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

20 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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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 (2D):

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

Ondas (2D):

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

Calor (1D):

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

Ondas (1D):

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

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. Notación

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.

Por tanto, el dominio de todos nuestros problemas va a ser un rectángulo en dos o tres dimensiones. Sobre este rectángulo, crearemos una malla a partir de unos parámetros que formarán parte de la entrada de los algoritmos, que será el subconjunto finito del dominio donde trabajaremos.

Esta notación es consistente entre los distintos casos, pero las definiciones serán ligeramente distintas dependiendo de las dimensiones del problema.

Tipo I: Una variable temporal y otra espacial

Este es el caso de las ecuaciones de calor y onda lineales. El dominio será $R = \{(x,t) \mid a \leq x \leq b,\ 0 \leq t \leq t_{max}\}$ y solución real del problema la denotaremos por $u(x,t) : [a,b] \times \mathbb{R}^+ \longrightarrow \mathbb{R}$.

Los parámetros que necesitaremos para construir la malla (que los pediremos como entrada del algoritmo) serán los valores que delimitan el rectángulo (a, b y t_{max}), y el número de puntos que habrá en la malla en ambas dimensiones (n_x, n_t). A partir de estos datos podemos definir $\Delta x := \frac{b-a}{n_x-1}$ y $\Delta t := \frac{t_{max}}{n_t-1}$ y los conjuntos $R_x := \{a + i\Delta x | 0 <= i < n_x\} \subset [a,b]$ y $R_t := \{j\Delta x | 0 <= j < n_t\} \subset [0, t_{max}]$. En base a esas definiciones, el conjunto de puntos sobre el que trabajaremos será $R_x \times R_t$.

Salta a la vista que todos los puntos de la malla serán de la forma $(a+i\Delta x, j\Delta t)$, por lo que para simplificar la notación, definiremos[1] $x_i := a + i\Delta x$ y $t_j := j\Delta t$. Con esta notación se cumplen las relaciones $t_0 = 0$, $t_{n_t-1} = t_{max}$, $x_0 = a$ y $x_{n_x-1} = b$.

Para seguir simplificando la notación, cuando queramos evaluar cualquier función f en los puntos de la malla, podremos escribirlo de la forma $f_{i,j} := f(x_i, t_j) = f(a + i\Delta x, j\Delta t)$.

La función que calcularemos y utilizaremos para aproximar u será $U(x,t) : R_x \times R_t \longrightarrow \mathbb{R}$.

Y por último, como queremos aplicar métodos basados en las diferencias finitas, necesitaremos aproximar las distintas derivadas de la función por un cociente incremental. Para simplificar eso, utilizaremos la siguiente notación:

[1] Nótese que, aunque $(x_i, t_j) \in R \Leftrightarrow 0 \leq i < nx$ y $0 \leq j < n_t$, la definición es coherente para cualesquiera i y j, y aunque $(x_i, t_j) \notin R$, podemos seguir evaluando la solución (o su aproximación) en estos puntos siempre y cuando podamos asegurar la existencia y unicidad de la solución en un conjunto R' tal que $(x_i, t_j) \cup R \subset R'$.

$$U^x(x,t) := \frac{U(x+\Delta x,t)-U(x,t)}{\Delta x} \tag{1.1}$$

$$U^{\bar{x}}(x,t) := \frac{U(x,t)-U(x-\Delta x,t)}{\Delta x} \tag{1.2}$$

$$U^{\hat{x}}(x,t) := \frac{1}{2}[U^x(x,t)+U^{\bar{x}}(x,t)] \tag{1.3}$$

A estas funciones también podemos el convenio descrito arriba, y escribirlas sustituir x y t por los subíndices pertinentes, teniendo en cuenta que si $x = x_i$, $x \pm \Delta x = x_{i\pm1}$ (y de manera análoga, con la variable t).

Cabe destacar un caso de especial interés que usaremos más adelante, y es aplicar (en cualquier orden) a la función U (1.1) y (1.2), con lo que obtenemos la siguiente:

$$U^{x\bar{x}}_{i,j} = U^{\bar{x}x}_{i,j} := \frac{1}{\Delta x^2}[U_{x+1,j} - 2U_{i,j} + U_{i-1,j}] \tag{1.4}$$

Como es natural, las cuatro funciones descritas arriba pueden definirse de manera completamente análoga sobre la variable t en lugar de x.

