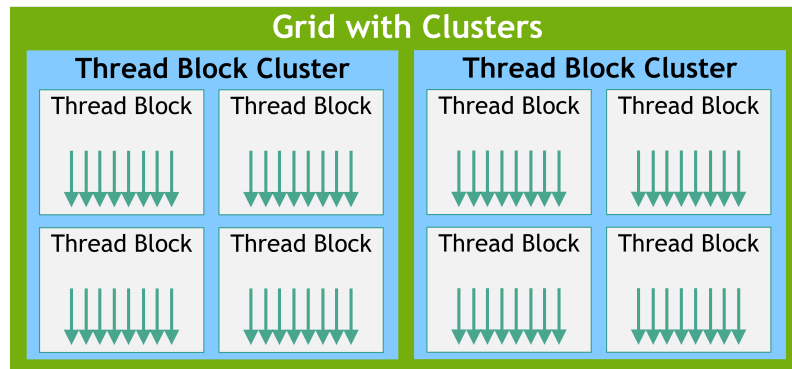


Figura 2.1: Cuadrícula de bloques de hilos

Memoria en el dispositivo

Al crear la memoria para el dispositivo desde el host, no podemos emplear las funciones habituales, pues estas no reservan memoria en la GPU. Existen funciones específicas para esto como *cudaMalloc*.

Compartir memoria entre dispositivo y host

Para esto, usamos unas funciones que nos proporciona pycuda, *memcpy_htod* y *memcpy_dtoh*, que nos permiten copiar memoria desde el host al dispositivo y en dirección contraria respectivamente.

2.3.2. Hilos y bloques

Al ejecutar un kernel, tenemos que especificar cuantos hilos ejecutarán esa función, los hilos se organizan en bloques, que a su vez se organizan en una cuadrícula (grid)² como podemos observar en la imagen 2.1

Un bloque es un conjunto de hilos (que se distribuyen en tres dimensiones) que, como máximo, puede tener 1024 hilos diferentes por motivos de implementación en memoria (podemos tener un bloque de 1024x1x1 o 256x2x2 por ejemplo, pero no de 256x4x2, ya que el límite se refiere a la cantidad total de hilos, no al máximo en cada dimensión).

Podemos tener tantos bloques como queramos, organizados a su vez en la cuadrícula (también tridimensional). En cada hilo podemos acceder al índice de su bloque y del hilo dentro del bloque con *blockIdx.dim* y *threadIdx.dim* respectivamente, siendo dim la dimensión, osea x, y o z.

Aunque a nivel de usuario pueda parecer enrevesada esta distribución, esto está estrechamente relacionado con la implementación y la memoria compartida como podemos observar en la imagen 2.2. Todos los hilos tienen su propia memoria privada y una memoria compartida entre los demás hilos de su bloque, pero para compartir memoria con otros bloques hay que utilizar otros mecanismos que pueden afectar

²En las últimas versiones también podemos tener Thread Block Clusters para agrupar bloques, pero esos no los trataremos en este trabajo.

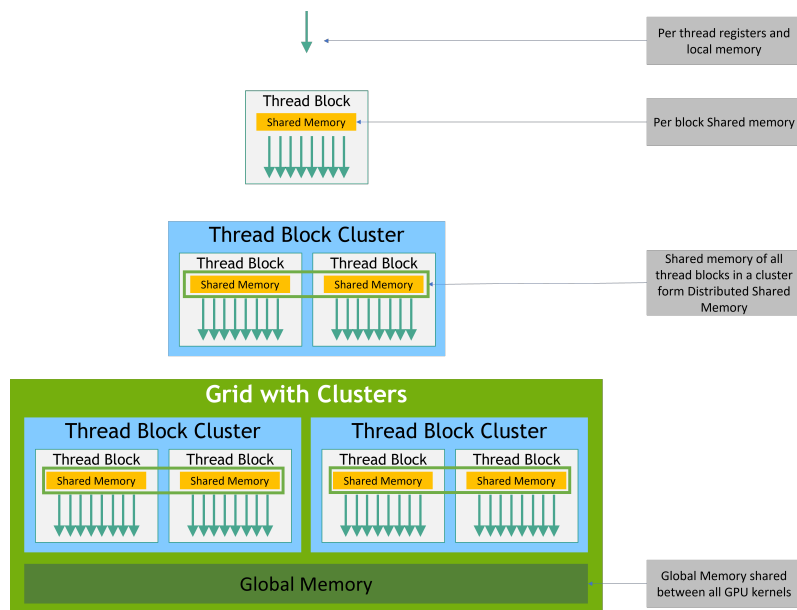


Figura 2.2: Memoria compartida entre elementos

al rendimiento, luego implementar los hilos de manera adecuada puede hacer más eficientes los programas.

2.4. Ejemplos

Procedemos ahora a hacer un par de ejemplos sencillos que nos permitan familiarizarnos con estos conceptos.

2.4.1. Hola Mundo

Como es costumbre a la hora de programar, lo primero es hacer un "hola mundo", un programa que escriba en la consola la frase "hola mundo". Como el objetivo es poner de manifiesto que las cosas se están ejecutando varias veces en la GPU, escribiremos el texto dos veces.

- En A.2 vemos un ejemplo de uso de *generateMod.py*, así como las funciones para reservar y copiar memoria en pycuda. Es importante destacar que a la hora de llamar al kernel desde la CPU, le tenemos que indicar qué forma va a tener cada uno de los bloques (osea, el número de hilos y cómo van a estar repartidos a lo largo de sus dimensiones).

- En A.3 podemos ver la implementación del hola mundo en CUDA, con el uso de la etiqueta `__global__` mencionada anteriormente.

- En A.4 vemos que la salida es la esperada.

2.4.2. Suma de vectores

Habiendo tenido ya nuestro primer contacto con pycuda, vamos ahora a poner de manifiesto la mejora de rendimiento que se puede conseguir. Vamos a realizar la suma de dos vectores de tamaños incrementalmente grandes, primero en CPU y luego en GPU, y vamos a comparar los tiempos que tarda en hacer ambas cosas.

- En A.5 podemos ver un ejemplo más complejo de código Python usando pycuda. La función `add_random_vects` genera dos vectores de números aleatorios y los suma, primero en la CPU, y luego en GPU, midiendo el tiempo de ambos. Hay que destacar un par de cosas importantes:

Por un lado, hay que hacer la gestión de memoria, osea reservar memoria en GPU y copiar los datos con las funciones de pycuda. Por otro lado hay que gestionar el tamaño de bloque. Recordemos que los bloques pueden tener a lo sumo 1024 hilos, mientras que las grids pueden ser arbitrariamente grandes³, esto implica que podríamos simplemente ejecutar un hilo por bloque y tantas mallas como el tamaño del vector, pero si recordamos la imagen 2.2 podemos observar que los hilos de cada bloque comparten memoria local, por lo que si hacemos el máximo uso posible de los bloques (osea tratar con bloques de 1024 hilos), haremos un menor uso de la memoria y, por tanto, obtendremos resultados notablemente mejores.

