UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Simulación de ondas estacionarias mediante computación paralela y software estándar de computación científica

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Moinelo Torres, Mateo José

Tutor/a: Román Moltó, José Enrique

CURSO ACADÉMICO: 2021/2022

Resumen

Las ondas estacionarias son perturbaciones oscilatorias en las que unos determinados puntos permanecen inmóviles mientras el resto se desplazan de posición. Un ejemplo de este fenómeno serían las figuras de Chladni, unos patrones naturales que surgen al esparcir un material granulado en una superficie, y someter esta superficie a una onda acústica. Esto provoca que dicho material forme unas determinadas formas dependiendo de que frecuencia se utilice. La forma y el patrón que estos dibujos tendrán se puede determinar empleando técnicas y métodos de computación científica, y actualmente existen códigos secuenciales que llevan a cabo dichos procedimientos. En este estudio se pretende implementar una solución que pueda llevar a cabo la resolución de este problema empleando herramientas de computación paralela, y analizar su rendimiento y su escalabilidad, con el fin de comprobar la eficiencia de la solución desarrollada.

Palabras clave: computación científica, computación paralela, figuras de Chladni, ondas estacionarias, cálculo de autovalores, matrices dispersas, SLEPc

Resum

Les ones estacionàries són pertorbacions oscil·latòries en les quals uns determinats punts romanen immòbils mentre la resta es desplacen de posició. Un exemple d'aquest fenomen serien les figures de Chladni, uns patrons naturals que sorgeixen en escampar un material granulat en una superfície, i sotmetre aquesta superfície a una ona acústica. Això provoca que aquest material forme unes determinades formes depenent del fet que freqüència s'utilitze. La forma i el patró que aquests dibuixos tindran es pot determinar emprant tècniques i mètodes de computació científica, i actualment existeixen codis seqüencials que duen a terme aquests procediments. En aquest estudi es pretén implementar una solució que pot dur a terme la resolució d'aquest problema emprant eines de computació paral·lela, i analitzar el seu rendiment i la seua escalabilitat, comparant-lo amb altres solucions seqüencials ja existents, amb la finalitat de comprovar l'eficiència de la solució desenvolupada.

Paraules clau: computació científica, computació paral·lela, figures de Chladni, ones estacionàries, càlcul d'autovalors, matrius disperses, SLEPc

Abstract

Stationary waves are oscillatory perturbations in which specific points remain immobile while the rest keep moving. An example of this phenomenon are the Chladni figures, a series of natural patterns that appear when a grainy material is scattered across a surface, and this surface is affected by by an acoustic wave. This causes that such material can create different shapes depending on which frequency is being used on the wave. The shape and pattern of these drawings can be determined by making use of computational science techniques and methods, and nowadays, there are sequential codes that can run these procedures. In this study, the purpose is implementing a solution that can solve this problem by making use of parallel computing tools, and analyze its performance and its scalability, in order to check the efficiency of the developed solution.

Key words: computational science, parallel computing, Chladni figures, stationary waves, eigenvalue calculation, sparse matrix, SLEPc

Índice general

Apéndices

CAPÍTULO 1
Introducción

Alrededor de 1787, el físico y músico alemán Ernst Florens Friedrich Chladni [1] descubrió que, al someter una placa (o malla) metálica a una onda estacionaria, con la cuerda de un violín, esta hacía sonidos diferentes dependiendo del punto de contacto entre la placa y la cuerda. Además, cuando se esparcía una sustancia polvorienta sobre la placa, esta formaba figuras distintas y muy llamativas en cada caso . Estos patrones pasaron a conocerse como figuras de Chladni, y llamaron la atención en la comunidad científica. Algunas de estas formas se pueden ver en la Figura 1.1.

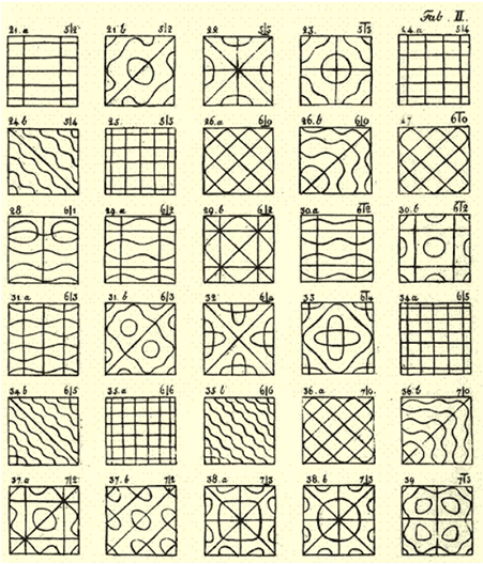


Figura 1.1: Figuras originales obtenidas por Chladni
(Imagen extraída de https://kripkit.com/ernst-chladni).

Debido a la excepcionalidad de su naturaleza, muchos trataron de descubrir el origen de estas formas y como se podían recrear. A pesar de esto, los cálculos de este fenómeno resultaron ser demasiado complicados y complejos durante casi un siglo. El primer modelo matemático fue formulado por Sophie Germain más de 20 años después, y más tarde fue retocado y mejorado por los matemáticos Joseph-Louis Lagrange y Siméon Denis Poisson. Sin embargo, el modelo definitivo sería concebido en 1850 por Gustav Robert

Kirchhoff, cuando publicó uno de los artículos más importantes en relación a la teoría de la elasticidad.

En este artículo, Kirchhoff explicaba que las distintas figuras que se podían formar correspondían a los autovalores y autovectores de la ecuación biarmónica, con condiciones de frontera libre. Más adelante, en 1909, Walther Ritz logró emplear estos descubrimientos para llevar a cabo la primera representación de las figuras de Chladni [2].

Actualmente, existen una gran cantidad de fenómenos que tienen mucha importancia en el mundo científico y que se pueden resolver de forma similar al de las figuras de Chladni, y una gran cantidad de recursos y de tiempo va en intentar simular dichos fenómenos de la forma más precisa y eficiente posible. Existen una gran variedad de técnicas y herramientas que se suelen emplear en este tipo de simulaciones, las cuales se emplean en este trabajo, que están englobadas en dos grandes ámbitos: la **computación científica**, y la **computación paralela**.

La **computación científica** es un campo de estudio que consiste en el uso de computadores para resolver todo tipo de problemas científicos. Esta destaca por no ser una ciencia cerrada, ya que requiere del conocimiento de otras ciencias más amplias [3]. En concreto, para poder practicar la computación científica efectivamente, hay que aplicar conceptos de: matemáticas, con el fin de conocer los métodos matemáticos y la fórmulas relacionadas con el fenómeno a estudiar; informática, con el fin de poder desarrollar el software y emplear todos los sistemas necesarios para poder resolver los problemas computacionalmente complejos que se presenten; y por supuesto, la disciplina asociada al suceso que se quiera estudiar, con el fin de poder entender correctamente que se quiere analizar. Esta última suele ser ciencias como la biología o la física.

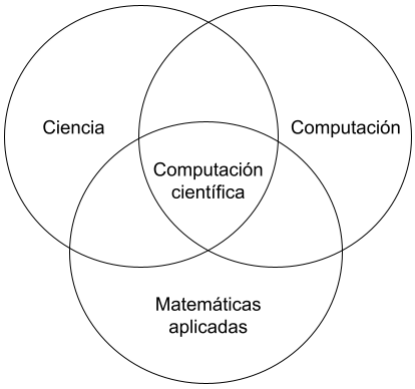


Figura 1.2: La computación científica consiste en la combinación de matemáticas, informática, y la disciplina de la aplicación.

El uso más frecuente de la computación científica es el de las simulaciones por computadora. Tanto en los estudios científicos como en el desarrollo de nuevos productos, es extremadamente importante llevar a cabo simulaciones de sistemas físicos, con el fin de obtener datos precisos que reflejan la realidad y que se tengan en cuenta todos los factores, de forma que luego no surjan efectos imprevistos. Esto es por ejemplo, cuando se quiere diseñar un coche y estudiar su resistencia en accidentes, o comprobar la resisten-

cia de un puente frente a los posibles fenómenos naturales, como tornados o terremotos sísmicos.

Es en este tipo de simulaciones donde se ve la fortaleza de la computación científica. Anteriormente, para llevar a cabo un estudio de un fenómeno físico, las únicas opciones eran teorizar, tratar de observar un caso que ocurriera de forma natural o llevar a cabo un experimento controlado. Sin embargo esto podía ser muy costoso, y en algunos casos imposible (por ejemplo, no se puede realizar un experimento real con un terremoto, ni se puede permitir observar un accidente de avión después de lanzarlo a producción y ser utilizado). En cambio, con la computación científica, lo único necesario es disponer de la tecnología lo suficientemente potente para llevar a cabo los estudios que se requieran, que la mayoría de organizaciones e individuos se pueden permitir en la sociedad actual, en la que la tecnología está ampliamente distribuida.

Aún así, debido a la complejidad de los cálculos que suelen llevar este tipo de proyectos y con el fin de reducir el tiempo que pueden requerir, en la práctica de la computación científica se suele aplicar el uso de la **computación paralela**. Esta consiste en una forma de computación en la que muchos procesos son ejecutados al mismo tiempo de forma coordinada, comunicándose entre ellos. Se basa en el principio de que las operaciones de gran escala se pueden reducir en otras tareas de menor tamaño que pueden ejecutarse al mismo tiempo. De esta forma, el tiempo de ejecución total de un código se puede reducir considerablemente.

En el ámbito del la computación paralela, se suele referir como *granularidad* al tamaño de las distintas tareas en las que se puede dividir la operación final. Estas tareas son ejecutadas al mismo tiempo en unidades físicas distintas de procesamiento, por lo que se denominan *procesos* o *hilos*, de forma que en cada unidad hay un proceso distinto. Estos procesos pueden tener *memoria compartida* (comparten la misma copia de la información), o usar *memoria distribuida* (cada proceso guarda una parte de los datos). Debido a que los procesos tendrán dependencias entre ellos, ya sea porque una tarea dependa de los cálculos de otra, o porque compartan la misma información, es necesario emplear *protocolos de comunicación* para coordinarlos y garantizar la sincronización entre ellos. Por otro lado, la suma del tiempo de calculo de un algoritmo, más los tiempos de comunicación y espera entre procesos, se denomina *tiempo de ejecución paralelo*. Esto depende de factores como el tiempo de espera de un proceso para que termine otro cuyo resultado necesita [4].

La asignación de las tareas a los distintos procesos se conoce como *schedulling*, y la asignación equilibrada de trabajo a los distintos procesos se denomina *balanceo de carga*,

que asegura que ningún proceso ralentiza al resto. Una forma frecuente con la que se lleva a cabo el schedulling es mediante una cola, denominada comúnmente como *cola de trabajos*. En esta se guardan las distintas tareas, y a medida que un proceso queda libre, se le asigna la primera tarea de la cola. Las tareas se suelen guardar por orden de prioridad, y el criterio de prioridad puede variar dependiendo de la implementación. Por ejemplo, los casos más comunes son las colas donde la primera tarea que se asigna es la primera que llega a la cola (cola FIFO, *first-in-first-out*), o la inversa, donde tiene mayor prioridad la última tarea que llega (cola LIFO, *last-in-first-out*) [5].