Tipo II: Una variable temporal y dos espaciales

Este es el caso de las ecuaciones del calor y ondas y, de manera muy parecida al apartado anterior, definiremos el dominio como $R = \{(x,y,t) \mid a \leq x \leq b, c \leq y \leq d, 0 \leq t \leq t_{max}\}$.

Los parámetros para construir la malla serán los mismos que en el apartado anterior, añadiéndoles los límites de la nueva dimensión (c,d) y el número de puntos que queremos en esta dimensión (n_y).

Las definiciones del apartado anterior de los incrementos de x y t siguen siendo válidas, pero tenemos que añadir la definición de $\Delta y := \frac{d-c}{n_y-1}$ y el conjunto $R_y := \{c + i\Delta x | 0 <= k < n_y\} \subset [c,d]$. Con todas las definiciones que hemos hecho, el subconjunto finito sobre el que estamos trabajando (la malla) es $R_x \times R_y \times R_t$.

La solución exacta será denotada por $u(x,t) : [a,b] \times \mathbb{R}^+ \longrightarrow \mathbb{R}$, y la función utilizada para aproximarla será $U(x,y,t) : R_x \times R_y \times R_t \longrightarrow \mathbb{R}$.

Como en el apartado anterior, al evaluar una función en un punto de la malla utilizaremos subíndices (en este caso i, k y j) para identificar al punto. Por último se modifican ligeramente las ecuaciones (1.1), (1.2), (1.3) y (1.4) para que tengan sentido en tres dimensiones.

Tipo III: Dos variables espaciales

Este caso solo se corresponde con la ecuación de Laplace en el plano,y utilizaremos una notación muy parecida al la que se emplea en el Tipo I, pero limitando ambas variables por intervalos $[a,b]$ y $[c,d]$ como hicimos en el Tipo II.

COMENTARIO: Las definiciones necesarias para el Tipo III son prácticamente iguales que las de los tipos anteriores y por eso he decidido no extenderme más en esto. No obstante, si lo ves necesario puedo hacer una construcción totalmente completa (como en el tipo I) o desarrollar un poco más (como en el Tipo II).

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 máquina[2]

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 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.

{**TODO TODO TODO:** Completar con qué más hago en el TFG}

[2]Si 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 cómputos que se requieren para procesar estos se puede ejecutar de manera simultánea en paralelo.

Dada la naturaleza de las tarjetas gráficas, resulta natural el utilizarlas para implementar algoritmos que puedan beneficiarse de este paralelismo. Con ese fin, NVIDIA desarrolló el lenguage de programación CUDA[1], 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

[1]Puede consultarse el manual completo de CUDA en este *enlace*

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 se 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.