Por último, comprobamos que las sumas coincidan y mostramos el incremento de eficiencia. Al ejecutar este script, simplemente llamamos a la función para valores de n entre 1 y 10^{10}

- En A.6 el código es bastante inmediato, calculamos el índice al que le corresponde nuestro hilo concreto y hacemos la suma. Solo hay una sutileza, y es que, en el último bloque que utilizemos, probablemente algunos hilos estén trabajando posiciones no válidas (porque todos los bloques tienen la misma cantidad de hilos, luego hay más hilos que posiciones en el vector). Para no acceder a posiciones de memoria posiblemente inválidas, simplemente añadimos a los datos de entrada el tamaño del vector y, si el hilo tiene un índice superior al tamaño del vector, no hacemos nada.

- En A.7 se pone de manifiesto la mejora en la eficiencia de los algoritmos paralelos, que pueden ser hasta 10 veces más rápidos que los convencionales para datos de entrada suficientemente grandes.

³En realidad hay un límite de tamaño dependiente del hardware, pero es tan grande que puede desestimarse.

Capítulo 3

Ecuación del calor

RESUMEN: En este capítulo desarrollaremos dos métodos numéricos, basados en las diferencias finitas, para aproximar la solución a la ecuación del calor en una y dos dimensiones.

3.1. Caso lineal

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

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

La condición que pediremos para asegurar la existencia y unicidad de la solución será tener un valor concreto en el tiempo inicial ($t = 0$) y conocer el valor de la solución a lo largo de la frontera del intervalo, por lo que el problema queda de la siguiente manera

$$\begin{cases} \frac{\partial u}{\partial t}(x, t) = \frac{\partial^2 u}{\partial x^2}(x, t), & a \leq x \leq b, \ 0 \leq t \leq T, \\ u(x, 0) = f(x), & a \leq x \leq b, \\ u(a, t) = \alpha(t), & 0 < t, \\ u(b, t) = \beta(t), & 0 < t. \end{cases} \quad (3.1)$$

Salta a la vista que el dominio de este problema es un conjunto convexo, $R := \{(x, t) \mid a \leq x \leq b, \ 0 \leq t \leq T\}$.

3.1.1. Existencia y unicidad

Antes de demostrar la existencia y unicidad necesitamos definir un par de conceptos:

Definición 3.1.1 (Frontera parabólica). *Definimos como frontera parabólica, que denotaremos como B al conjunto $\partial R \setminus \{(x, T) \mid a < x < b\}$, siendo ∂R la frontera (en el sentido topológico usual) de R .*

Definición 3.1.2 (Función continua a trozos). *Una función es continua a trozos si la función es continua en todo su dominio excepto en una cantidad finita de puntos.*

Enunciamos ahora dos teoremas que necesitaremos.

Teorema 3.1.1 (Unicidad). *Sean u y v soluciones del problema (3.1) continuas en R , si $u = v$ en B entonces $u = v$ en todo R*

Teorema 3.1.2 (Unicidad extendida). *Sean u y v soluciones del problema (3.1) continuas a trozos en R con una cantidad finita de discontinuidades acotadas, si $u = v$ en B (excepto los puntos de discontinuidad) entonces $u = v$ en todo R*

Estos teoremas nos dicen que basta con comprobar que todas las soluciones coinciden en la frontera del rectángulo para ver que la solución es única.

Demostración. Ver [1, Th. 1.6.4] y [1, Th. 1.6.6] □

Teorema 3.1.3 (Existencia y unicidad). *Sean f , α y β funciones continuas a trozos tales que $f(a) = \alpha(0)$ y $f(b) = \beta(0)$, la función*

$$u(x, t) = \int_a^b \theta(x - \xi, t) - \theta(x + \xi, t) f(\xi) d\xi \\ - 2 \int_0^t \frac{\partial \theta}{\partial x}(x, t - \tau) \alpha(\tau) d\tau + 2 \int_0^t \frac{\partial \theta}{\partial x}(x - 1, t - \tau) \beta(\tau) d\tau \quad (3.2)$$

donde $\theta(x, t)$ y $K(x, t)$ se definen como

$$\theta(x, t) = \sum_{m=-\infty}^{\infty} K(x + 2m, t) \quad t > 0$$

$$K(x, t) = \frac{e^{-\frac{x^2}{4t}}}{\sqrt{4\pi t}} \quad t > 0$$

es la única solución acotada del problema 3.1

Demostración. Puede verse en [1, Secs. 6.1-2] que en efecto (3.2) es solución de la ecuación de (3.1), por lo que solo tenemos que preocuparnos por la unicidad. Esto es inmediato por el teorema 3.1.3, ya que las condiciones iniciales fijan el valor de cualquier solución en B . □

Ahora podemos asegurar que (3.1) tiene una única solución, por lo que podemos proceder a aproximarla con un método de diferencias finitas.

3.1.2. Aproximación de la solución¹

Teniendo en cuenta toda la notación descrita en la Sección 1.3.2, así como la definición 1.3.1, vamos a aproximar la solución por la función U en los puntos de la malla $M^2(a, b, 0, T, n_x, n_t)$, que a partir de ahora denotaremos simplemente como M . Como en este caso el dominio es el rectángulo R , está claro que $(x_i, t_j) \in R \iff 0 \leq i < n_x, 0 \leq j < n_t$.

Pediremos a la función de aproximación U que cumpla la igualdad $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$, pero sustituyendo las derivadas parciales por sus aproximaciones mediante las diferencias finitas (1.1) y (1.3), lo que nos lleva a lo siguiente

$$U^t(x, t) = U^{x\bar{x}} \Rightarrow \frac{U(x, t + \Delta t) - U(x, t)}{\Delta t} = \frac{1}{\Delta x^2} [U(x + \Delta x, t) - 2U(x, t) + U(x - \Delta x, t)].$$

Sean i y j tales que $x = x_i$ y $t = t_j$, si reescribimos la ecuación anterior, obtenemos

$$\frac{U_{i,j+1} - U_{i,j}}{\Delta t} = \frac{1}{\Delta x^2} [U_{i+1,j} - 2U_{i,j} + U_{i-1,j}]$$

lo que, tras despejar y definir $\lambda \equiv \frac{\Delta t}{\Delta x^2}$ nos lleva a la fórmula explícita

$$U_{i,j+1} = (1 - 2\lambda)U_{i,j} + \lambda(U_{i+1,j} + U_{i-1,j}). \quad (3.3)$$

La fórmula (3.3) nos permite calcular el valor de $U_{i,j}$ para cualquier $i \in \{1, \dots, n_x - 2\}$ siempre que conozcamos los valores de $U_{i,j-1} \forall i \in \{0, 1, \dots, n_x - 1\}$.

Si definimos $U_{i,0} := f(a + i\Delta x)$, $U_{0,j} := \alpha(j\Delta t)$ y $U_{n_x-1,j} := \beta(j\Delta t)$, nos aseguramos que la aproximación de la solución U es exacta en los puntos de la malla que coinciden con el conjunto B . Utilizando los valores iniciales de U y la fórmula (3.3) podemos calcular todos los valores $U_{i,j}$.

En general, supondremos que el objetivo es aproximar los valores de $u(x, T)$ en los puntos de la malla que proceden, osea calcular los valores $U_{i,n_t-1} \forall i \in \{1, \dots, n_x - 2\}$ (los valores para $i = \{0, n_x - 1\}$ los sabemos por las condiciones de contorno). En la Figura 3.1 podemos observar una representación a pequeña escala de qué puntos tenemos por las condiciones de contorno e iniciales, qué puntos queremos aproximar, y qué puntos intermedios aproximamos para poder llegar al objetivo.