Todo estos factores mencionados se han de tener en cuenta para garantizar que el paralelismo se está aplicando correctamente y de la forma más eficiente posible. La mejora de tiempo alcanzada con una ejecución paralela con respecto a una secuencial se conoce como *speed-up*, y se suele calcular como la división del tiempo de ejecución secuencial t_s entre el tiempo de ejecución en paralelo t_p (la fórmula sería $speed\text{-}up = t_s / t_p$).

Para llevar a cabo este tipo de ejecuciones paralelas, se requieren sistemas con una características específicas, capaces de ejecutar varios procesos al mismo tiempo. Dependiendo de como estén implementadas estas máquinas, sus arquitecturas se pueden clasificar en distintas modalidades [6], siendo las siguientes las principales:

1. **Computación multinucleo**: esta es una modalidad que hace referencia a aquellas máquinas que contienen varios núcleos en un mismo chip, de forma que pueden ejecutar varias instrucciones al mismo tiempo. Estos procesadores pueden ser idénticos (homogéneos) o tener diferencias entre ellos (heterogéneos).

2. **Multiprocesamiento simétrico**: esta consiste en computadoras que poseen dos o más procesadores conectados por un bus, de forma que comparten la misma unidad de memoria, además de que poseen su propia memoria de caché. Así, cada procesador puede trabajar sin tener que asegurar al ejecutar cada instrucción si posee o no la información que necesita para llevar a cabo sus operaciones.

3. **Computadores de memoria distribuida**: esta es la modalidad más común para computación paralela a gran escala, como en el caso de este trabajo. Este tipo de arquitectura incluye sistemas compuestos por distintos ordenadores que se comunican entre ellos por red, y que cooperan para llevar a cabo las diferentes tareas que pueden recibir. Debido a los altos números de ordenadores que se pueden incluir, y a la independencia de fallo entre ellos, este tipo de sistemas se suelen emplear en aplicaciones que deben procesar una gran demanda de peticiones.

4. **Procesamiento paralelo masivo**: este tipo de arquitectura es similar a la de los sistemas distribuidos, con la diferencia que se componen de procesadores en una misma máquina que se comunican por redes de interconexión, en vez de de consistir en diferentes máquinas.

En este proyecto se han realizado una serie de pruebas que requerían llevar a cabo una serie de ejecuciones paralelas. Para ello, se ha empleado un **clúster**, el cual es un tipo de sistema distribuido. Similar a como se ha mencionado anteriormente, un clúster es un sistema compuesto por varias computadoras que se comunican entre si a través de una red de alta velocidad y que para el interlocutor externo actúan como si fueran una única computadora. Además, todos estos trabajan constantemente en la misma tarea, pero repartiéndose la carga de trabajo. Son las arquitecturas computacionales paralelas más extendidas, debido a su alta tolerancia a fallos y sus bajos precios.

La computación paralela es empleada en muchos ámbitos [7], debido a que es una herramienta con el potencial de mejorar el rendimiento de cualquier tecnología y favorecer a todo tipo de organizaciones que puedan requerir de sus servicios. Pero además, la computación paralela es ampliamente utilizada en las mismas aplicaciones de la computación científica ya que, como se mencionó, requieren del calculo de operaciones complejas con un alto coste computacional, cuyo tiempo se puede reducir empleando paralelismo. Esto mismo es lo que se pretende aplicar a lo largo de este trabajo.

1.1 Motivación

La simulación de problemas científicos reales es una de las aplicaciones más importantes que tiene la informática actualmente. Desde la reproducción de maremotos y terremotos de gran escala, la simulación de dinámicas de fluidos sobre aviones, submarinos o turismos, o la representación de fenómenos astronómicos, la computación científica nos permite estudiar fenómenos poco comunes o difíciles de replicar, y ver que podría suceder en estos casos antes de que sucedan. Esto no solo aumenta nuestro conocimiento del mundo en el que vivimos, sino que en algunos casos, aumenta nuestra seguridad, reflejando la importancia que tiene esta materia.

Otro factor de mucha relevancia no es solo poder representar estos fenómenos, sino también la precisión con la que se representan, y además, la eficiencia. Estas simulaciones pueden consumir una gran cantidad de recursos en tiempo y dinero que no siempre son viables, por lo cual muchas tecnologías como la computación paralela han tratado de

solventar estos problemas. Es por estas razones por las que me he sentido motivado para hacer este trabajo.

1.2 Objetivos

Los objetivos de este trabajo serían, por un lado, elaborar un algoritmo en C que sea capaz de calcular y representar figuras de Chladni en distintos tamaños de malla, tanto de forma secuencial como paralela, empleando librerías especializadas en computación científica escalable.

Por otro lado, se pretende analizar la eficiencia y la escalabilidad del nuevo código desarrollado al ejecutarlo con diferente número de procesos, y comprobar la mejora que se puede llegar a alcanzar al emplear paralelismo en la nueva solución. Así, se busca poder demostrar la utilidad de emplear técnicas de computación paralela en este tipo de problemas.

1.3 Estructura de la memoria

La estructura de esta memoria es la siguiente: en el segundo capítulo, se describen una serie de conceptos teóricos relacionados con los contenidos de este trabajo, así como se exponen algunas de las herramientas empleadas, con el fin de facilitar la comprensión del lector. En el siguiente capítulo, se describe brevemente algunos aspectos de una solución del problema propuesta en un artículo publicado en una revista de la Sociedad de Matemáticas Aplicadas e Industriales o SIAM (*Society for Industrial and Applied Mathematics*) en 2012. En el cuarto capítulo, se revisa el código de Matlab y se analiza su rendimiento según ciertos parámetros del algoritmo empleado, buscando la versión con la máxima eficiencia. En el quinto capítulo, se describe el nuevo código escalable desarrollado en C, la forma en la que está implementado, y los aspectos más destacables en relación a su desarrollo, y en el siguiente capítulo, se comprueba su rendimiento según el número de procesos empleados. Finalmente, en el último capítulo, se concluye sobre el rendimiento de la nueva solución desarrollada, y cuanta mejora se ha obtenido empleando técnicas de programación paralela, además de que se reflexiona sobre algunos temas relacionados con el desarrollo de este trabajo.

Aparte, al final de esta memoria se encuentran la bibliografía y tres apéndices. En el primer apéndice se describen las características de las máquinas empleadas para hacer

las pruebas de los códigos. En el segundo, se encuentran las cuarenta y seis primeras figuras de Chladni que el nuevo código paralelo logró formar. Finalmente, en el último apéndice, se muestra el código paralelo desarrollado, en caso de que se quiera ver de forma más completa como está formado el algoritmo.

CAPÍTULO 2

Fundamentos teóricos

Previamente a la explicación de la investigación, se deben explicar ciertos conceptos teóricos estrechamente relacionados con el contenido del trabajo. Además, conviene describir el software y las diferentes librerías de programación que se utilizaron en la realización de este trabajo.

2.1 Conceptos teóricos

2.1.1. Problema de los autovalores

Este en un problema muy relevante en el mundo matemático [8]. Los autovectores o vectores propios, son vectores no nulos que al multiplicarse por una matriz no cambian su dirección, sino que se convierten en un múltiplo escalar de sí mismos. Así, el problema de los autovalores (o valores propios) se define como que, dada una matriz $A \in \mathbb{R}^{nxn}$, encontrar aquellos valores $\lambda \in \mathbb{R}$ y vectores $x \in \mathbb{R}^n$ que cumplan la ecuación:

$$Ax = \lambda x \tag{2.1}$$

siendo λ y x los autovalores y autovectores de la matriz, respectivamente. Así, cada autovalor tiene un autovector asociado. Por otro lado, tenemos el problema generalizado de los autovalores, idéntico al problema anterior, pero en el que se incluye una nueva matriz $B \in \mathbb{R}^{nxn}$, de forma que los autovalores y autovectores tiene que cumplir la ecuación:

$$Ax = \lambda Bx \tag{2.2}$$

En el caso de las figuras de Chladni, A es la matriz que representa los puntos de la malla y las relaciones de estos con los otros puntos. Así, cada autovector encontrado correspondería a una figura de Chladni distinta.

Una herramienta que se suele usar a la hora de resolver este tipo de problemas es la **transformación espectral**. Esto convierte el problema en uno nuevo, de forma que los autovalores se mueven a una nueva posición, mientras que los autovectores se quedan igual. De esta forma, se acelera la convergencia, y por lo tanto, el tiempo de computación.

Un ejemplo de importancia en este trabajo es la técnica del *shift-and-invert* [9], en la que se acelera la convergencia a un determinado valor σ, convirtiendo las ecuaciones a

$$(A - \sigma I)^{-1}x = \theta x (A - \sigma B)^{-1}Bx = \theta x \tag{2.3}$$

siendo $\theta = 1/(\lambda - \sigma)$.

2.1.2. Método de discretización

En el caso de las figuras de Chladni, el valor de onda en todos los puntos cumple la ecuación biarmónica, la cual es una ecuación diferencial. Sin embargo, calcular este tipo de ecuaciones puede ser muy costoso computacionalmente, por lo que han surgido diversas alternativas para resolver dichas ecuaciones de forma más eficiente.

El método de diferencia finita es una técnica matemática que permite calcular ecuaciones diferenciales, aproximándolas a diferencias finitas más sencillas de resolver. La idea principal en la que se basa es que, teniendo en cuenta que la derivada de una ecuación representa su variación o su "pendiente", el valor en un determinado punto se puede calcular a partir de la diferencia de sus valores cercanos.

Por ejemplo, teniendo en cuenta que N es el tamaño de la malla del problema, y que $i, j \in \{1, 2, ..., N\}$ imaginemos la ecuación:

$$-\frac{d^2u(x_i, y_j)}{dx^2} - \frac{d^2u(x_i, y_j)}{dy^2} = f(x_i, y_j) \tag{2.4}$$

en la que (x_i, y_j) son puntos discretos del dominio donde se quiere resolver la ecuación diferencial, f es una función de la que conocemos su valor en todos los puntos, y u es una función cuyos valores desconocemos y queremos calcular.

Siguiendo el principio de discretización descrito, se cumple la siguiente aproximación:

$$\frac{d^2u(x_i)}{dx^2} \sim \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} \tag{2.5}$$

siendo $h \in \mathbb{R}$ la distancia entre dos puntos. Además, para simplificar la ecuación, se usa la notación $u(x_i) = u_i$.

Así, podemos ver que la ecuación 2.4 se puede convertir a:

$$-\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} - \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} = f(i,j) \tag{2.6}$$

Esta ecuación es equivalente a:

$$4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f_{i,j} \tag{2.7}$$

De esta forma, habrá que calcular esta ecuación para cada punto de la malla de nuestro problema, por lo que tendremos un sistema de ecuaciones. Sin embargo, hemos convertido lo que era una ecuación diferencial a otra que cuenta únicamente con diferencias finitas, con lo cual hemos disminuido considerablemente la potencia computacional requerida para resolver el problema.

2.1.3. Matrices dispersas

Para la resolución de este problema y su posterior representación, es necesario guardar la información de cada punto de la malla, además de la relación de estos con los otros puntos. Para ello, en ambas implementaciones se hace uso de una matriz. Sin embargo, véase lo que conlleva esto: en caso de que tengamos una malla cuadrada relativamente pequeña de por ejemplo, 1.000 puntos, será necesario tener una matriz de 1.000.000 de elementos.