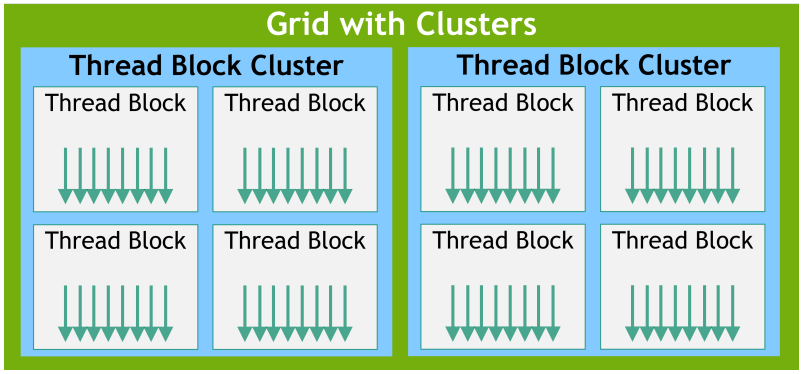

Figura 2.1: Cuadrícula de bloques de hilos

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

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

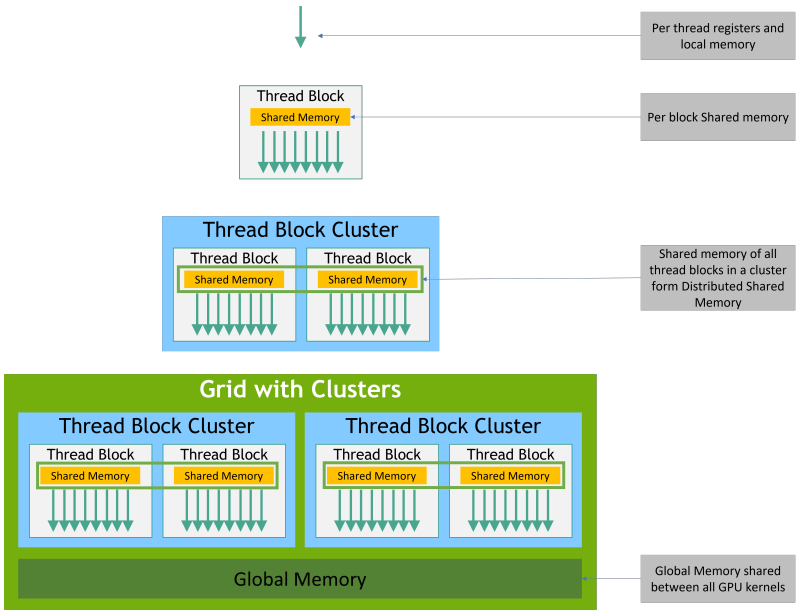

Figura 2.2: Memoria compartida entre elementos

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 grandes[3], 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 utilicemos, 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, y es que cuando el tamaño del vector es suficientemente grande, el algoritmo es unas 10 veces más rápido.

[3]En 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, tenemos la ecuación

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{3.1}$$

Se pueden pedir diferentes condiciones iniciales para asegurar la existencia y unicidad de la ecuación 3.1, pero nosotros utilizaremos en concreto las condiciones

$$\begin{cases} u(x,0) = f(x), & a \leq x \leq b, \\ u(a,t) = \alpha(t), & 0 < t, \\ u(b,t) = \beta(t), & 0 < t. \end{cases} \tag{3.2}$$

El problema de valor inicial que hemos definido tiene como dominio un rectángulo con uno de sus lados abierto hacia el infinito. Si fijamos $T > 0$ tenemos ahora el rectángulo con el que trabajaremos. Definimos pues B_T como la frontera del rectángulo y D_T el interior de este, y por último definimos $D = D_T \cup B_T$.

3.1.1. Existencia y unicidad

Antes de estudiar la existencia y unicidad de la solución, necesitamos definir un concepto.

Definicion 3.1.1 (Función continua a trozos)**.** *Una función es continua a trozos si es continua en todos sus puntos salvo en una cantidad finita.*

Ahora enunciamos unas condiciones para la unicidad de las soluciones

Teorema 3.1.1 (Unicidad). *Sean u y v soluciones de la ecuación 3.1 en D_T continuas en D, si $u = v$ en B_T entonces $u = v$ en D*

Teorema 3.1.2 (Unicidad extendida). *Sean u y v soluciones de la ecuación 3.1 en D_T continuas a trozos en D con una cantidad finita de discontinuidades acotadas, si $u = v$ en B_T (excepto los puntos de discontinuidad) entonces $u = v$ en D*

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], p.22 □

Teorema 3.1.3 (Existencia y unicidad). *Sean f, α y β funciones continuas a trozos, la función*

$$u(x,t) = \int_a^b \theta(x-\xi,t) - \theta(x+\xi,t) f(\xi) d\xi - 2\int_0^t \frac{\partial \theta}{\partial x}(x, t-\tau)\alpha(\tau)d\tau + 2\int_0^t \frac{\partial \theta}{\partial x}(x-1, t-\tau)\beta(\tau)d(\tau) \quad (3.3)$$

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

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

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

Es la única solución acotada del problema del valor inicial

$$\begin{cases} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} & a < x < b, \quad 0 < t, \\ u(x,0) = f(x), & a < x < b, \\ u(a,t) = \alpha(t), & 0 < t, \\ u(b,t) = \beta(t), & 0 < t. \end{cases} \quad (3.4)$$

Demostración. Puede verse en [1] que en efecto 3.3 es solución de la ecuación del calor, 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_T. □

Ahora podemos asegurar que 3.4 tiene una única solución, por lo que podemos proceder a aproximarla con un método de diferencias finitas.

3.1.2. Aproximación de la solución[1]

Utilizando la notación descrita en la sección 1.3.1, aproximaremos la ecuación 3.1 por el cociente incremental

$$u_t(x,t) = u_{x\bar{x}}(x,t) \Rightarrow \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{1}{\Delta x^2}[u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)] \tag{3.5}$$

Ahora, si nos ceñimos a una malla de puntos como definimos en 1.3.1, siendo Δt y Δx el incremento entre los puntos de cada dimensión, tendremos:

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{1}{\Delta x^2}[u_{i+1,j} - 2u_{i,j} + u_{i-1,j}] \qquad \forall j \in \mathbb{N}^+ \tag{3.6}$$

Lo que, despejando y definiendo $\lambda \equiv \frac{\Delta t}{\Delta x^2}$ nos lleva a la fórmula explícita

$$u_{i,j+1} = (1-2\lambda)u_{i,j} + \lambda(u_{i+1,j} + u_{i-1,j}) \qquad \forall j \in \mathbb{N}^+ \tag{3.7}$$

Analicemos un poco la fórmula 3.7. Queremos calcular una aproximación de la solución en los puntos de la malla para $a \leq x \leq b$ y $0 \leq t$.