Antes de estudiar la convergencia de (3.3), presentamos el siguiente lema:

Lema 3.1.1. *Suponiendo que u es suficientemente diferenciable, se tiene que*

$$u^t(x, t) - u^{x,\bar{x}}(x, t) = \tau(x, t), \quad (3.4)$$

donde τ es el error de truncamiento, definido como

$$\tau(x, t) = \frac{\Delta t}{2} \frac{\partial^2 u}{\partial t^2} + \frac{\Delta x^2}{12} \frac{\partial^4 u}{\partial x^4} + \xi_{\Delta x, \Delta t}. \quad (3.5)$$

Siendo $\xi_{\Delta x, \Delta t}$ un número real que cumple que

$$\xi_{\Delta x, \Delta t} \xrightarrow{\Delta x \rightarrow 0, \Delta t \rightarrow 0} 0$$

¹Las demostraciones de toda esta sección son modificaciones propias de [1], con el fin de hacerlas lo más sencillas posibles.

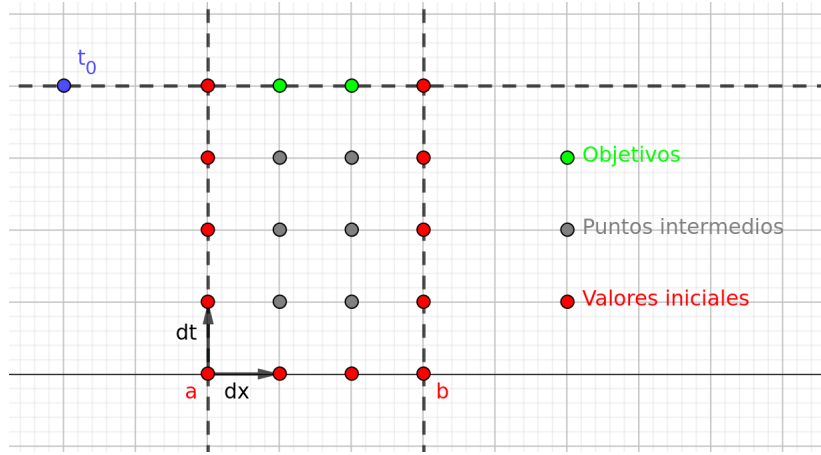


Figura 3.1: Representación en la malla del esquema (3.3)

Demostración. Haciendo el desarrollo de Taylor de orden 2 para la función u con centro en el punto (x, t) tenemos

$$u(x, t + \Delta t) = u(x, t) + \frac{\partial u}{\partial t} \Delta t + \frac{1}{2} \frac{\partial^2 u}{\partial t^2} \Delta t^2 + \xi_{\Delta x, \Delta t}^1 \Delta t^2 \Rightarrow$$

$$u^t(x, t) = \frac{\partial u}{\partial t} + \frac{\Delta t^2}{2} \frac{\partial^2 u}{\partial t^2} \Delta t + \xi_{\Delta x, \Delta t}^1 \Delta t,$$

siendo $\xi_{\Delta x, \Delta t}^1$ un número que cumple la propiedad que le pedimos a $\xi_{\Delta x, \Delta t}$.

Si nos fijamos en la igualdad resultante, la parte izquierda es exactamente el sumando izquierdo de (3.4), y la parte derecha es el primer sumando de (3.5) más los términos $\frac{\partial u}{\partial t}$ y $\xi_{\Delta x, \Delta t}^1$.

Si ahora hacemos los desarrollos de Taylor de orden 4 para la misma función y el mismo centro pero en otros puntos, obtenemos

$$u(x + \Delta x, t) =$$

$$u(x, t) + \frac{\partial u}{\partial x} \Delta x + \frac{1}{2!} \frac{\partial^2 u}{\partial x^2} \Delta x^2 + \frac{1}{3!} \frac{\partial^3 u}{\partial x^3} \Delta x^3 + \frac{1}{4!} \frac{\partial^4 u}{\partial x^4} \Delta x^4 + \xi_{\Delta x, \Delta t}^2 \Delta x^4 \quad (3.6)$$

y

$$u(x - \Delta x, t) =$$

$$u(x, t) - \frac{\partial u}{\partial x} \Delta x + \frac{1}{2!} \frac{\partial^2 u}{\partial x^2} \Delta x^2 - \frac{1}{3!} \frac{\partial^3 u}{\partial x^3} \Delta x^3 + \frac{1}{4!} \frac{\partial^4 u}{\partial x^4} \Delta x^4 + \xi_{\Delta x, \Delta t}^3 \Delta x^4$$

siendo $\xi_{\Delta x, \Delta t}^2$ y $\xi_{\Delta x, \Delta t}^3$ números que cumplen la propiedad que le pedimos a $\xi_{\Delta x, \Delta t}$. Ahora, si sumamos las dos últimas ecuaciones, ya que se nos cancelan los términos con exponente impar, obtenemos

$$u^{x, \bar{x}} - \frac{\partial^2 u}{\partial x^2} = \frac{1}{12} \frac{\partial^2 u}{\partial x^2} \Delta x^2 + (\xi_{\Delta x, \Delta t}^2 + \xi_{\Delta x, \Delta t}^3) \Delta x^2. \quad (3.7)$$

Si restamos (3.6) y (3.7) (teniendo en cuenta que $\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0$) obtenemos precisamente la igualdad (3.4), siendo $\xi_{\Delta x, \Delta t} = \xi_{\Delta x, \Delta t}^1 \Delta t - (\xi_{\Delta x, \Delta t}^2 + \xi_{\Delta x, \Delta t}^3) \Delta x^2$, por lo que se confirma que tiende a 0 cuando los incrementos tienden a 0. \square

Teorema 3.1.4. Si $0 < \lambda < \frac{1}{2}$, el método numérico (3.3) es convergente², o dicho de otra forma, si definimos el error de aproximación del método $\epsilon_{i,j} := |u_{i,j} - U_{i,j}|$, $\sup_{i,j} \epsilon(i,j) \rightarrow 0$ si $\Delta t, \Delta x \rightarrow 0$.