Se puede ver que a medida que se vaya aumentando el número de puntos de la malla que queramos visualizar, el tamaño de la matriz aumentará de forma cuadrática, y muy rápidamente, llegará a un punto en el que la cantidad de recursos de memoria y procesamiento para administrar dicha matriz no sean factibles para la mayoría de usuarios. Por ello, en este trabajo se hacen uso de **matrices dispersas**. Estas son un tipo especial de

matriz en la que solo se almacena la posición y el valor de aquellos elementos no nulos de la matriz, y del resto de celdas se asume que el valor es cero.

En el caso de las figuras de Chladni, por cada punto solo es necesario almacenar información del punto en si y sus cuatro vecinos más cercanos. Por lo tanto, por cada fila, solo habrán cinco celdas con valor no nulo. Al ser la mayoría de elementos de la matriz nulos, utilizar matrices dispersas resulta muy útil, ya que se reduce considerablemente la cantidad de recursos necesarios.

2.2 Software empleado

2.2.1. PETSc

Una de las herramientas fundamentales para la realización de este trabajo fue la librería de código abierto **PETSc** (Portable, Extensible Toolkit for Scientific Computation). Esta consiste en un conjunto de rutinas y estructuras de datos que permiten representar modelos científicos definidos por ecuaciones diferenciales parciales, empleando soluciones escalables y paralelizables [11].

PETSc da acceso a diferentes objetos que administran la información del problema de forma eficiente, como vectores y matrices paralelizables. Además, una de las principales fortalezas de PETSc es que permite el uso de matrices dispersas, descritas anteriormente, y permite almacenarlas con distintos formatos. Uno de los formatos más comunes es el de fila dispersa comprimida o AIJ, en el que se almacenan los valores no nulos en un vector, sus índices de columna se almacenan en otro vector, y en un tercer vector se indica a que filas pertenecen cada columna. Otros formatos ampliamente usados son el formato BAIJ (igual que AIJ, pero las filas se almacenan por bloques, reduciendo el número de índices) o el SBAIJ (versión simétrica de BAIJ, solo se procesa la parte triangular superior de la matriz), entre otros.

Otras funcionalidad importante de PETSc para este trabajo fueron los *solvers* de ecuaciones lineales escalables, o `KSP`, que ofrece. Estos objetos actúan como una API a estructuras y métodos que permiten de forma sencilla resolver sistemas lineales, no lineales, y sistemas de ecuaciones diferenciales, empleando métodos de Krylov [10]. Estos objetos incluyen unos precondicionadores, o `PC`, que permiten aplicar ciertas operaciones sobre las matrices, como factorización de Cholesky o LU, con el fin de acelerar el proceso. Además, al estar usando métodos de Krylov, se permite definir cuando queremos que se

apliquen estos precondicionadores. En caso de que solo queramos que se aplique una vez al principio del procesamiento, el `KSP` resolverá el sistema con el método directo.

Finalmente, una clase de PETSc de gran utilidad en la solución del problema fue la clase de objetos `IS`, o sets de índices. Su función es almacenar conjuntos de índices de una matriz o un vector, siguiendo una implementación que los hace fácilmente escalables, a la hora de repartir los índices entre varios procesos.

2.2.2. SLEPc

Otra librería de gran importancia en la investigación fue **SLEPc** (Scalable Library for Eigenvalue Problem Computations), una extensión de PETSc desarrollada por investigadores de la Universidad Politécnica de Valencia [13]. Esta librería esta centrada en la solución de problemas de autovalores lineales y no lineales, ya sean estándar, generalizados o cuadráticos, así como en el calculo de la descomposición de valores singulares de una matriz. Al igual que PETSc, SLEPc permite paralelizar todas sus rutinas y métodos de forma sencilla, y también toma ventaja de utilizar matrices dispersas.

SLEPc incluye unos objetos denominados EPS (*eigenvalue problem solver*), que son los principales objetos que proporciona la librería para definir y resolver los problemas de autovalores, empleando métodos como el de Krylov-Schur [14]. Estos incluyen también un objeto ST (*spectral transformation*), que permite aplicar transformaciones espectrales sobre el problema, como *shift-and-invert*, y definir a partir de que valores se quiere que el algoritmo comience a buscar autovalores (se puede indicar que desde el valor más pequeño, o proporcionar un valor real) . Además, este `ST` incluye un objeto `KSP` (que a su vez dispone de un precondicionador `PC`), permitiendo definir más profundamente el procesamiento de la solución.

2.2.3. MPI

Con el fin de calcular la solución de forma paralela, ambas librerías requieren de un sistema para coordinar los distintos procesos. Para ello, tanto PETSc como SLEPc hacen uso de **MPI** (Message Passing Interface). MPI es un estándar de paso de mensajes entre procesos de una misma ejecución, y define la sintaxis y la semántica de las rutinas con la finalidad de que sean portables y puedan usar en diversos lenguajes. Mediante esta interfaz, los distintos procesos de una ejecución pueden coordinarse entre ellos para llevar a cabo las operaciones de forma eficiente.

Para organizar los procesos, MPI hace uso del concepto de comunicadores. Estos agrupan los procesos en grupos, y dentro de cada grupo asignan a cada proceso un identificador o rango único, permitiendo al programador identificar a los procesos durante la ejecución y especificar que procesos ejecutaran determinadas operaciones en el código. Por ejemplo, el comunicador `MPI_COMM_WORLD`, (o `PETSC_COMM_WORLD`, en el caso de utilizar PETSc) hace referencia a todos los procesos de la ejecución, mientras que el comunicador `MPI_COMM_SELF`, (o `PETSC_COMM_SELF`) hace referencia únicamente al proceso que lo invoca.

A la hora de enviar mensajes entre distintos procesos, MPI cuenta con dos tipos de funciones: funciones de punto a punto, y funciones colectivas. Las funciones de punto a punto, como `MPI_Send` o `MPI_Receive` permiten a dos procesos enviar y recibir información que necesitan pero que otro proceso tiene. Por otro lado, las funciones colectivas implican a todos los procesos, y son útiles en situaciones en las que el resultado dependa de una estructura o conjunto de datos repartido entre todos los procesos. Algunos ejemplos son `MPI_Barrier` o `MPI_Reduce` [15].

CAPÍTULO 3

El problema de las figuras de Chladni

Como se mencionó anteriormente en la introducción, para resolver el problema de Chladni y obtener las soluciones que nos permitirán representar las figuras, se debe resolver el problema de los autovalores de la ecuación biharmónica. Esto correspondería a resolver la ecuación

$$\nabla^4 v = \lambda v$$

siendo *nabla*, representada por el símbolo ∇, la derivada (a la cuarta) de la autofunción v, que en este caso devolvería los autovectores de la solución. De igual forma que en el apartado 2.1, λ correspondería a los autovalores. Así, cada solución del problema correspondería a una figura distinta.

Esto requeriría el cálculo de derivadas, lo cual consumiría muchos recursos computacionales si se calcularan directamente. Por ello, una solución propuesta en un artículo publicado por la Sociedad de Matemáticas Aplicadas e Industriales *SIAM* [2], hace uso del método de discretización para simplificar el problema.

El primer paso que se hace en esta solución es simplificar la ecuación. Esta cuenta con una cuarta derivada, la cual puede ser muy complicada de discretizar. Por ello, la solución reduce esta derivada a una segunda derivada, ya que al ser $\nabla^4 v = \nabla(\nabla^2 v)$ y creando una función w equivalente a $-\nabla^2 v$, de forma que la ecuación original se convierte a

$$-\nabla^2 w = \lambda v$$

Así, siguiendo el método de disctretización, podemos convertir la función w a

$$w = \frac{4v_{i,j} - v_{i-1,j} - v_{i+1,j} - v_{i,j-1} - v_{i,j+1}}{h^2}$$

De esta forma, se convierte una ecuación diferencial que contaba con una cuarta derivada, a una que cuenta únicamente con diferencias finitas. Así, la complejidad del problema de autovalores se reduce considerablemente, y los recursos necesarios se hacen más tolerables.

3.1 Puntos físicos, puntos límite y puntos fantasma

Como se puede ver en la discretización de la función w, el valor de la solución de cada punto depende de los valores de sus vecinos más cercanos. Sin embargo, no todos los puntos tienen el mismo número de vecinos, ya que depende de en que posición se encuentran. Es por eso por lo que en este problema, se han de tener en cuenta tres tipos de puntos distintos en la malla.

Por un lado, están los **puntos físicos** (physical points), de los cuales se intenta calcular el valor de onda y es donde se representarán las figuras. Así, independientemente de que algoritmo se use, estos puntos siempre se tendrán en cuenta a la hora de obtener la solución del problema. Estos puntos son los más comunes, y son los que tienen al menos cuatro vecinos cercanos.

Por otro lado están los **puntos límite** (boundary points). Estos son puntos físicos al mismo tiempo, pero tienen la condición especial de que, al no tener vecinos en al menos uno de sus lados (por ejemplo, los puntos límite del borde superior no tiene ningún vecino superior) sus coeficientes serán distintos en la matriz del problema. Cabe destacar que los puntos límite de las esquinas de la malla también podrían considerarse un tipo distinto de puntos, ya que no tiene vecinos en dos de sus lados, y por lo tanto sus coeficientes también serían distintos del resto.

Finalmente, están los **puntos fantasma** (ghost points), los cuales son necesarios para la definición del problema, aunque más adelante son descartados, y no se representan en la solución. Sin embargo, en algunas versiones del algoritmo, estos si se tienen en cuenta, ya que el no eliminarlos puede reducir tiempo de ejecución, aunque los resultados no sean tan precisos. Para descartar estos puntos, se hace uso de las condiciones de frontera, las cuales especifican el valor de la solución en los bordes del domino. En el caso de

las figuras de Chladni, estas son condiciones de frontera libre: los puntos del límite del dominio tienen asociadas incógnitas que se podrán calcular al resolver el problema y no se ven impuestas por restricciones principales a la ecuación principal en cuestión.

En la imagen inferior (Figura 3.1), se puede ver un ejemplo de lo explicado. Las celdas *g* representan los puntos fantasmas, las celdas *b* representan los puntos límite, y las celdas *p* representan los puntos físicos.

g	g	g	g	g	g	g
g	b	b	b	b	b	g
g	b	p	p	p	b	g
g	b	p	p	p	b	g
g	b	p	p	p	b	g
g	b	b	b	b	b	g
g	g	g	g	g	g	g

Figura 3.1: Ejemplo de puntos en una malla de 7x7

CAPÍTULO 4

Análisis de la solución en Matlab

Como se ha comentado anteriormente, ya se ha logrado llevar a cabo el desarrollo de algoritmos que calculen las soluciones del problema de Chladni, siendo uno de los más destacables el algoritmo propuesto en el artículo de la SIAM explicado en el anterior capítulo. Este está implementado en código Matlab [16], y solo se puede utilizar de forma secuencial. Sin embargo, hay diversos parámetros del algoritmo que se pueden modificar, y que alteran el rendimiento y el resultado del algoritmo considerablemente. En este capítulo discutiremos las distintas posibilidades.