Gracias al PVI tenemos los valores de $u_{i,0}$, $u_{0,j}$ y $u_{N,j}$ $\forall i,j >= 0$ (siendo N tal que $b = a + N\Delta x$). Con esos valores está claro que podemos utilizar la fórmula 3.7 en todos los puntos de la malla.

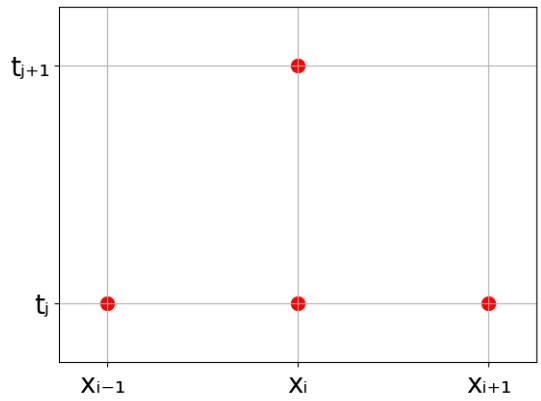

Figura 3.1: Representación en la malla del esquema 3.7

Para estudiar la convergencia de 3.7, primero estudiaremos el siguiente lema:

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

$$u_t(x,t) - u_{x,\bar{x}}(x,t) = \tau(x,t) \tag{3.8}$$

donde t es el error de truncamiento, definido como

$$\tau(x,t) = \frac{\Delta t}{2}\frac{\partial^2 u}{\partial t^2} + \frac{\Delta x^2}{12}\frac{\partial^4 u}{\partial x^4} + \xi_{\Delta x,\Delta t}. \tag{3.9}$$

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

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

$$\xi_{\Delta x,\Delta t} \xrightarrow[\Delta x\to 0,\Delta t\to 0]{} 0$$

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

$$\begin{aligned} u(x,t+\Delta t) = u(x,t) + \frac{\partial u}{\partial t}\Delta t + \frac{1}{2}\frac{\partial^2 u}{\partial t^2}\Delta t^2 + \xi^1_{\Delta x,\Delta t}\Delta t^2 \Rightarrow \\ u_t(x,t) - \frac{\partial u}{\partial t} = \frac{\Delta t^2}{2}\frac{\partial^2 u}{\partial t^2}\Delta t + \xi^1_{\Delta x,\Delta t}\Delta t \end{aligned}$$

Siendo ξ^1 un número con las mismas propiedades que ξ. Despejando la última ecuación, obtenemos el primer sumando de 3.8, así como el de la ecuación 3.9, y nos sobra $-\frac{\partial u}{\partial t}$ a la izquierda y $\xi^1_{\Delta x,\Delta t}\Delta t$ a la derecha.

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

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

y

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

Siendo $\xi^2_{\Delta x,\Delta t}$ y $\xi^3_{\Delta x,\Delta t}$ números que cumplen las mismas propiedades que ξ. Ahora, si sumamos las dos últimas ecuaciones, ya que se nos cancelan los términos con exponente impar, obtenemos

$$u_{x,\bar{x}} - \frac{\partial^2 u}{\partial x^2} = \frac{1}{12}\frac{\partial^2 u}{\partial x}\Delta x^2 + (\xi^2_{\Delta x,\Delta t} + \xi^3_{\Delta x,\Delta t})\Delta x^2$$

Si restamos todo, tendiendo en cuenta que $\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0$, obtenemos precisamente la igualdad 3.8, siendo $\xi_{\Delta x,\Delta t} = \xi^1_{\Delta x,\Delta t}\Delta t - (\xi^2_{\Delta x,\Delta t} + \xi^3_{\Delta x,\Delta t})\Delta x^2$, por lo que se confirma que tiende a 0 cuando los incrementos tienden a 0. □

Teorema 3.1.4. *Si $0 < \lambda < \frac{1}{2}$, el método numérico 3.7 es convergente, o dicho de otra forma, si $\epsilon_{i,j}$ es el error de aproximación del método, $\sup_{i,j}\epsilon(i,j) \to 0$ si $\Delta t, \Delta x \to 0$ y el error inicial tiende a 0.*