Demostración. Teniendo en cuenta el lema anterior, si ahora repetimos las cuentas que nos llevaron a (3.3) pero para u en lugar de para U , obtenemos la ecuación

$$u_{i,j+1} = (1 - 2\lambda)u_{i,j} + \lambda(u_{i-1,j} + u_{i+1,j}) + \Delta t \tau_{i,j},$$

por lo que

$$\begin{aligned} \epsilon_{i,j+1} &= |u_{i,j+1} - U_{i,j+1}| = \\ &= |((1 - 2\lambda)u_{i,j} + \lambda(u_{i-1,j} + u_{i+1,j}) + \Delta t \tau_{i,j}) \\ &\quad - ((1 - 2\lambda)U_{i,j} + \lambda(U_{i-1,j} + U_{i+1,j}))| = \\ &= |(1 - 2\lambda)(u_{i,j} - U_{i,j}) + \lambda(u_{i+1,j} - U_{i+1,j} + u_{i-1,j} - U_{i-1,j}) + \Delta t \tau_{i,j}|. \end{aligned}$$

Utilizando la desigualdad triangular, obtenemos

$$\begin{aligned} \epsilon_{i,j+1} &\leq (1 - 2\lambda)|u_{i,j} - U_{i,j}| + \lambda(|u_{i+1,j} - U_{i+1,j}| + |u_{i-1,j} - U_{i-1,j}|) + \Delta t |\tau_{i,j}| \\ &= (1 - 2\lambda)\epsilon_{i,j} + \lambda(\epsilon_{i+1,j} + \epsilon_{i-1,j}) + \Delta t |\tau_{i,j}|. \end{aligned}$$

Ahora, si definimos

$$E_j := \sup_i |\epsilon_{i,j}| \quad \tau := \sup_{i,j} |\tau_{i,j}|$$

tenemos que

$$E_{j+1} \leq E_j + \Delta t \tau,$$

mediante una inducción trivial concluimos que

$$E_j \leq E_0 + j \Delta t \tau = E_0 + t_j \tau = t_j \tau, \quad \forall j \geq 0$$

donde E_0 es el supremo del error en $t = 0$ (que es 0 porque $U_{i,0} = f(x_i) = u_{i,0}$ por definición).

Con esto hemnos acabado la demostración, pues está claro (por su definición) que τ tiende a 0 cuando $\Delta t, \Delta x$ tienden a 0, luego el error está acotado por algo que tiende a 0, lo que implica que tiende a 0. \square

3.2. Caso bidimensional

En el caso bidimensional, la ecuación que tenemos será

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

COMENTARIO: Voy a dejar esto en stand by, estoy teniendo problemas para encontrar algún teorema de existencia y unicidad para alguna condición inicial, y por tanto no sé que condiciones iniciales voy a acabar pidiéndole.

²De hecho, el método converge si y solo si $0 < \lambda < \frac{1}{2}$, pero no lo probaremos por que solo nos interesa cuando converge

Capítulo 4

Ecuación de onda

RESUMEN: En este capítulo desarrollaremos dos métodos numéricos, basados en las diferencias finitas, para aproximar la solución a la ecuación de onda en una y dos dimensiones.

4.1. Caso lineal

La ecuación de onda tiene la forma

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

y consideraremos el dominio $R = \{(x, t) \mid 0 \leq t\}$ y el problema inicial que fija el valor de la función y su derivada en $t = 0$, lo que nos deja con el problema

$$\begin{cases} \frac{\partial^2 u}{\partial t^2}(x, t) - c^2 \frac{\partial^2 u}{\partial x^2}(x, t) = 0, & -\infty < x < \infty, \quad 0 \leq t, \\ u(x, 0) = f(x), & -\infty < x < \infty, \\ \frac{\partial u(x, 0)}{\partial t} = g(x), & -\infty < x < \infty. \end{cases} \quad (4.1)$$

4.1.1. Existencia y unicidad

Como es costumbre, empezamos con un resultado¹ sobre la existencia y unicidad de la solución. Supongamos que u fuera solución de (4.1), si hacemos el cambio de variable $\xi = x - ct$, $\eta = x + ct$, aplicando la regla de la cadena, obtenemos

$$\frac{\partial u}{\partial x} = \frac{\partial \xi}{\partial x} \frac{\partial u}{\partial \xi} + \frac{\partial \eta}{\partial x} \frac{\partial u}{\partial \eta} = \frac{\partial u}{\partial \xi} + \frac{\partial u}{\partial \eta}$$

y

$$\frac{\partial u}{\partial t} = \frac{\partial \xi}{\partial t} \frac{\partial u}{\partial \xi} + \frac{\partial \eta}{\partial t} \frac{\partial u}{\partial \eta} = -c \frac{\partial u}{\partial \xi} + c \frac{\partial u}{\partial \eta}.$$

¹Este puede encontrarse en [3]

Derivando una segunda vez, llegamos a

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial \xi^2} + 2 \frac{\partial^2 u}{\partial \xi \partial \eta} + \frac{\partial^2 u}{\partial \eta^2}$$

y

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial \xi^2} - 2c^2 \frac{\partial^2 u}{\partial \xi \partial \eta} + c^2 \frac{\partial^2 u}{\partial \eta^2}.$$

Luego, si sustituimos en (4.1) utilizando las últimas expresiones, concluimos que u tiene que cumplir la siguiente ecuación

$$\frac{\partial^2 u}{\partial \xi \partial \eta} = 0.$$

Si integramos dos veces, podemos observar que u tiene que ser de la forma

$$u(x, t) = P(\xi) + Q(\eta) = P(x - ct) + Q(x + ct) \quad (4.2)$$

para unas funciones P y Q arbitrarias, como u es solución de (4.1), tiene que cumplir sus restricciones, por lo que P y Q tienen que cumplir.

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

Ahora, si integramos la segunda relación en el intervalo $[0, x]$, y la sumamos y restamos a la primera, concluimos que las funciones P y Q tienen que ser exactamente de la forma (suponiendo que la función g sea integrable)

$$P(x) = \frac{1}{2}f(x) + \frac{1}{2c} \int_0^x g(\zeta) d\zeta + K$$

y

$$Q(x) = \frac{1}{2}f(x) - \frac{1}{2c} \int_0^x g(\zeta) d\zeta - K,$$

siendo $K = \frac{1}{2}[P(0) - Q(0)]$.

Por tanto, si suponemos que (4.1) tiene solución, hemos encontrado como tienen que ser las funciones P y Q , pero si ahora sustituimos eso en (4.2), concluimos que la solución tiene que ser exactamente de la forma

$$u(x, t) = \frac{1}{2}[f(x + ct) + f(x - ct)] + \frac{1}{2c} \int_{x-ct}^{x+ct} g(\zeta) d\zeta. \quad (4.3)$$

Con esto no solo hemos probado que la solución es única (ya que cualquier solución arbitraria tiene que ser exactamente la función (4.3)), también hemos probado que siempre que la función u esté bien definida (osea, siempre que g sea integrable), existe una solución a la ecuación, por tanto tenemos el siguiente resultado.

Teorema 4.1.1 (Existencia y unicidad). *Si g es una función integrable, el problema (4.1) tiene una única solución.*

4.1.2. Aproximación de la solución²

Teniendo en cuenta toda la notación descrita en la Sección 1.3.2, así como la definición 1.3.1, vamos a aproximar la solución por la función U en los puntos de la malla $M^2(a, b, 0, T, n_x, n_t)$, que a partir de ahora denotaremos simplemente como M . Como en este caso el dominio es el semiplano $\{0 \leq t\}$, está claro que $(x_i, t_j) \in R \iff 0 \leq j$.