4.1 Parámetros analizados

Los parámetros principales, cuyas propiedades sobre el rendimiento del código se han observado, son el **tamaño de malla** y el **número de autovalores**. Cabe destacar que cuando hablamos del tamaño de la malla, esto se refiere al número de puntos en un lado de la malla (por ejemplo, si el tamaño es 100, quiere decir que en total habrán 10.000 puntos en la malla), por lo que se esperaría que el rendimiento aumentara de forma cuadrática, a medida que se va aumentando el tamaño de la malla. Por otro lado, el número de autovalores haría referencia al número de figuras que queremos que se calculen. Esto puede afectar de forma menos predecible al rendimiento, ya que si todos los autovalores tienen el mismo signo, el aumento del tiempo podría ser lineal, pero si no se da el caso, es posible que el aumento sea mayor.

Otro factor que se ha tenido en cuenta es la versión del problema de autovalores a utilizar en el algoritmo. Inicialmente, en la solución se tiene en cuenta una version generalizada del problema de Chladni, en el que se utiliza una matriz A_0 que solo incluye los puntos físicos, y otra matriz diagonal B_0 que incluye valores dependiendo de que la

celda a la que corresponda cada fila sea una celda completa, media celda o un cuarto de celda. Así, el problema a resolver por el algoritmo sería

$$A_0 x = \lambda B_0 x$$

Sin embargo, para este trabajo también se ha analizado el rendimiento del problema estándar, en el que también se calcula el valor de onda en los puntos fantasma. Esta versión, aunque debería ser menos precisa a la hora de calcular las figuras ya que se tienen en cuenta unos puntos que no pertenecen realmente al problema, debería ser más rápida computacionalmente, debido a que no es necesario calcular la matriz B_0, ni realizar todo el preprocesamiento de la matriz A_0 para descartar los puntos fantasmas. Así, se tendría que utilizar la matriz A presente en el código, que incluye todos los puntos, y la ecuación a resolver por el algoritmo sería

$$Ax = \lambda x$$

Debido a esto, hay que tener en cuenta que cuando se resuelve el problema estándar en una malla de tamaño de 100, estamos calculando la solución en 100x100 puntos, mientras que si resolvemos el problema generalizado, estamos descartando los puntos fantasma, por lo que estamos calculando la solución en una malla de 98x98 puntos (los puntos fantasma son los bordes de la malla), por lo que, si el tamaño de malla de nuestro problema es n, el tamaño de las figuras es nxn con el problema estándar, y n-2xn-2 con el problema generalizado, lo que hace que esta última variante tenga mayor coste en este aspecto.

Finalmente, otro elemento del algoritmo que se estudió es el tipo de **transformación** espectral a utilizar. En la solución original, se empleaba la técnica de *shift-and-invert*, con el valor **sigma** igual a 0 como predicción inicial para los autovalores. Esto es equivalente a empezar la convergencia, encontrando el autovalor más pequeño. Sin embargo, para este trabajo también se ha probado a especificar un valor para la transformación, con el fin de ver si este se acerca más al primer autovalor y acelera la convergencia como consecuencia.

4.2 Metodología

Las pruebas se llevaron a cabo en la máquina *GPU* proporcionada por la Universitat Politècnica de València, y cuyas especificaciones se pueden encontrar en el apéndice A. Para llevar a cabo dichas pruebas, se empleo el software de Matlab instalado en la máquina, pero sin llegar a abrir la interfaz gráfica de la aplicación, y todas las ejecuciones se hicieron por línea de comandos. Además, al hacer las ejecuciones de prueba, no se procesaron los dibujos de las figuras, y solo se tuvo en cuenta el calculo de los autovalores y autovectores del problema.

Se han hecho un total de 120 pruebas, a partir de las combinaciones de los parámetros. Para el tamaño de malla, se probó con diez tamaños de malla: 100, 200, 300... y 1.000 puntos por lateral. En cuanto al número de autovalores, se realizaron ejecuciones que calcularon 50, 100, y 150 autovalores distintos. Por otro lado, como se ha mencionado en el anterior apartado, todas estas pruebas se realizaron tanto con la versión generalizada como estándar del problema. Finalmente, se observó el rendimiento del algoritmo empleando transformación espectral con sigma 0, y también con sigma -0.01.

Aparte, también se realizaron otras ejecuciones sin medir los tiempos, en las que se recopilaron las figuras del algoritmo con cada combinación de parámetros. Cabe destacar que de estas figuras se descartaron las tres primeras, ya que eran correspondientes a tres autovalores cero, debido a que el algoritmo al principio no tiene suficiente información para converger.

4.3 Resultados

4.3.1. Problema general con sigma 0

Esta versión del problema corresponde a la solución original. A continuación se muestran los tiempos de ejecución observados, en la Tabla 4.1. Las columnas diferencian los tamaños de malla (n), y las filas diferencian el número de autovalores (k). También se muestran los datos en la Figura 4.1, en formato de gráficas, con el fin de discernir más fácilmente el cambio de tiempo de procesamiento con respecto a los parámetros.

Tabla 4.1: Medidas de tiempo del problema generalizado con sigma 0

n	100	200	300	400	500	600	700	800	900	1000
k=50	0.753	3.324	8.925	18.738	34.461	55.197	81.246	123.108	165.958	249.886
k=100	1.038	5.661	15.274	25.649	46.201	72.499	121.726	174.197	242.355	295.771
k=150	1.300	6.554	17.848	34.513	59.611	93.740	135.005	194.714	269.097	377.558

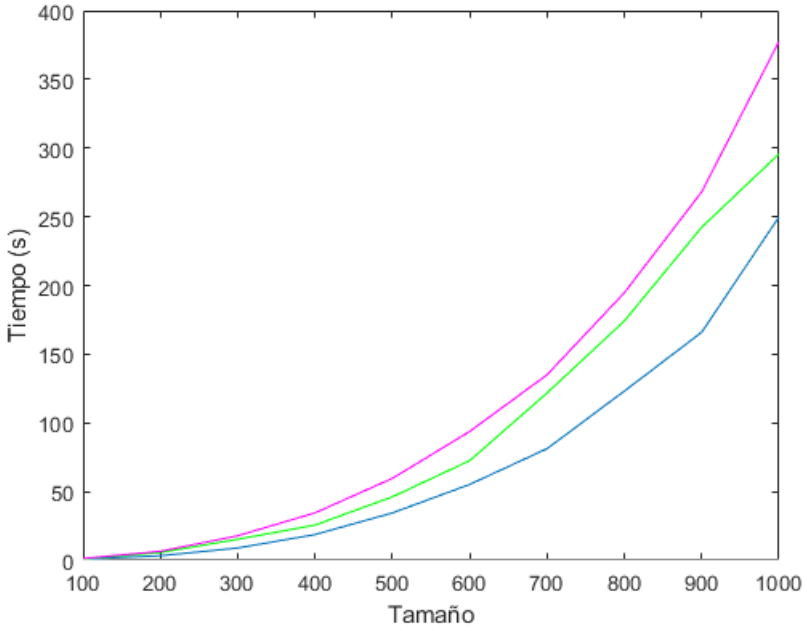


Figura 4.1: Resultados del algoritmo original. En la línea azul se encuentran los tiempos para calcular 50 autovalores, en la línea verde para calcular 100 autovalores, y en la línea magenta para 150 autovalores

Como se puede ver, el tiempo de procesamiento es casi cuadrático con respecto al tamaño de malla en el caso de 50 autovalores, lo cual es normal teniendo en cuenta que ha medida que se va aumentando este parámetro, el número de puntos a calcular aumenta de forma cuadrática. Sin embargo, cuando se calculan 100 o 150 autovalores, el crecimiento es mayor, ya que se está calculando un número mayor de figuras. Por otro lado, se puede vislumbrar que la diferencia de aumento entre 50 a 100 autovalores es mayor que la de 100 a 150 autovalores, con lo que se podría deducir que a medida que cuanto más aumenta el número de autovalores a calcular, menor será el aumento de tiempo.Esto se puede deber a que, cuantas más figuras son calculadas, el algoritmo obtiene más información, y por lo tanto cada vez converge más rápido.

Algunas figuras creadas por el algoritmo se muestran a continuación. Como se puede observar en la Figura 4.2, las figuras resultantes son muy similares a las figuras originales descubiertas por Chladni en 1787. Así, se espera que las otras versiones del algoritmo tengan resultados similares.

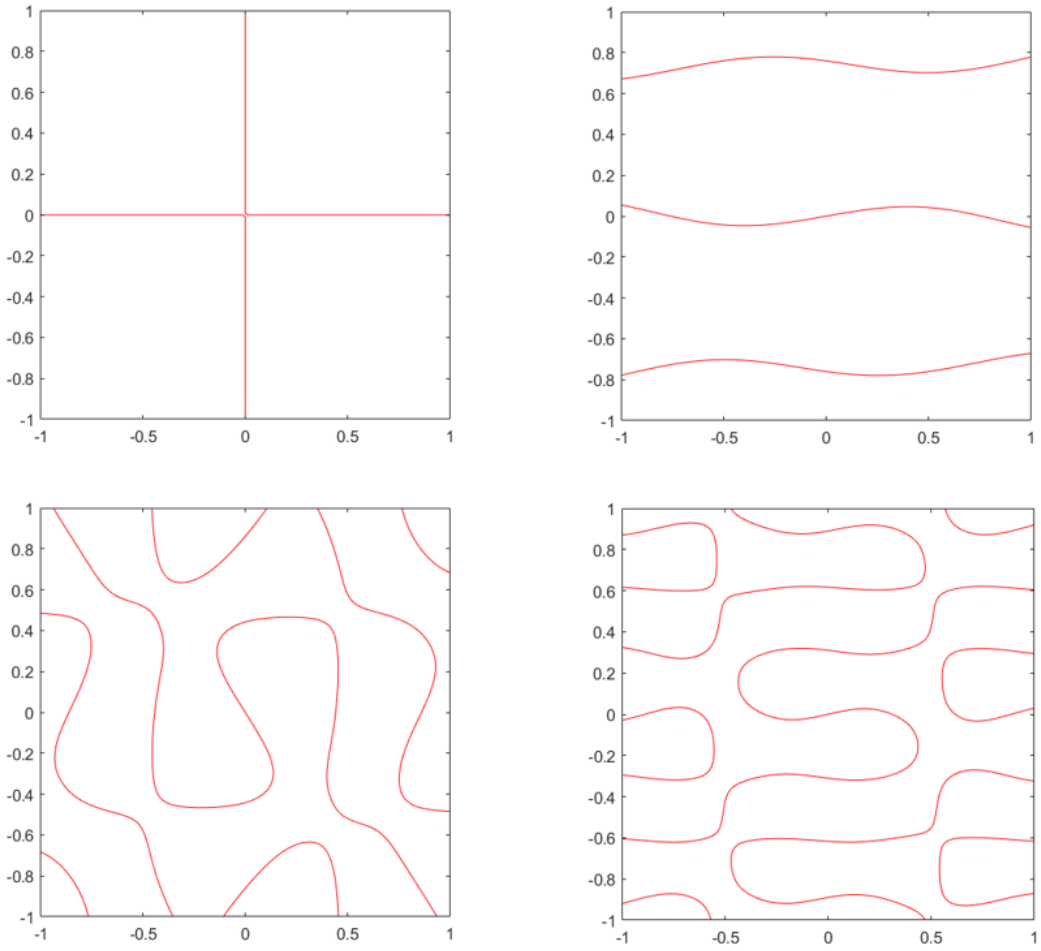


Figura 4.2: Figuras correspondientes a los autovalores 4 (izquierda superior), 10 (derecha superior, 30 (izquierda inferior) y 50 (derecha inferior)

4.3.2. Problema estándar con sigma 0

En este caso, el algoritmo es igual al original, pero en vez de intentar resolver el problema generalizado, se intenta resolver el problema estándar. Para esto solo hay que hacer una pequeña modificación al código, cambiando los parámetros de la llamada

```
eigs( A0/h^4, B0, M, 'SM');
```

por la siguiente instrucción

```
eigs( A/h^4, M, 'SM');
```

En la instrucción original, se llama a una función estándar de Matlab que calcula autovalores y autovectores. En la versión inicial, se le pasa a matriz `A_0`, que corresponde a la parte izquierda de la ecuación, y la matriz `B_0`, que corresponde a la parte derecha. La variable `M` indica el número total de autovalores a calcular, y `SM` es la predicción inicial al aplicar la transformación inicial (`SM` se traduce a "smallest magnitude", que es equivalente

a utilizar una valor de sigma igual a cero). En la nueva instrucción, solo le pasamos la matriz `A`, con lo cual pasamos a utilizar el problema estándar.