Demostración. Teniendo en cuenta el lema anterior, si ahora repetimos las cuentas de 3.5 obtendríamos 3.7 pero añadiendo el sumando $\Delta t\tau_{i,j}$, por lo que cada vez que utilizamos esa fórmula estamos añadiendo ese error. Por ello, podemos deducir que

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

Ahora, si definimos

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

tenemos que

$$E_j + 1 \leq E_j + \Delta t\tau$$

por tanto, mediante una inducción trivial tenemos que

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

Con esto ya tenemos el resultado, 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. □

3.2. Caso bidimensional

En el caso bidimensional, la ecuación que tenemos será

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{3.10}$$

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.

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

Tenemos la siguiente ecuación:

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0 \tag{4.1}$$

a la que le pediremos las condiciones iniciales

$$\begin{cases} u(x,0) = f(x) & -\infty < x < \infty \\ \frac{\partial u(x,0)}{\partial t} = g(x) & -\infty < x < \infty \end{cases} \tag{4.2}$$

Como es costumbre, lo primero que tenemos que hacer es probar la existencia y unicidad de la solución antes de aproximarla

4.1.1. Existencia y unicidad[1]

Si hacemos el cambio de variable $\xi = x - ct,\ \eta = x + ct$. Aplicando la regla de la cadena, obtenemos

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

y

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

[1]Esta demostración puede encontrarse en el artículo de [3]

Luego, si sustituimos en la ecuación 4.1 con las igualdades anteriores, obtenemos que una función es solución de 4.1 si y solo sí cumple

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

Es trivial (integrando dos veces) observar que esta la solución a esta última ecuación será de la forma

$$u(x,t) = P(\xi) + Q(\eta) = P(x-ct) + Q(x+ct) \tag{4.3}$$

para unas funciones P y Q arbitrarias. Ahora, si utilizamos las condiciones iniciales 4.2 tenemos las relaciones

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

Ahora, si integramos la segunda relación en el intervalo $[0, x]$, y la sumamos y restamos a la primera y definimos $K = \frac{1}{2}[P(0) - Q(0)]$, obtenemos cómo son las funciones

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

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

Luego, simplemente sustituyendo en 4.3, obtenemos que

$$u(x,t) = \frac{1}{2}[f(x+ct) + f(x-ct)] + \frac{1}{2c} \int_{x-ct}^{x+ct} g(\zeta) d\zeta. \tag{4.4}$$

Teorema 4.1.1 (Existencia y unicidad)**.** *Si f y g son funciones integrables, la ecuación 4.1 con los valores iniciales 4.2 tiene solución y además esta es única.*

Demostración. La función definida en 4.4 es, por construcción, solución de la ecuación diferencial, luego la existencia está demostrada. Demostrar la unicidad es sencillo, pues si suponemos que $v(x,t)$ es solución de 4.1, llegamos a la conclusión de que tiene que ser de la forma 4.4, ya que todos los argumentos hechos en esta subsección son reversibles. □

4.1.2. Aproximación de la solución[2]

Utilizando la notación descrita en la sección 1.3.1, aproximaremos la ecuación 4.1 por el cociente incremental

$$u_{t\bar{t}}(x,t) - c^2 u_{x\bar{x}}(x,t) = 0 \Rightarrow \frac{1}{\Delta t^2}[u(x,t+\Delta t) - 2u(x,t) + u(x,t-\Delta t)] = \left(\frac{c}{\Delta x}\right)^2 [u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)]$$

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

Si ahora nos ceñimos a la malla de puntos de la que hablamos en 1.3.1, tenemos

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

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

Lo que, tras combinar elementos, nos queda en

$$u_{i,j+1} = 2\left[1 - \left(c\frac{\Delta t}{\Delta x}\right)^2\right] u_{i,j} + \left(c\frac{\Delta t}{\Delta x}\right)^2 [u_{i+1,j} + u_{i-1,j}] - u_{i,j-1}. \tag{4.5}$$

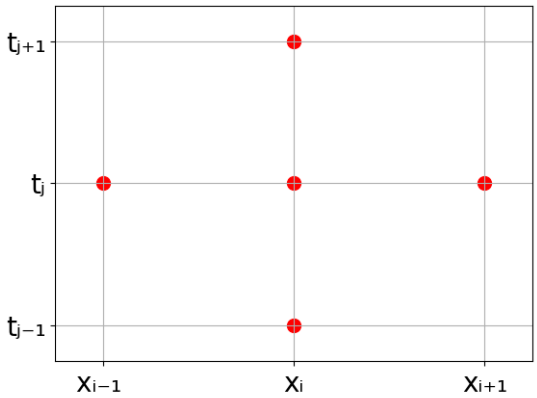

Figura 4.1: Representación en la malla del esquema 4.5

Esto indica que, para calcular $u_{i,j}$ para cualquier tiempo $j \in \mathbb{N}$ necesitamos conocer el valor de la función en los dos tiempos anteriores. Esto no será un problema, pues los valores para $u(x,0) = f(x)$ por el problema del valor inicial.