Pediremos a la función de aproximación U que cumpla la igualdad $\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$, pero sustituyendo las derivadas parciales por sus aproximaciones mediante las diferencias finitas (1.3), lo que nos lleva a lo siguiente

$$U^{tt}(x, t) - c^2 U^{xx}(x, t) = 0 \Rightarrow \frac{1}{\Delta t^2} [U(x, t + \Delta t) - 2U(x, t) + U(x, t - \Delta t)] = \left(\frac{c}{\Delta x}\right)^2 [U(x + \Delta x, t) - 2U(x, t) + U(x - \Delta x, t)].$$

Sean i y j tales que $x = x_i$ y $t = t_j$, si reescribimos la ecuación anterior, obtenemos

$$\begin{aligned} \frac{1}{\Delta t^2} [U_{i,j+1} - 2U_{i,j} + U_{i,j-1}] &= \left(\frac{c}{\Delta x}\right)^2 [U_{i+1,j} - 2U_{i,j} + U_{i-1,j}] \Rightarrow \\ U_{i,j+1} &= \left(c \frac{\Delta t}{\Delta x}\right)^2 [U_{i+1,j} - 2U_{i,j} + U_{i-1,j}] + 2U_{i,j} - U_{i,j-1} \end{aligned}$$

que, tras combinar elementos y definir $\lambda = c \frac{\Delta t}{\Delta x}$, nos lleva a la fórmula explícita

$$U_{i,j+1} = 2[1 - \lambda^2] U_{i,j} + \lambda^2 [U_{i+1,j} + U_{i-1,j}] - U_{i,j-1}, \quad j \geq 1. \quad (4.4)$$

La fórmula (4.4) nos permite calcular el valor de $U_{i,j}$ para cualquier $i \in \mathbb{Z}$ siempre que conozcamos los valores de $U_{i,j-1}$ y $U_{i,j-2} \forall i \in \mathbb{Z}$, luego antes de poder utilizar esta fórmula, necesitaremos aproximar los valores de $U_{i,0}$ y $U_{i,1}$.

Los valores para $t = 0$ son sencillos, pues por las condiciones iniciales sabemos el valor exacto de la función u en estos puntos, por lo que definimos $U_{i,0} := f(a + i\Delta x) \forall i \in \mathbb{Z}$ y sabemos que el error cometido en esta aproximación es nulo.

Para $t = 1$, utilizaremos el desarrollo de Taylor de orden uno de la función $v(t) = u(x, t)$ centrado sobre el 0, lo que nos da la siguiente igualdad

$$u(x, t) = u(x, 0) + \Delta t \frac{\partial u}{\partial t} + \xi_{\Delta t} = f(x) + \Delta t g(x) + \xi_{\Delta t},$$

donde $\epsilon_{\Delta t}$ tiende a 0 cuando Δt tiende a 0. Por tanto, si definimos $U_{i,1} := f(a + i\Delta x) + \Delta t g(a + i\Delta x)$, el error que estamos cometiendo es $\xi_{\Delta t}$, que converge a cero si hacemos la malla más pequeña.

Ahora estamos en condiciones de calcular todos los valores de $U_{i,j}$. Supondremos que el objetivo es aproximar los valores de $u(x, T)$ en los puntos de la malla que proceden, o sea calcular los valores $U_{i,n_t-1} \forall i \in \{0, \dots, n_x - 1\}$. En la figura 4.1 podemos observar una representación a pequeña escala de qué puntos hemos definido a partir de las condiciones iniciales, qué puntos queremos aproximar y qué puntos intermedios aproximamos para llegar al objetivo.

²Las demostraciones de toda esta sección son modificaciones propias de [2]

Si tenemos en cuenta que $u_{i-j-1,0} = f_{i-j-1}$, y gracias a (4.1.2), podemos sustituir los sumandos del sumatorio por términos más simples, y obtener

$$U_{i,j+1} = f_{i-j-1} + \sum_{k=0}^j D_{i+j-2k,1} = f_{i-j-1} + \sum_{k=0}^j (U_{i+j-2k,1} - U_{i+j-2k-1,0}).$$

El valor exacto de $U_{i,0}$ y $U_{i,1}$ es f_i y $f_i + \Delta t g_i$ respectivamente, por lo que podemos sustituir y obtener

$$U_{i,j+1} = f_{i-j-1} + \Delta t \sum_{k=0}^j g_{i+j-2k} + \sum_{k=0}^j (f_{i+j-2k} - f_{i+j-2k-1}).$$

Como $t_n = n\Delta t$, se tiene que $x_n = n\Delta x = nc\Delta t = ct_n$, por ello, para cualquier función F se tiene que $F_{i+j} := F(x_{i+j}) = F(x_i + ct_j)$.

Como f tiene derivada continua, usando el teorema del valor medio obtenemos que $\exists \theta_k \in (-1, 0)$ tal que

$$f_{i+j-2k} - f_{i+j-2k-1} = f(x_{i-2k} + ct_j) - f(x_{i-2k} + ct_j - \Delta x) = \Delta x f'(x_{i-2k} + ct_j + \theta_k \Delta x).$$

Además, $f_{i-j-1} = f(x_{i-j-1}) = f(x_i - (j+1)\Delta x) = f(x_i - (j+1)c\Delta t) = f(x_i - ct_{j+1})$. Haciendo algo muy similar, obtenemos las igualdades $g_{i+j-2k} = g(x_i + ct_{j+1} - [2k+1]\Delta x)$ y $f'(x_{i-2k} + ct_j + \theta_k \Delta x) = f'(x_i + ct_{j+1} + \theta_k \Delta x - [2k+1]\Delta x)$

Sean x, t fijos hagamos tender $\delta x = c\Delta t$ a 0, y i, j tender a infinito de manera que se mantengan las relaciones $x_i = x$ y $t_j + 1 = t$ siempre. Si sustituimos en 4.1.2 con todas las igualdades que hemos obtenido, tendremos

$$\begin{aligned} u_{i,j} &= f(x_i - ct_{j+1}) + \frac{1}{2c} \sum_{k=0}^j 2\Delta x g(x_i + ct_{j+1} - [2k+1]\Delta x) + \\ &\quad \frac{1}{2} \sum_{k=0}^j 2\Delta x f'(x_i + ct_{j+1} + \theta_k \Delta x - [2k+1]\Delta x) \end{aligned}$$

Que, pasando al límite se nos queda en

$$\begin{aligned} u_{i,j} &= f(x_i - ct_{j+1}) + \frac{1}{2c} \lim_{\Delta x \rightarrow 0} \sum_{k=0}^j 2\Delta x g(x_i + ct_{j+1} - [2k+1]\Delta x) + \\ &\quad \frac{1}{2} \lim_{\Delta x \rightarrow 0} \sum_{k=0}^j 2\Delta x f'(x_i + ct_{j+1} + \theta_k \Delta x - [2k+1]\Delta x) = \\ &= f(x - ct) + \frac{1}{2c} \int_0^{2ct} g(x + ct - \xi) d\xi + \frac{1}{2} \int_0^{2ct} f'(x + ct - \xi) d\xi \end{aligned}$$

Pero el final de la última serie de igualdades es exactamente (4.3), luego hemos demostrado que la aproximación tiende a la solución exacta de la ecuación. \square

4.2. Caso bidimensional

Capítulo 5

Ecuación de Laplace

RESUMEN: En este capítulo desarrollaremos un método numérico, basado en las diferencias finitas, para aproximar la solución a la ecuación de Laplace en dos dimensiones.