Los medidas de tiempo de todo el algoritmo con el cambio mostrado en la instrucción anterior se muestran en la Tabla 4.2, con las mismas propiedades que en el anterior caso. Igualmente, en la Figura 4.3 se muestran las gráficas con el fin de poder observar la evolución con respecto a los parámetros.

Tabla 4.2: Medidas de tiempo del problema estándar con sigma 0

n	100	200	300	400	500	600	700	800	900	1000
k=50	0.938	3.194	8.565	18.990	33.579	52.296	83.145	121.053	160.795	245.877
k=100	0.920	4.609	12.464	25.134	45.649	70.140	105.660	145.671	195.813	278.188
k=150	1.415	6.982	17.998	35.515	63.043	94.863	141.898	193.487	257.346	359.137

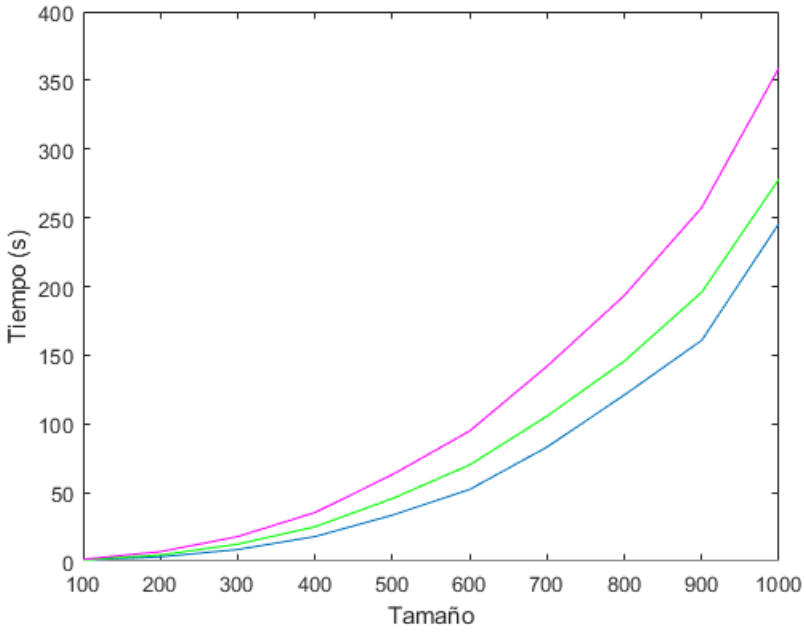


Figura 4.3: Resultados del algoritmo con problema estándar y sigma 0. En la línea azul se encuentran los tiempos para calcular 50 autovalores, en la línea verde para calcular 100 autovalores, y en la línea magenta para 150 autovalores

Como se puede observar, el crecimiento sigue siendo casi cuadrático con 50 autovalores, y mayor que cuadrático con 100 y 150 autovalores. Y como en el anterior caso, el aumento de tiempo sigue siendo mayor al calcular entre 50 y 100 autovalors, que entre 100 y 150. Sin embargo, se esperaba que al evadir el procesamiento de las matrices B, $B0$ y $A0$, el tiempo se viera reducido. Sin embargo, en comparación con respecto a la versión generalizada del problema, los tiempos siguen siendo muy similares, con lo que no se consigue la ventaja esperada.

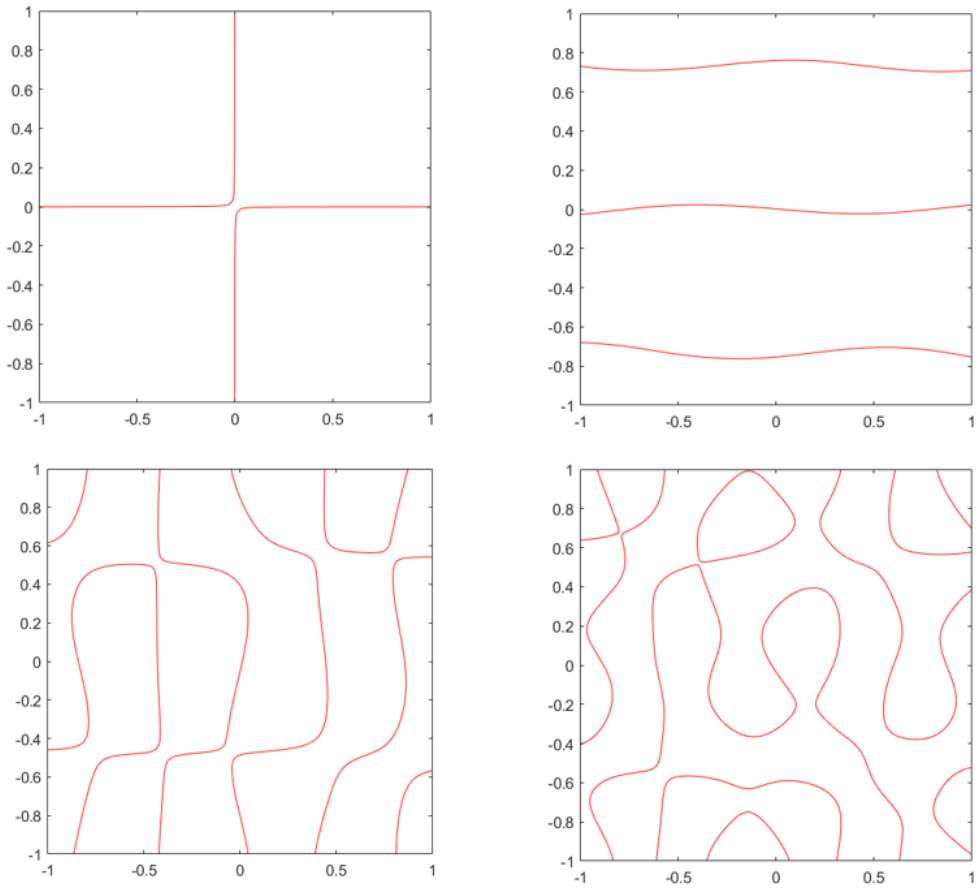


Figura 4.4: Figuras correspondientes a los autovalores 4 (izquierda superior), 10 (derecha superior, 30 (izquierda inferior) y 50 (derecha inferior). Aunque las figuras de los autovalores pequeños son similares a las originales, las de los mayores valores se diferencia de mayor forma.

Por otro lado, observando la Figura 4.4, podemos ver que las figuras resultantes tienen ciertas diferencias con las de la versión original, lo cual era de esperar teniendo en cuenta que esta vez no hemos descartado los puntos fantasma. Con esto, se podría llegar a la conclusión de que utilizar esta versión no es eficiente.

4.3.3. Problema general con sigma -0.01

En el algoritmo original, se aplica transformación espectral con sigma cero, con el fin de acelerar la convergencia a los autovalores más pequeños. En esta nueva versión, se cambia el valor de sigma, con el fin de comprobar si esto acelera la convergencia, y que los resultados o figuras resultantes sean similares a los de la versión original.

Para llevar a cabo esta modificación, solo hubo que cambiar el parámetro correspondiente en la llamada a la función `eigs` mostrada anteriormente, por el valor *-1e-2*, como se muestra a continuación

```
eigs(A0/h^4, B0, M, -1e-2);
```

De esta forma, se aplica *shift-and-invert* con un valor de sigma de -0.01. En la Tabla 4.3 y en las gráficas de la Figura 4.5, se muestran los resultados y la variación del tiempo de procesamiento, respectivamente.

Tabla 4.3: Medidas de tiempo del problema generalizado con sigma -0.01

n	100	200	300	400	500	600	700	800	900	1000
k=50	0.757	2.503	5.888	12.736	26.208	37.434	56.932	90.047	120.865	149.292
k=100	0.822	4.568	10.599	21.922	41.968	49.700	85.424	137.042	166.089	189.588
k=150	0.992	5.609	12.853	26.007	50.412	66.561	98.558	155.261	178.627	239.285

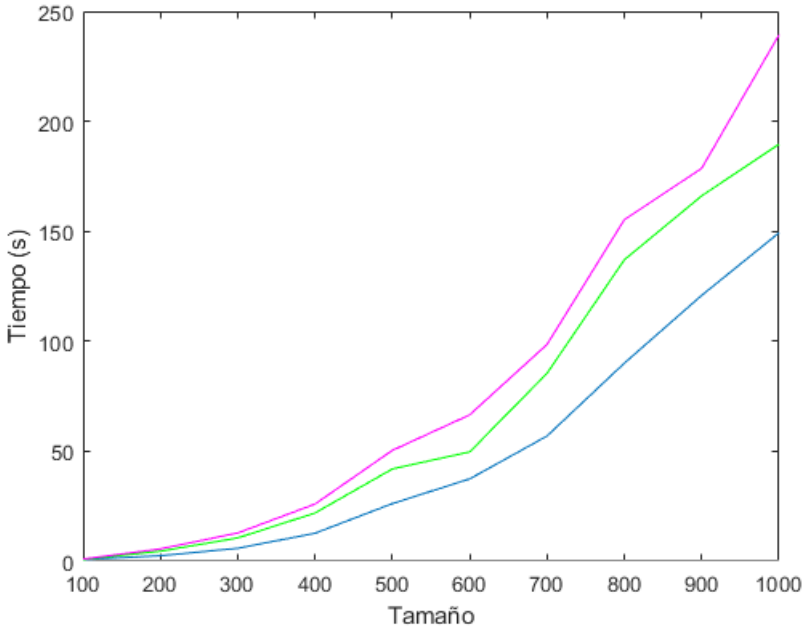


Figura 4.5: Resultados del algoritmo con problema general y sigma -0.01. En la línea azul se encuentran los tiempos para calcular 50 autovalores, en la línea verde para calcular 100 autovalores, y en la línea magenta para 150 autovalores

Como se puede observar, la variación del tiempo con respecto al tamaño de la malla no es tan uniforme que en los otros casos, cuando se empleaba un valor de sigma cero, lo cual podría dificultar más predecir cuanto durará la ejecución del algoritmo. Sin embargo, a diferencia de cuando se cambió al problema estándar, cambiar el valor de sigma reduce considerablemente el tiempo de ejecución, debido a la aceleración en la convergencia de los autovalores. Por ejemplo, en el caso de calcular 150 autovalores en una malla de 1000x1000, el tiempo en el algoritmo original era de 337.556 segundos, mientras que en esta nueva versión son 239.285 segundos.