Calcular los valores para $t = \Delta t$ tampoco es difícil, pues $u(x, \Delta t) \equiv u(x,0) + \Delta t u_t(x,0) \equiv f(x) + \Delta t g(x)$.

Si definimos $\lambda = c\frac{\Delta t}{\Delta x}$, la convergencia del esquema numérico depende exclusivamente de su valor. Si $\lambda > 1$, el método diverge [3]. Por otro lado, si $\lambda <= 1$, el esquema converge.

Lo probaremos solo para $\lambda = 1$ (que es el caso que usaremos), pero puede encontrarse en [2] una demostración para $0 < \lambda <= 1$.

Teorema 4.1.2. *Si $\lambda = 1$ y $f, g \in \mathscr{C}^2(\mathbb{R})$, la aproximación dada por 4.5 converge a la solución de 4.1 cuando $\Delta x = c\Delta t \longrightarrow 0$.*

Demostración. Primero definimos el siguiente valor.

$$D_{i,j} := u_{i,j} - u_{i-1,j-1} \qquad \forall 1 <= j.$$

Ahora, como $\lambda = 1$ podemos rescribir la ecuación 4.5 como

$$u_{i,j+1} - u_{i-1,j} = u_{i+1,j} - u_{i,j-1} \Rightarrow D_{i,j+1} = D_{i+1,j}.$$

[3] Esto no lo probaremos, pero puede encontrarse una demostración en [2], p. 487.

Aplicando ahora esa fórmula de manera recursiva, podemos observar la siguiente relación

$$D_{i,j+1} = D_{i+k,j-k+1} \qquad \forall 0 <= k <= j. \tag{4.6}$$

Además, por como ha sido definido D, es fácil observar que para cualesquiera i, j, se tiene que

$$\sum_{k=0}^{j} D_{i-k,j-k+1} = u_{i,j+1} - u_{i-j-1,0}$$

Despejando en la última ecuación podemos obtener la fórmula explícita

$$u_{i,j+1} = u_{i-j-1,0} + \sum_{k=0}^{j} D_{i-k,j-k+1}$$

Pero, primero teniendo en cuenta que $u_{i-j-1,0} = f_{i-j-1}$, y gracias a 4.6, podemos sustituir los sumandos del sumatorio por términos más simples, y obtener

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

Por otro lado, ya vimos que $u_{i,1} = f_i + \Delta t g_i$, luego en última instancia, la ecuación se nos queda como

$$u_{i,j+1} = f_{i-j-1} + \Delta t \sum_{k=0}^{j} g_{i+j-2k} + \sum_{k=0}^{j} (f_{i+j-2k} - f_{i+j-2k-1}) \tag{4.7}$$

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 4.7 con todas las igualdades que hemos obtenido, tendremos

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

Que, pasando al límite se nos queda en

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

Pero el final de la última serie de igualdades es exactamente 4.4, luego hemos demostrado que la aproximación tiende a la solución exacta de la ecuación. □

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

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{5.1}$$

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

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 comparalos 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 rellenando la tabla con la fórmula de abajo a arriba.