La ecuación de Laplace en dos dimensiones es de la forma

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

En nuestro caso, para simplificar, supondremos que el dominio es un rectángulo de la forma $D := \{(x, y) | 0 \leq x \leq a, 0 \leq y \leq b\}$. Por tanto, es claro que la frontera es $C := \{(x, y) | x = 0, a, 0 \leq y \leq b; y = 0, b, 0 \leq x \leq a\}$.

Capítulo 6

Algoritmos Clásicos

RESUMEN: En este capítulo implementaremos todos los métodos numéricos desarrollados en las secciones anteriores en python, con métodos de computación clásica, para más adelante poder compararlos con sus equivalentes en programación en GPU.

6.1. Ecuación del calor lineal

Recordamos que la fórmula a implementar es

$$u_{i,j+1} = (1 - 2\lambda)u_{i,j} + \lambda(u_{i+1,j} + u_{i-1,j})$$

y las condiciones iniciales que tenemos son de contorno, osea que sabemos el valor exacto de la solución en los extremos, por lo que para si queremos saber el estado el intervalo $[a, b]$ en el tiempo t_0 , habrá que calcular una serie de puntos intermedios que podemos observar en la figura 6.1.

Nuestro algoritmo (que puede encontrarse en A.8) recibe como entrada los siguientes datos:

- **intervalo:** El intervalo sobre el que vamos a trabajar.
- f, α, β : Las funciones que determinan las condiciones de contorno.
- **t_obj:** El tiempo objetivo al que queremos llegar.
- **nt, nx:** El número de fracciones de tiempo (sin incluir la primera) y de fracciones de espacio (sin incluir los extremos) respectivamente.

Primero calcula dt, dx y λ con los datos que le hemos dado, y nos avisa si el método pudiera no converger con esos datos.

Luego rellena la matriz de resultados, comenzando por los valores iniciales (utilizando las funciones f, α y β) y luego va rellenoando la tabla con la fórmula de abajo a arriba.

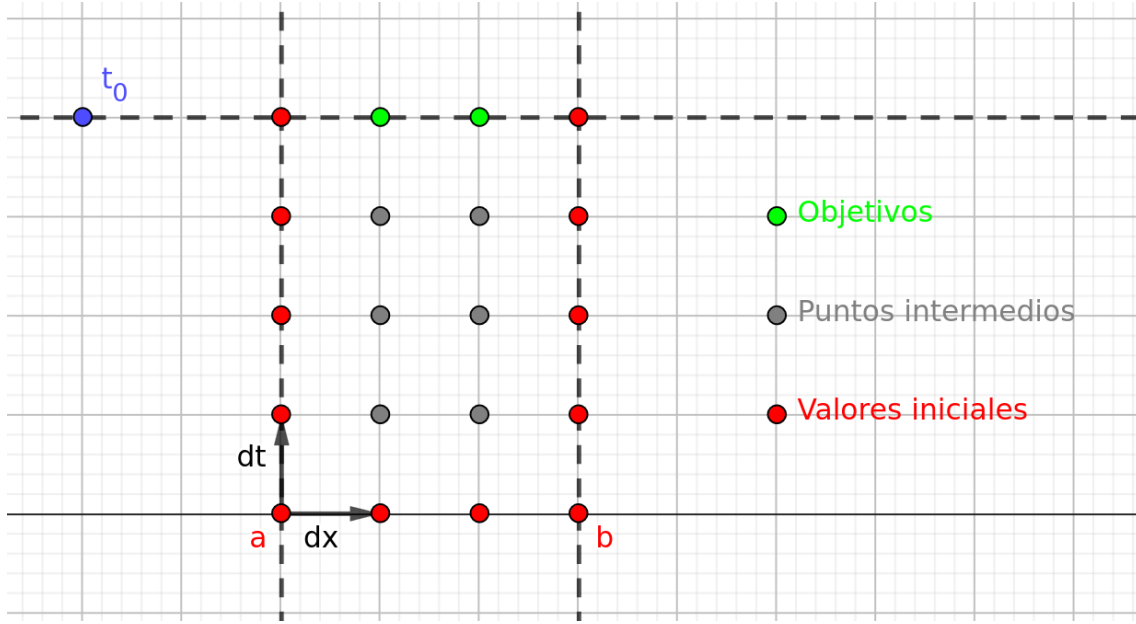


Figura 6.1: Diagrama que muestra qué puntos intermedios necesitamos calcular para obtener el resultado deseado

6.2. Ecuación del calor en un plano

6.3. Ecuación de onda lineal

En este caso, la fórmula a implementar es

$$u_{i,j+1} = 2 \left[1 - \left(c \frac{\Delta t}{\Delta x} \right)^2 \right] u_{i,j} + \left(c \frac{\Delta t}{\Delta x} \right)^2 [u_{i+1,j} + u_{i-1,j}] - u_{i,j-1}$$

y tenemos unas condiciones iniciales que nos dan el valor de la función en los dos primeros instantes de tiempo para cualquier x . Esto hará que para calcular los valores de la solución en el intervalo $[a, b]$ en t_0 necesitaremos conocer bastantes más puntos intermedios que en otros casos, como podemos observar en la figura 6.2.

El algoritmo (que se encuentra en A.11), recibe como entrada los siguientes datos:

- **a, n_puntos:** El valor inicial del intervalo objetivo y el número de punto que tiene.
- **f, g, c:** Las funciones que determinan el problema de valor inicial y la constante de la ecuación.
- **t_obj:** El tiempo objetivo al que queremos llegar.
- **nt:** El número de fracciones de tiempo (sin incluir la primera) que vamos a hacer. El número de fracciones de espacio se calcula a partir de esto para que $\lambda = 1$.

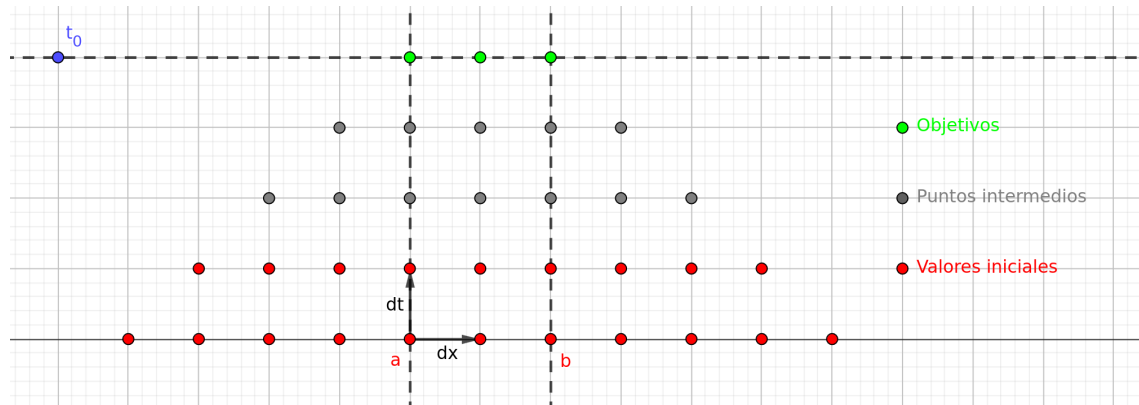


Figura 6.2: Diagrama que muestra qué puntos intermedios necesitamos calcular para obtener el resultado deseado

El algoritmo primero calcula dt , dx y nx a partir de los valores que le hemos dado, luego rellena usando las funciones f y g las dos primeras filas de la matriz (valor inicial) y rellena el resto de esta de abajo a arriba utilizando la fórmula.