Por otro lado, la Figura 4.6 nos muestra que los resultados son muy similares a las originales. Viendo esto, podemos concluir que esta versión es muy eficiente y precisa, convirtiéndola en una posible alternativa al algoritmo original.

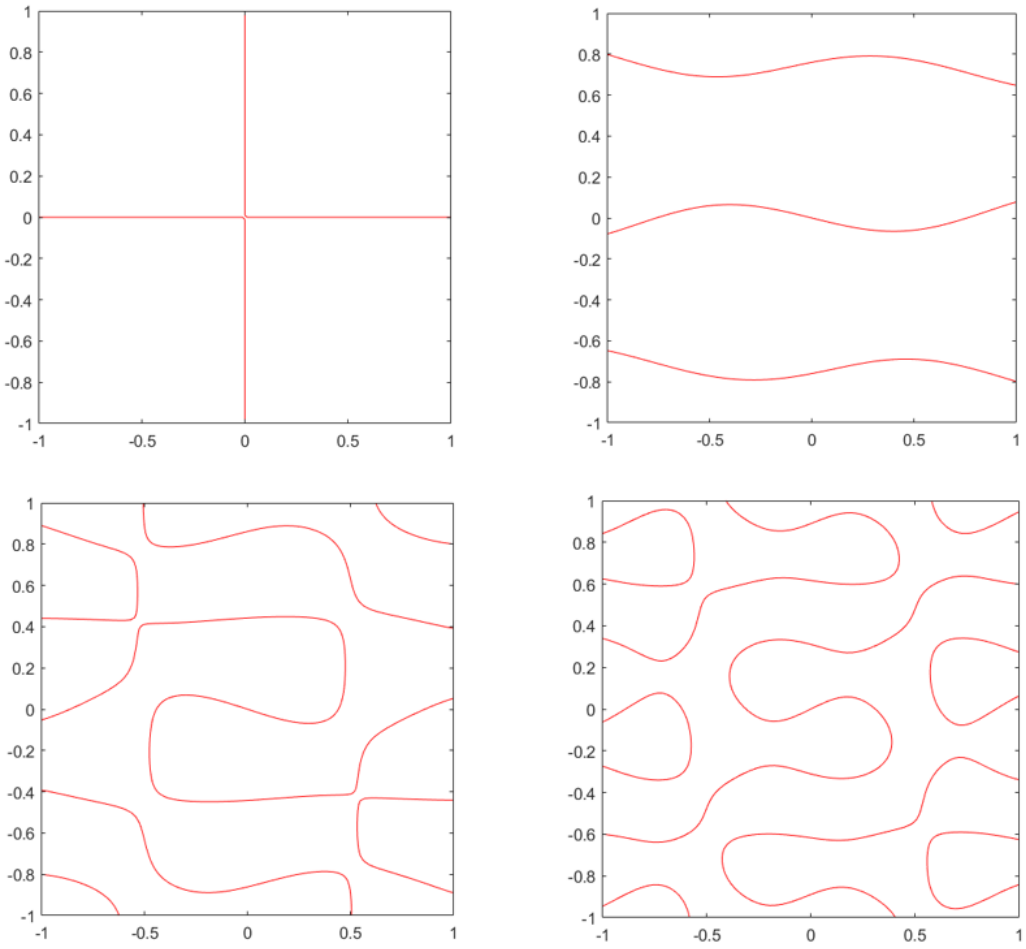


Figura 4.6: Figuras correspondientes a los autovalores 4 (izquierda superior), 10 (derecha superior, 30 (izquierda inferior) y 50 (derecha inferior). Salvo algunas diferencias, los resultados son muy parecidos a los originales

4.3.4. Problema estándar con sigma -0.01

En este último caso, combinamos todas las modificaciones que hemos hecho hasta ahora. Por un lado, intentamos resolver el problema estándar en vez del problema general, y en vez de utilizar un valor de sigma cero para la transformación espectral, utilizamos un valor de -0.01. Para ello, como se ha visto en los anteriores casos, podemos eliminar todo el proceso de creación de las matrices `A_0`, `B` y `B_0`, y cambiar la llamada de la función `eigs` a

```
eigs(A/h^4, M, -1e-2);
```

A continuación, en la Tabla 4.4 y en la Figura 4.7, se muestran los resultados de esta versión. Viendo como anteriormente los tiempo de ejecución no mejoraron al cambiar al problema estándar, es posible que en este caso se de lo mismo. Sin embargo, ahora que utilizamos un valor distinto de sigma, cabe la posibilidad de que la velocidad de la convergencia aumente.

Tabla 4.4: Medidas de tiempo del problema estándar con sigma -0.01

n	100	200	300	400	500	600	700	800	900	1000
k=50	0.375	2.421	5.760	11.752	24.291	35.696	53.820	82.915	118.178	181.526
k=100	0.804	3.752	10.262	17.053	34.475	48.979	72.476	111.791	134.164	193.997
k=150	0.995	5.812	12.925	25.173	48.321	65.325	95.473	148.467	175.962	229.566

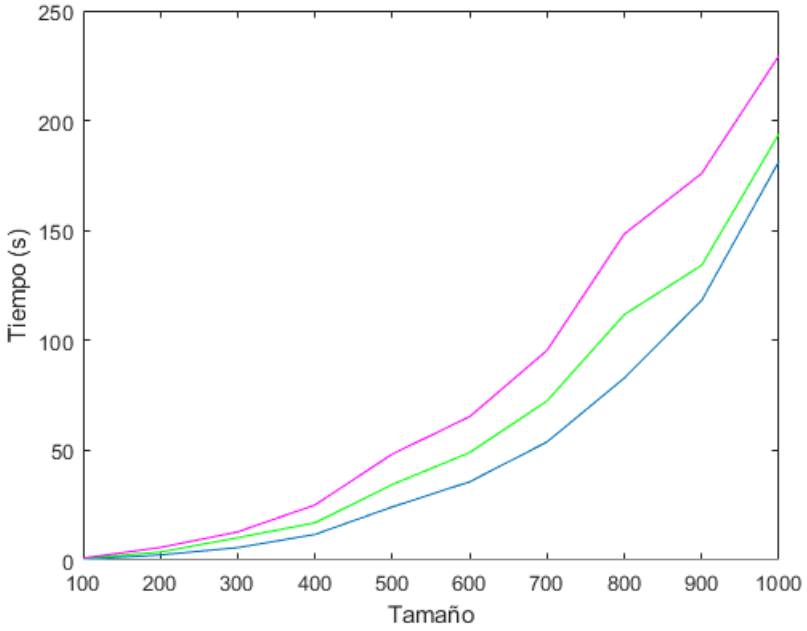


Figura 4.7: Resultados del algoritmo con problema estándar y sigma -0.01l. En la línea azul se encuentran los tiempos para calcular 50 autovalores, en la línea verde para calcular 100 autovalores, y en la línea magenta para 150 autovalores

Como se puede observar, se han reducido considerablemente los tiempos con respecto a la versión original. Sin embargo, en comparación a la versión general con sigma -0.01, los tiempos son muy similares. Por otro lado, las forma resultantes que se muestran en la Figura 4.8 son muy precisas, y no difieren en gran medida de las figuras originales.

Con esto, podemos concluir que emplear esta versión del algoritmo también es considerable, debido a la precisión de los resultados, y la reducción de los tiempos de procesamiento. A pesar de esto, la versión del problema generalizado con sigma -0.01 también ofrecía resultados muy precisos con tiempo similares, por lo que ambas versiones se podrían considerar como alternativas a la versión original.

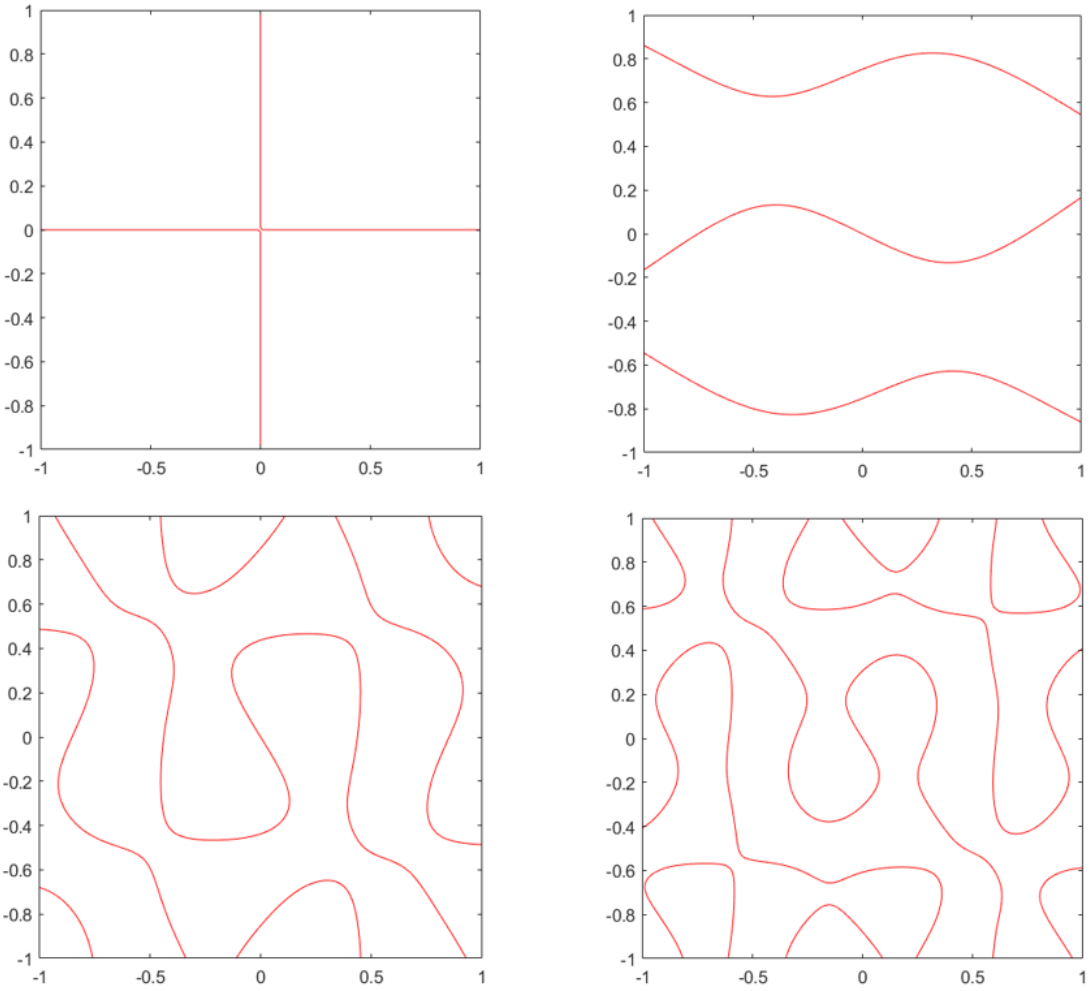


Figura 4.8: Figuras correspondientes a los autovalores 4 (izquierda superior), 10 (derecha superior, 30 (izquierda inferior) y 50 (derecha inferior). Salvo algunas diferencias, los resultados son muy parecidos a los originales

4.4 Conclusiones del análisis

Como se ha podido observar, las versiones que emplean la versión del problema generalizado son muchos más rápidas que las versiones que emplean las versión estándar. Además, emplear un valor de sigma -0.01 acelera más la velocidad, y proporciona resultados muy similares a los originales. En conclusión, la mejor versión a utilizar es la **generalizada con sigma -0.01**, la cual es la versión que se ha empleado como punto de partida a la hora de desarrollar nuestra propia versión de la solución.