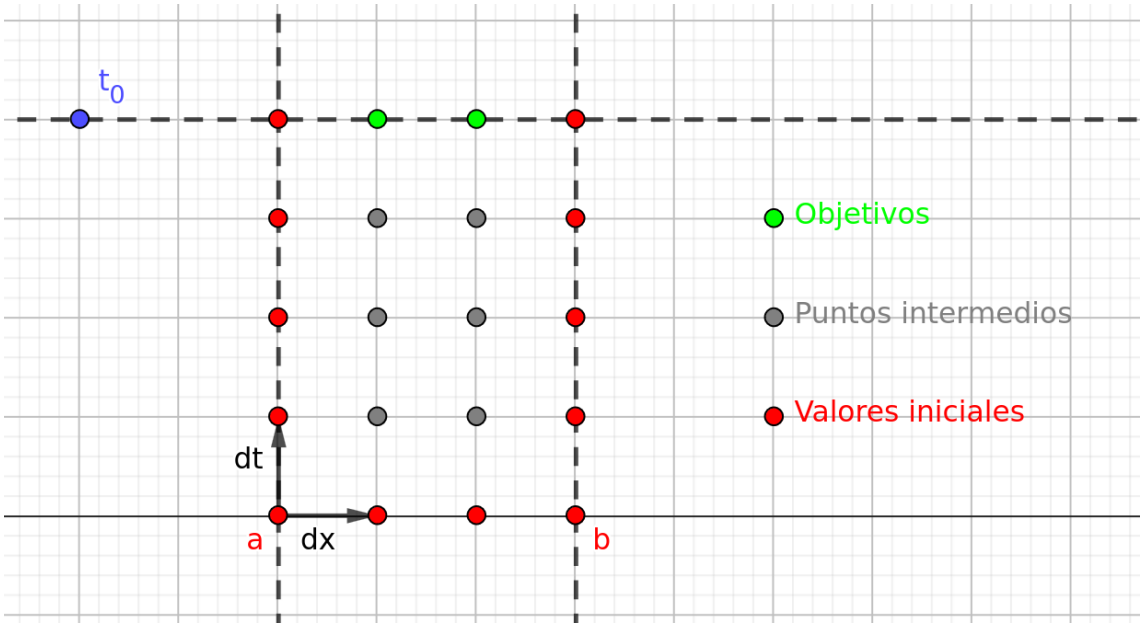

Figura 6.1: Diagrama que muestra qué puntos intermedios necesitamos calcular para obtener el resultado deseado

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 algoritmos (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 $\lambda = 1$.

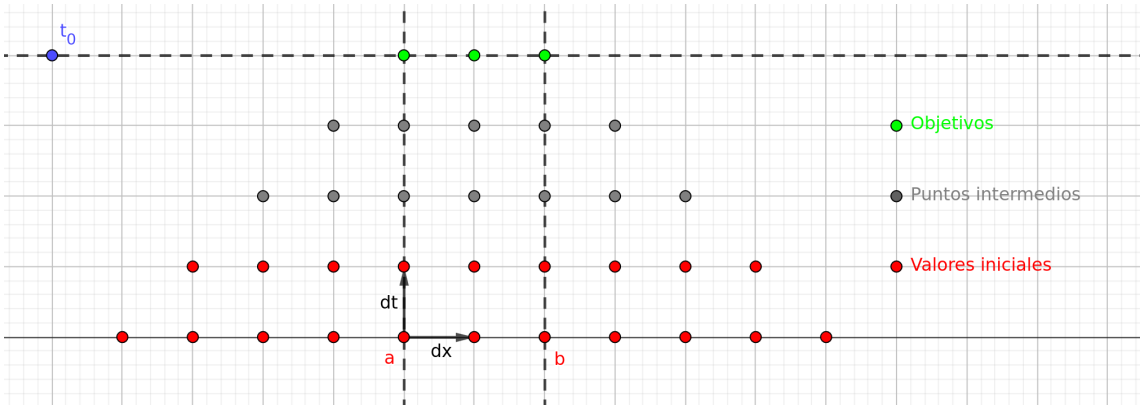

Figura 6.2: Diagrama que muestra qué puntos intermedios necesitamos calcular para obtener el resultado deseado

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

Capítulo 7

Algoritmos en GPU

Resumen: En este capítulo implementaremos los mismos métodos numéricos que en el capítulo 6, 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 3.7 y 4.5). 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 celebro 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

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

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.getcwd() + os.sep + "CUDA" + os.sep + file, 'r').read() + '\n'
    return SourceModule(code)



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

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

```python
import generateMod as generateMod
import pycuda.driver as cuda
import numpy as np
import sys


mod = generateMod.init(["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))
```

Listing 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]);
}
```

Listing 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".
```

Listing A.4: Hola Mundo (Salida)

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

mod = generateMod.init(["vectorAdd.cu"])
vectorAdd = mod.get_function("vectorAdd")
def add_random_vects(n, out):
    #Tiempo en CPU (Contando reservar memoria)
    start = timer()
    a = np.random.randn(n).astype(np.float32)
    b = np.random.rand(n).astype(np.float32)
    c2 = np.zeros(n).astype(np.float32)
    np.add(a,b,out=c2)
    CPUt = timer()-start
    #Tiempo en GPU (Contando reservar y copiar los datos al
    dispositivo)
    start = timer()
    c1 = np.zeros(n).astype(np.float32)
    a_gpu = cuda.mem_alloc(a.nbytes)
    b_gpu = cuda.mem_alloc(b.nbytes)
    c_gpu = cuda.mem_alloc(c1.nbytes)
    n_gpu = cuda.mem_alloc(np.int32(n).nbytes)
    cuda.memcpy_htod(a_gpu, a)
    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
```

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

Listing A.7: Suma de vectores (Salida)