6.4. Ecuación de onda en el plano

6.5. Ecuación de Laplace en el plano

Algoritmos en GPU

RESUMEN: En este capítulo implementaremos los mismos métodos numéricos que en el capítulo ??, pero utilizando la GPU para la parte con un coste computacional elevado.

7.1. Introducción

La idea para aprovechar el paralelismo que nos ofrece la GPU es que cada hilo se encargue de calcular (de manera paralela) el valor de una casilla de la matriz distinta, pero no podemos simplemente hacer un hilo por cada casilla y ponerlos a calcular, primero necesitamos entender un poco el concepto de sincronización.

Cuando se trabaja con programación paralela, la sincronización es lo más importante a tener en cuenta. Pongamos que estamos calculando valores en una matriz donde cada casilla de una fila depende exclusivamente de los valores de las casillas de filas menores (como pasa en las fórmulas (3.3) y (4.4)). En ese caso, podemos calcular todas las casillas de la misma fila al mismo tiempo (pues sus valores son independientes entre si), pero tendremos que esperar a que los valores necesarios de las filas anteriores estén calculados.

La idea es que, a parte de todo lo que

7.2. Ecuación del calor lineal

El código de este algoritmo está en A.9 y A.10. En esencia hace prácticamente lo mismo que su equivalente en python (A.8), con la diferencia de que el paralelismo nos permite calcular todos los elementos de una fila de manera simultánea y de la instrucción `cuda.Context.synchronize()`, que es necesaria para asegurar que todos los hilos han terminado de calcular los valores para una fila antes de seguir con la siguiente.

El paralelismo para este caso en concreto no va a introducir mucha mejora, porque como necesitamos que $\frac{\Delta t}{\Delta x^2} = \lambda < \frac{1}{2}$ para que converja, tenemos que $\Delta x >$

$\sqrt{2\Delta t}$. Esto significa que, para reducir el tamaño de Δx , tendremos que reducir el tamaño de Δt de manera cuadrática, por lo que para tener una malla muy densa en el eje x , necesitamos una malla muy densa en el eje t . Como estamos en un ordenador y tenemos recursos limitados, no podemos trabajar con mallas muy densas en el eje x (porque si no, la malla tendrá que ser mucho más densa en el eje t para mantener la convergencia), y por tanto no llegamos a aprovechar el paralelismo tanto como sería deseado.

{**TODO TODO TODO:** Hacer un estudio con tiempos concretos.}

7.3. Ecuación del calor en un plano

7.4. Ecuación de onda lineal

7.5. Ecuación de onda en el plano

7.6. Ecuación de Laplace en el plano

Bibliografía

*Y así, del mucho leer y del poco dormir, se
le secó el cerebro de manera que vino a
perder el juicio.*

Miguel de Cervantes Saavedra

- [1] J. R. Cannon. *The One-dimensional heat equation*. Reading, Massachusetts [etc] : Addison-Wesley, 1984.
- [2] E. Isaacson and H. B. Keller. *Analysis of Numerical Methods*. John Wiley and Sons, 1966.
- [3] 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

```
1 import pycuda.autoinit
2 from pycuda.compiler import SourceModule
3 import os, sys
4
5 def search_dir(path, files):
6     files += [f for f in os.listdir(path) if os.path.isfile(path +
7     os.sep + f) and f[-3:] == ".cu"]
8     for d in [path + os.sep + d for d in os.listdir(path) if os.
9     path.isdir(path+os.sep+d) and d != "__pycache__"]:
10         search_dir(d, files)
11
12 def init(files=None):
13     """
14     Generates SourceModule with the code and autoinits pycuda.
15
16     :param [str] files: The files from which the code is generated (
17     if none, all the files in the directory (recursive) are
18     considered)
19     :return: The SourceModule
20     """
21     if files == None:
22         files = []
23         search_dir(os.getcwd(), files)
24     code = ""
25     for file in files:
26         code += open(os.getcwd() + os.sep + "CUDA" + os.sep + file,
27         'r').read() + '\n'
28     return SourceModule(code)
29
30 if __name__ == "__main__":
31     files = []
32     search_dir(os.getcwd(), files)
33     print(files)
```

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

```

1 import generateMod as generateMod
2 import pycuda.driver as cuda
3 import numpy as np
4 import sys
5
6
7 mod = generateMod.init(["HelloWorld.cu"])
8 hello_world = mod.get_function("hello_world")
9 a = [1.0, 2.0]
10 a = np.asarray(a).astype("float32") #Esta es la memoria que
    gestionaremos en host, arrays de numpy
11 a_gpu = cuda.mem_alloc(a.nbytes) #Reservamos la memoria en GPU
12 cuda.memcpy_htod(a_gpu, a)
13 hello_world(a_gpu, block = (2,1,1))

```

Listing A.2: Hola Mundo (Código Python)

```

1
2 __global__ void hello_world(float* a){
3     int id = threadIdx.x;
4     printf("Hello World, my id is %i and my number is \"%f\".\n",
5         id, a[id]);
6 }

```

Listing A.3: Hola Mundo (Código CUDA)

```

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

```

Listing A.4: Hola Mundo (Salida)

```

1 import numpy as np
2 import generateMod as generateMod
3 import pycuda.driver as cuda
4 from timeit import default_timer as timer
5
6 mod = generateMod.init(["vectorAdd.cu"])
7 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.randn(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")

```

Listing 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 }

```

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

```

1 -----N = 1-----
2     Los resultados coinciden:
3     Tiempo en GPU 0.00044668700047623133
4     Tiempo en CPU: 4.9682000280881766e-05
5     La GPU es un 11.1% mas rapida
6 -----N = 100-----
7     Los resultados coinciden:
8     Tiempo en GPU 0.00012892499944427982
9     Tiempo en CPU: 2.4683999981789384e-05
10    La GPU es un 19.1% mas rapida
11 -----N = 10000-----
12    Los resultados coinciden:
13    Tiempo en GPU 0.00013271799980429932
14    Tiempo en CPU: 0.00031606100037606666
15    La GPU es un 238.1% mas rapida
16 -----N = 1000000-----
17    Los resultados coinciden:
18    Tiempo en GPU 0.0033975989999817102
19    Tiempo en CPU: 0.031145410999670275
20    La GPU es un 916.7% mas rapida
21 -----N = 100000000-----
22    Los resultados coinciden:
23    Tiempo en GPU 0.26329770699976507

```

```

24      Tiempo en CPU: 2.8926681190005183
25      La GPU es un 1098.6% mas rapida

```

Listing A.7: Suma de vectores (Salida)