CAPÍTULO 5

Desarrollo de la solución en PETSc

En este capítulo, se describe el diseño e implementación de la solución para el problema de las figuras de Chladni, empleando las librerías de programación PETSc y SLEPc, y aplicando técnicas de computación paralela y científica. Este código fue escrito en el lenguaje de programación C, y sigue una estructura muy similar a la del código original en Matlab. Para ver el código con más detalle, se puede consultar en el apéndice C.

5.1 Clases empleadas

En la implementación de este código, todos los objetos empleados pertenecen exclusivamente a las librerías PETSc y SLEPc. Por ejemplo, para almacenar las variables numéricas del problema, como el tamaño de malla (`n`), la distancia entre dos puntos (`h`) o el número de puntos físicos y puntos fantasma (`sp` y (`sr`)), se utilizaron los tipos de datos básicos **PetscInt** y **PetscScalar**. Estos permiten almacenar valores enteros y reales, al igual que otros tradicionales tipos de datos como `int` o `double`, con la diferencia de que están adaptadas a PETSc y permiten modificar el número de bits con los que se representaran.

Para almacenar las matrices del algoritmo, como la matriz `A` que guarda los coeficientes de la discretización, se hace uso de la clase `Mat`. Al ser este tipo de matriz parte de la librería PETSc, hacer uso de las funciones de esta es mucho más sencillo. En la mayoría de casos estas matrices son de tipo dispersa, a excepción de las matrices que se pasan a funciones que requieren de una matriz densa.

En este código también se emplean objetos de la clase `IS`, perteneciente a PETSc. La función principal que tienen es indexar determinados puntos de las matrices dependien-

do del tipo. Más específicamente, se tienen dos objetos IS denominados `phys`, con los que se seleccionan a los puntos físicos de la matriz, y `ghost`, que permite seleccionar los puntos fantasma.

Por otro lado, en un determinado punto del algoritmo, es necesario calcular el complemento de Schur de la matriz `A` [18], para poder determinar la solución en los puntos físicos, sin tener que calcular también la solución en los puntos fantasma. La fórmula de este complemento sería

$$A/A_0 := A_0 - A_1 A_3^{-1} A_2 \tag{5.1}$$

donde A_0 sería la submatriz de A en los puntos de los índices `phys` x `phys`, A_1 sería la submatriz en los índices `phys` x `ghost`, A_2 sería la submatriz en los índices `ghost` x `phys` y A_3 sería la submatriz de los índices `ghost` x `ghost`. De esta forma, estamos calculando el complemento en los puntos físicos.

Como se puede ver, para llevar a cabo esta operación, es necesario calcular la inversa de la matriz A_3, que representa uno de los cuadrantes de la matriz A. Calcular una inversa es equivalente a resolver el sistema $A_3X = I$), siendo I la matriz identidad. Para llevar a cabo esta operación, se emplea un objeto de la clase `KSP`, que como se explico previamente, ofrece una interfaz directa para resolver sistemas de ecuaciones lineales. A parte, para resolver este problema eficientemente, es necesario factorizar la matriz con el método LU, por lo que un precondicionador `PC` también fue necesario para definir estas condiciones.

Finalmente, uno de los componentes más importantes en el funcionamiento del algoritmo fue un objeto de la clase `EPS`, de la librería de SLEPc. Es la parte del código que recibe las matrices de la ecuación del problema de los autovalores, y calcula la solución. Es en este objeto en el que se define el tipo de problema, el número de autovalores que se quieren calcular y que clase de transformación espectral se quiere utilizar, mediante un objeto `ST` conectado al objeto `eps`. También se emplea otro precondicionador `pc` para aplicar factorización LU sobre el problema.

A parte, con el fin de representar las figuras una vez se han calculado los autovalores, se utilizan cuatro objetos PETSc de clases distintas para llevarlo a cabo. El primero es un vector `x` de la clase `Vec`, que se emplea para almacenar el autovector. El segundo es un objeto `DMDA` (*Distributed array*), que administra los valores del autovector para que se pueda representar en una malla 2D. Los otros dos objetos son de las clases `PetscViewer`

y `PetscDraw`, y son los que permiten visualizar gráficamente y almacenar las figuras resultantes.

5.2 Métodos empleados

El algoritmo principal se denomina `chladni`, y recibe dos parámetros. El primero es el tamaño de malla `n`, y el segundo es el número de autovalores `M` que se quieren calcular. Así, poder realizar las operaciones necesarias en esta función y administrar todos los objetos mencionados anteriormente, se utilizan funciones PETSc [12] diseñadas para llevar a cabo todo tipo de operaciones, haciendo uso de clases PETSc.

Por un lado, para crear las matrices, se sigue la rutina estándar recomendada por los desarrolladores de PETSc que consiste en, primero, crear la matriz con la función `MatCreate`, que la coloca en memoria y define su comunicador (que procesos van a tener acceso a ella, como `PETSC_COMM_WORLD`). Después, se establecen sus medidas, con `MatSetSizes`, con la que definimos el número de filas y de columnas tanto locales como globales. Finalmente, se establece la matriz en la máquina, con la función `MatSetUp`.

A lo largo del código, es necesario introducir o añadir valores en las matrices, para lo cual se utiliza las funciones `MatSetValue` y `MatSetValues`. A esta se le pasa la matriz, las filas y las columnas que queremos modificar de dicha matriz, el valor o el array con valores y la operación que queremos realizar (`INSERT_VALUES` para insertar los valores o `ADD_VALUES` para añadir los valores). Estas últimas funciones requerían que los índices de las filas y las columnas estuvieran almacenadas en arrays, por lo que no fue posible aplicar el uso de objetos `IS`. Sin embargo, la función `MatZeroRowsIS` si que lo permitía. Esta función llena de ceros las filas de una matriz, y requería de la matriz original, los indices de las filas almacenados en `IS` y un valor para asignarlo a las celdas de la diagonal, en caso de que se requiriera que no fuera cero (en nuestro caso, no era necesario, por lo que este valor era cero).

Para crear los index sets, se utiliza la función `ISCreateGeneral`, que recibe el comunicador de los procesos que lo tendrán, el número de elementos locales, el array de valores y la operación que se quiere realizar (en nuestro caso es `PETSC_COPY_VALUES`, que copia los valores al `IS`). Por otro lado, para construir los array con índices, se creó la función `createVec`, que recibe el puntero al array, el tamaño y el primer valor de cada cuarto, y a partir de esto introduce todos los valores.

Un punto importante en cuanto a las matrices, es que cada vez que se cambian algunos de sus valores, antes de utilizarlas en otras operaciones de cálculo, es necesario ensamblarlas. Para ello, se emplean las funciones `MatAssemblyBegin` y `MatAssemblyEnd`. Por otro lado, al realizar algunas operaciones, es posible en las matrices dispersas se queden restos de celdas con valores nulos. Para eliminarlas, después de cada operación se emplea la función `MatSetOption`, con el parámetro `MAT_IGNORE_ZERO_ENTRIES` y el valor booleano `PETSC_TRUE`, con lo que se indica que se quieren borrar las celdas nulas. En otra parte del código, justo antes de introducir filas de ceros con `MatZeroRowsIS`, es necesario indicar a la matriz que pueden haber elementos no nulos en celdas que previamente estaban vacias. Para indicarlo, se llama también a `MatSetOption`, pero con la opción `MAT_NEW_NONZERO_LOCATION_ERR`.

Dejando de lado estas funciones de administración de matrices, en el código se usan otros métodos que realizan operaciones matemáticas más complejas. Por ejemplo, la función `MatMatMult` recibe dos matrices y las multiplica, además de un puntero a la matriz donde se guardará el resultado. Otra función es `MaxAXPY`, que recibe dos matrices A y B, y un valor v, y realiza la operación $A + vB$, sobrescribiendo el resultado en A (en nuestro caso, le pasamos matrices `A_0`, `A_1`, y el valor -1, por lo que realiza la operación $A_0 - A_1$). Además, en varias ocasiones es necesario crear una matriz identidad, para lo cual se combina las funciones `MatZeroEntries`, que coloca ceros en todas las celdas de la matriz que reciba, y `MatShift`, que suma un valor a los elementos de la diagonal de la matriz (en este caso, el valor 1). Por otro lado, también se utiliza la función `MatScale`, que multiplica una matriz por un valor.

Sin embargo, una de las operaciones más complejas del algoritmo es la función `KSPMatSolve`, que en nuestro caso, recibe un objeto `KSP`, una matriz `A_2`, una `F` y otra `A_4`, y resuelve el sistema $A_2X = F$, almacenando el resultado `X` en `A_4`. Esta operación se lleva a cabo en el calculo del complemento de Schur, que una parte, incluye la multiplicación de una inversa por otra matriz (`A_3`). En un principio se pensó en calcular este producto directamente, resolviendo el sistema $A_2X = A_3$, pero esto tenía un inconveniente. Debido a la implementación de la operación `KSPMatSolve`, la matriz de la derecha de la ecuación tenía que ser densa, lo cual implicaba crear una matriz A3 de tamaño gxp (g es el número de puntos fantasma y p es el de puntos físicos, $g \ggg p$, ya que los puntos fantasma son solo el borde de la malla, y los físicos son el resto). Como se mencionó, las matrices densas pueden consumir muchos recursos si son muy grandes, que fue el caso al aplicar esta opción, ya que las matrices eran demasiado grandes y la maquina en la que se ejecutaba

el código no tenía suficiente memoria. Sin embargo, al calcular primero la inversa y luego realizar la multiplicación, solo se requería una matriz F de tamaño gxg, de mucho menor tamaño, y que no presentó complicaciones.

Para crear las submatrices de A del complemento de Schur, se emplea la función `MatCreateSubMatrix`, que recibe una matriz, unos `IS` de filas y de columnas, y crea la submatriz de esta en dichas filas y columnas. Por otro lado, para crear las matrices densas, se emplea el método `MatCreateDense`, que recibía un comunicador de procesos, el número de filas y columnas locales y globales, y un puntero a la matriz donde se almacenaban. Aparte, como `KSPMatSolve` es la única función que requería de matrices densas, estas se pueden volver a convertir a matrices dispersas, para lo que se utiliza la función `MatConvert`. También se utiliza para convertir algunas de las submatrices a densa, y recibe como parámetro la matriz a convertir, el tipo a lo que lo queremos convertir (`MATDENSE` para convertir a matriz densa, y `MATAIJ` para matriz dispersa).