```python
import numpy as np
import pandas as pd
from math import sqrt
import warnings


def calor1D(intervalo, f, alpha, beta, t_obj, nt, nx):
    """Solves heat equation in 1D. t_0 is assumed to be 0.
    intervalo: (a,b) where a and b are the bounds of the dim
    f: The heat function for t=0
    alpha: Heat function in a for t>0
    beta: Heat function in b for t>0
    t_obj: Final time
    nt: Number of points in time, not including t=0
    nx: Number of internal points in space
    Returns an np.matrix with the values in the grid.
    If lambda is greater than 0.5, a message will be shown but
    """

    dt = t_obj/nt #La matriz va de 0 a nt
    dx = (intervalo[1] - intervalo[0])/(nx + 1) #La matriz va de 0 a nx + 1
    lam = dt/pow(dx, 2) #Calculamos lambda
    if(lam > 1/2):
        print("Warning: dt/dx^2 must be less or equal than 0.5, and it's ", lam, ", the method might not converge.")
        print("dt = ", dt)
        print("dx = ",dx)

    resultado = np.zeros(shape=(nt + 1, nx + 2))
    #Rellenamos datos iniciales
    for i in range(nt + 1):
        t = i*dt
        resultado[i][0] = alpha(t)
        resultado[i][-1] = beta(t)
    for j in range(nx + 2):
        x = intervalo[0]+j*dx
        resultado[0][j] = f(x)


    #Rellenamos el resto de la matriz
    for i in range(1,nt+1):
        for j in range(1, nx+1):
            resultado[i][j]  = (1-2*lam)*resultado[i-1][j] + (resultado[i-1][j-1] + resultado[i-1][j+1])*lam

    return resultado
```

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

```python
import numpy as np
```

```
import pandas as pd
from math import sqrt
import warnings
import generateMod as generateMod
import pycuda.driver as cuda


def calor1D(intervalo, f, alpha, beta, t_obj, nt, nx):
    """Solves heat equation in 1D on the GPU. t_0 is assumed to be 0.
    intervalo: (a,b) where a and b are the bounds of the dim
    f: The heat function for t=0
    alpha: Heat function in a for t>0
    beta: Heat function in b for t>0
    t_obj: Final time
    nt: Number of points in time, not including t=0
    nx: Number of internal points in space
    Returns an np.matrix with the values in the grid.
    If lambda is greater than 0.5, a message will be shown but
    """

    dt = t_obj/nt #La matriz va de 0 a nt
    dx = (intervalo[1] - intervalo[0])/(nx + 1) #La matriz va de 0 a nx + 1
    lam = dt/pow(dx, 2) #Calculamos lambda
    if(lam > 1/2):
        print("Warning: dt/dx^2 must be less or equal than 0.5, and it's ", lam, ", the method might not converge.")
        print("dt = ", dt)
        print("dx = ",dx)

    resultado = np.zeros(shape=(nt + 1, nx + 2)).astype(np.float32)
    #Rellenamos datos iniciales
    for i in range(nt + 1):
        t = i*dt
        resultado[i][0] = alpha(t)
        resultado[i][-1] = beta(t)
    for j in range(nx + 2):
        x = intervalo[0]+j*dx
        resultado[0][j] = f(x)

    #Iniciamos pycuda
    mod = generateMod.init(["heat1D.cu"])
    heat1D = mod.get_function("heat1D")
    #Reservamos memoria y copiamos
    resultado_gpu = cuda.mem_alloc(resultado.nbytes)
    cuda.memcpy_htod(resultado_gpu, resultado)
    #Calculamos cuantos hilos y bloques necesitamos
    b = nx if nx < 1024 else 1024 #El bloque va a ser de 1024 (siempre que n > 1024)
    g = nx // 1024 + 1 #Tantos grids como haga falta


    #Rellenamos el resto de la matriz en la gpu
    for i in range(1,nt+1): #Para cada t
        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
    tiene sentido
33              resultado[i][j]  = resultado[i-1][j+1] + resultado[i
    -1][j-1] - resultado[i-2][j]
34
35      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