```

1  import numpy as np
2  import pandas as pd
3  from math import sqrt
4  import warnings
5
6
7  def calor1D(intervalo, f, alpha, beta, t_obj, nt, nx):
8      """Solves heat equation in 1D. t_0 is assumed to be 0.
9      intervalo: (a,b) where a and b are the bounds of the dim
10     f: The heat function for t=0
11     alpha: Heat function in a for t>0
12     beta: Heat function in b for t>0
13     t_obj: Final time
14     nt: Number of points in time, not including t=0
15     nx: Number of internal points in space
16     Returns an np.matrix with the values in the grid.
17     If lambda is greater than 0.5, a message will be shown but
18     """
19
20     dt = t_obj/nt #La matriz va de 0 a nt
21     dx = (intervalo[1] - intervalo[0])/(nx + 1) #La matriz va de 0
22     a nx + 1
23     lam = dt/pow(dx, 2) #Calculamos lambda
24     if(lam > 1/2):
25         print("Warning: dt/dx^2 must be less or equal than 0.5, and
26         it's ", lam, ", the method might not converge.")
27         print("dt = ", dt)
28         print("dx = ", dx)
29
30     resultado = np.zeros(shape=(nt + 1, nx + 2))
31     #Rellenamos datos iniciales
32     for i in range(nt + 1):
33         t = i*dt
34         resultado[i][0] = alpha(t)
35         resultado[i][nx+1] = beta(t)
36     for j in range(nx + 2):
37         x = intervalo[0]+j*dx
38         resultado[0][j] = f(x)
39
40     #Rellenamos el resto de la matriz
41     for i in range(1,nt+1):
42         for j in range(1, nx+1):
43             resultado[i][j] = (1-2*lam)*resultado[i-1][j] + (
44             resultado[i-1][j-1] + resultado[i-1][j+1])*lam

```

Listing A.8: Método numérico para la ecuación del calor en 1D (Código Python)

```

1  import numpy as np

```

```

2 import pandas as pd
3 from math import sqrt
4 import warnings
5 import generateMod as generateMod
6 import pycuda.driver as cuda
7
8
9 def calor1D(intervalo, f, alpha, beta, t_obj, nt, nx):
10     """Solves heat equation in 1D on the GPU. t_0 is assumed to be
11     0.
12     intervalo: (a,b) where a and b are the bounds of the dim
13     f: The heat function for t=0
14     alpha: Heat function in a for t>0
15     beta: Heat function in b for t>0
16     t_obj: Final time
17     nt: Number of points in time, not including t=0
18     nx: Number of internal points in space
19     Returns an np.matrix with the values in the grid.
20     If lambda is greater than 0.5, a message will be shown but
21     """
22
23     dt = t_obj/nt #La matriz va de 0 a nt
24     dx = (intervalo[1] - intervalo[0])/(nx + 1) #La matriz va de 0
25     a nx + 1
26     lam = dt/pow(dx, 2) #Calculamos lambda
27     if(lam > 1/2):
28         print("Warning: dt/dx^2 must be less or equal than 0.5, and
29         it's ", lam, ", the method might not converge.")
30         print("dt = ", dt)
31         print("dx = ", dx)
32
33     resultado = np.zeros(shape=(nt + 1, nx + 2)).astype(np.float32)
34     #Rellenamos datos iniciales
35     for i in range(nt + 1):
36         t = i*dt
37         resultado[i][0] = alpha(t)
38         resultado[i][-1] = beta(t)
39     for j in range(nx + 2):
40         x = intervalo[0]+j*dx
41         resultado[0][j] = f(x)
42
43     #Iniciamos pycuda
44     mod = generateMod.init(["heat1D.cu"])
45     heat1D = mod.get_function("heat1D")
46     #Reservamos memoria y copiamos
47     resultado_gpu = cuda.mem_alloc(resultado.nbytes)
48     cuda.memcpy_htod(resultado_gpu, resultado)
49     #Calculamos cuantos hilos y bloques necesitamos
50     b = nx if nx < 1024 else 1024 #El bloque va a ser de 1024 (
51     siempre que n > 1024)
52     g = nx // 1024 + 1 #Tantos grids como haga falta
53
54     #Rellenamos el resto de la matriz en la gpu
55     for i in range(1,nt+1): #Para cada t
56         heat1D(resultado_gpu, np.int32(i), np.float32(lam), np.

```

```

int32(nx), grid=(g,1,1), block=(b,1,1))
54     cuda.Context.synchronize() #No podemos avanzar hasta que no
    hayamos terminado la linea
55
56     cuda.memcpy_dtoh(resultado, resultado_gpu)
57
58     return resultado

```

Listing A.9: Método numérico para la ecuación del calor en 1D utilizando CUDA (Código Python)

```

1  __device__ int index(int i, int j, int nx) { //Hace la conversion
    de i j a indice
2      return i*(nx+2)+j;
3  }
4
5  __global__ void heat1D(float* resultado, int i, float lam, int nx){
    //n es el tamanno de cada fila
6      //Implementa la formula: resultado[i][j] = (1-2*lam)*resultado
    [i-1][j] + (resultado[i-1][j-1] + resultado[i-1][j+1])*lam
7      int j = blockIdx.x * 1024 + threadIdx.x + 1; //El +1 es porque
    la columna 0 no la hacemos
8      if(1 <= j && j <= nx + 1){ //No queremos que calcule el ultimo
    ni el primero
9          resultado[index(i,j,nx)] = (1-2*lam)*resultado[index(i-1,j,
    nx)] + (resultado[index(i-1,j-1,nx)] + resultado[index(i-1,j+1,
    nx)])*lam;
10     }
11 }

```

Listing A.10: Kernel para calcular el resultado de una fila concreta (supuesto que las anteriores estén ya hechas)

```

1  import numpy as np
2  import pandas as pd
3  from math import sqrt, ceil
4  import warnings
5
6
7  def wave1D(f, g, c, nt, a=0, t_obj=1, n_puntos=1):
8      """Solves wave equation in 1D. t_0 is assumed to be 0.
9      a: The point where we want to know the solution
10     f: The wave function for t=0
11     g: The value of the derivative at t=0
12     t_obj: Final time
13     nt: Number of time slices (non counting 0)
14     c: Constant of the equation
15     Returns an np.matrix with the values in the grid.
16     """
17     dt = t_obj/(nt + 1)
18     dx = c*dt
19     nx = n_puntos + 2*nt #Esto son los puntos que va a tener la
    matriz, luego no devolveremos todos
20
21     resultado = np.zeros(shape=(nt + 1, nx)) #En realidad la matriz
    tendra columnas auxiliares que no devolveremos

```

```
22 #Casos iniciales
23 for j in range(nx):
24     x = a+(j-nt)*dx
25     resultado[0][j] = f(x) + dt*g(x)
26 for j in range(1, nx - 1):
27     x = a+(j-nt)*dx
28     resultado[1][j] = f(x)
29
30 #Rellenamos el resto de la matriz
31 for i in range(2,nt):
32     for j in range(i, nx-i): #Lo calculamos solo en los que
33         #tiene sentido
34         resultado[i][j] = resultado[i-1][j+1] + resultado[i-1][j-1] - resultado[i-2][j]
35
36 return resultado[:,nt:nx-nt]
```

Listing A.11: Método numérico para la ecuación de onda en 1D (Código Python)

Este texto se puede encontrar en el fichero Cascaras/fin.tex. Si deseas eliminarlo, basta con comentar la línea correspondiente al final del fichero TFGTeXiS.tex.

*–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.*

*Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes*

*–Buena está – dijo Sancho –; fírmela vuestra merced.
–No es menester firmarla – dijo Don Quijote–,
sino solamente poner mi rúbrica.*

*Primera parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes*