En relación al objeto `KSP`, primero se crea con la función `KSPCreate`, y para indicar la matriz del lado izquierdo de la ecuación, se emplea el método `KSPSetOperators`, que recibe el objeto `KSP` y dos matrices (en nuestro caso, ambas son *A_2*, ya que una será la matriz que se incluirá en la ecuación y la otra la que se usará en el precondicionado). Para indicar que se quiere aplicar precondicionamiento, se emplea la función `KSPGetPC` para obtener el objeto `PC` de nuestra instancia `KSP`, y se indica por un lado, que se quiere aplicar el precondiconador una única vez al principio de la resolución del problema (mediante la función `KSPSetType` y el parámetro `KSPPREONLY`), y por otro, que tipo de precondición se quiere utilizar (con la función `PCSetType`, con la variable `PCLU` para expresar que se quiere aplicar factorización LU).

A pesar de esto, la operación más compleja de todo el código es la resolución y el calculo del problema de autovalores, que es llevado a cabo al final por el objeto `EPS` de la librería SLEPc, al ejecutar `EPSSolve`. Para utilizarlo, el primer paso a llevar a cabo es crear el objeto mediante la función `EPSCreate`, que recibe un comunicador y el puntero al objeto. Después, se le pasan las matrices del problema, mediante la función `EPSSetOperators`, que tiene que recibir dos matrices. La segunda matriz es la que estará a la derecha de la ecuación, en caso de que sea un problema generalizado de autovalores, mientras que si es un problema estándar no hay que pasar explícitamente esta matriz (se pasa el parámetro `NULL`). En nuestro caso es un problema generalizado, por lo que las matrices son distintas (A_0 a la izquierda y B_0 a la derecha). Así, para indicar el tipo de problema, se emplea `EPSSetProblemType`, que recibe una cadena con el tipo de problema (`EPS_GHEP`

indica que es un problema hermitiano generalizado). Finalmente, también se tiene que definir las dimensiones del problema, mediante la función `EPSSetDimensions`, que recibe tres valores: el primero es el número de autovalores a calcular (`M`), y los otros dos son el número de dimensiones permitidas del problema proyectado y el número de dimensiones del subespacio, que en nuestro caso no tenían relevancia (por ello, se pasa la variable `PETSC_DEFAULT`, para que SLEPc elija los valores más apropiados).

A parte de establecer las características del problema, también se llevan a cabo una serie de parametrizaciones para optimizar el rendimiento. Por un lado, se aplica *shift-and-invert* sobre el problema, primero obteniendo el objeto ST de *solver* EPS (mediante `EPSGetST`), después indicando el tipo de transformación espectral mediante `STSetType` y pasando la constante STSINVERT, y finalmente indicando el valor de sigma que se empleará, mediante las funciones `EPSSetWhichEigenpairs` y `EPSSetTarget`. Con esta primera función podemos indicar que queremos emplear un valor específico para acelerar la convergencia (`EPS_TARGET_REAL`), y con la segunda función indicamos el valor.

En relación a la configuración del `EPS`, se realizan una últimas modificaciones. Por un lado, se aplican factorización LU sobre el problema, mediante `PCSetType`, como con el objeto `KSP`. Para obtener el precondicionador `PC` del *solver* `EPS`, primero hubo que obtener el *solver* lineal del transformador espectral mediante `STGetKSP`, y después utilizar la función `KSPGetPC`. Sin embargo, la modificación más importante consiste en la aplicación del funcionamiento de la librería MUMPS. Esta librería proporciona *solvers* directos, como LU o Cholesky, para matrices distribuidas. Por ello, se emplea la función `EPSSetFromOptions`, en combinación con el comando `-st_pc_factor_mat_solver_type mumps`. El comando indica que paquete se empleará para ejecutar el *solver*, y la variable `mumps` hace referencia a la librería MUMPS. Por otro lado, la función sirve para especificar que se quiere configurar el objeto `EPS` según los comandos que se pasen en la ejecución.

Finalmente, para dibujar las figuras, es necesario llevar una serie de pasos. Primero, se han de crear el visualizador `PETSC_VIEWER` y el dibujador gráfico `PETSC_DRAW`, con las funciones `PetscViewerDrawOpen` y `PetscViewerDrawGetDraw`, además de indicar que se quiere realizar un dibujo de contorno, mediante `PetscViewerPushFormat`, y la variable `PETSC_VIEWER_DRAW_CONTOUR`.

Más adelante, se crea el objeto `DMDA` de dos dimensiones, con el método `DMDACreate2d`, y se configura con `DMSetUp`. Aparte, mediante la función `DMCreateGlobalVector`, se pasa un objeto `DMDA` y un vector y se enlazan, de forma que cada vez que se visualice el vector, se dibujará automáticamente mediante el objeto `DMDA`.

Después, se comprueba el número de figuras que se han calculado y que hay que dibujar, ya que es posible que el `EPS` no haya calculado exactamente un número `M` de autovectores. El número real se obtiene mediante `EPSGetConverged`. Sabiendo esto, podemos obtener cada autovector mediante `EPSGetEigenvector`, que recibe el índice del autovector que pedimos y un vector Vec donde se almacenará. Finalmente, visualizamos el vector empleando `VecView`, que como se ha mencionado, empleará el visualizador configurado anteriormente.

Debido a que C no incluye liberación automática de recursos, es necesario que destruyamos manualmente todos los objetos que hemos ido creando. Por ello, PETSc incluye métodos con esta funcionalidad para todos sus objetos, como `MatDestroy` para las matrices, `ISDestroy` para los sets de índices, y `EPSDestroy` para los *solvers* de problemas de autovalores.

5.3 Paralelización

Debido a la utilización de exclusivamente funciones de la librería PETSc, realizar la paralelización del algoritmo es relativamente sencillo. La mayoría de métodos, como `MatSetValues`, `MatMult`, `KSPMatSolve` o `EPSSolve` son capaces de administrar automáticamente varios procesos, detectando cuando un proceso requiere de cierta información que posee otro proceso, y realizando las comunicaciones mediante MPI. Sin embargo, no todas las funciones son así, y en ciertas partes del código se llevan a cabo una serie de pasos para garantizar la escalabilidad del código.

En el código original en Matlab, se creaba una matriz `D` con los coeficientes de la discretización, y esta se copiaba a dos matrices `N` y `L` las cuales eran alteradas siguiendo las condiciones de frontera. En el nuevo código, las matrices `N` y `L` son creadas e inicializadas directamente, y sus valores son introducidos en un bucle mediante `MatSetValue`, similar a `MatSetValues`, con la diferencia de que solo recibe un índice de fila y de columna, y un valor. Aunque un proceso puede llamar a esta función para introducir un valor en una fila que pertenezca a otro proceso, lo más eficiente es que los procesos introduzcan valores solo en sus filas. Para controlar esto, se llama a la función `MatGetOwnershipRange`, que devuelve el rango de filas de una matriz que pertenecen al proceso que la ejecuta (los límites se guardan en las variables `Istart` e `Iend`). De esta forma, sabemos donde tiene que empezar cada proceso en el bucle y donde terminar. Además, los rangos de ambas matrices serán iguales para cada proceso, ya que en ambas han sido procesadas

con `MatSetSizes`, en las que hemos indicado que el número de filas y columnas locales son `PETSC_DECIDE`: esto hace que el compilador decida que filas tendrá cada proceso, y mientras el tamaño de las matrices sean iguales, los rangos también lo serán. Por esto, al modificar los valores de la matriz `B`, como esta tiene el mismo tamaño que `N` y `L`, no hace falta calcular de nuevos los rangos,

Sin embargo, en algunos casos hubo que concretar el número de filas y de columnas locales en cada proceso, como era el caso de los sets de índices y las matrices densas. Cuando se crean estos objetos, se les pasan los valores `sp` y `sr2`, que representan el número de puntos (filas) físicos y fantasma locales, respectivamente. Ambos valores se calculan al establecer los valores de `N` y `L`, ya que en el bucle se puede identificar fácilmente en que momento cada proceso esta trabajando con un punto físico y cuando con un punto fantasma: sabiendo la fila en la matriz, podemos calcular la fila y la columna del punto en la malla, y saber que tipo de punto es. Aparte de esto, al crear los sets de índices también se requiere del array de valores locales del proceso. Por ello, se crean unos vectores `p` y `g2`, que incluyen los puntos físicos y fantasmas respectivamente de cada proceso, y se crean de forma similar a como se inicializan los valores `sp` y `sr2`.

Por último, para optimizar el procesamiento, cuando se modifican los valores de las matrices con `MatSetValues`, se controla de forma que cada proceso modifique valores de filas que le pertenecen, en la medida de lo posible. Esto es el caso al cambiar las matrices `L`, `A` y `B`, y por lo explicado anteriormente, los procesos tienen los mismo rangos en todas las matrices, por lo que dichos rangos solo se tienen que obtener una vez.

CAPÍTULO 6

Análisis de la solución en PETSc

En este capítulo, se analiza el código desarrollado en C [17] con la librerías de PETSc y SLEPc, y se estudia su escalabilidad, con el fin de comprobar si es o puede llegar a ser más eficiente que el código original desarrollado en Matlab.

6.1 Parámetros analizados

Al igual que en el capitulo 4, en este apartado se estudia la variación del rendimiento de la nueva solución según el tamaño de malla y el número de autovalores a computar. Además, debido a que el código en PETSc se puede ejecutar en paralelo, que es donde reside su fortaleza, en este capítulo también se comprueba la escalabilidad del código ejecutándolo con distintos números de procesos.

En un principio, también se planeó estudiar los tiempos de computación según el tipo de problema de autovalores, pero esto al final no resultó viable. Para utilizar la versión estándar del problema, se debe eliminar todo el procesamiento de las matrices `A_0`, `B` y `B0`, y cambiar el tipo de problema del *solver* `EPS` a un problema estándar hermitiano, con la llamada

```
EPSSetProblemType(eps,EPS_HEP);
```

Aunque esta versión realiza menos operaciones antes de llamar al `EPS`, el número de puntos a calcular es mayor, ya que también se tienen en cuenta los puntos fantasma. Y, en este código, la parte que más recursos consume es la del *solver* de autovalores, por lo que se observó que los tiempos acababan siendo mayores que con la versión generalizada. Por lo que esta versión del algoritmo se descartó. También se consideró cambiar el valor de sigma en la transformación espectral, pero se comprobó que el código no era eficiente con un valor de sigma cero (como la versión original), pero si con un valor de -0.01.

Para cambiar el valor de sigma, simplemente había que cambiar el valor en la llamada a `EPSSetTarget`.

Es debido a estas razones por las que, en el estudio, el nuevo código solo se ejecuta con la versión generalizada del problema y un valor de sigma -0.01

6.2 Metodología

Para este estudio, se ha medido el tiempo de ejecución para la función encargada del cálculo de autovalores. Todas las ejecuciones, tanto las secuenciales como las paralelas, se han llevado a cabo en el clúster *Kahan*, también proporcionado por la UPV y cuyas propiedades se pueden encontrar en el apéndice A. Esta cuenta con dos partes: el front-end y el clúster de 4 nodos. Los usuarios solo pueden acceder por el front-end por lo que, para enviar los comandos al clúster, en este trabajo se ha creado un script y este se ha enviado con el comando `sbatch` a dicho clúster, el cual ejecutará las instrucciones en el script. En total se ha probado a procesar el código con 1, 2, 4, 8, 16, y 32 procesos.

El código ha sido compilado en C, con las librerías PETSc y SLEPc instaladas en las propias máquinas. De igual forma que con el anterior estudio, el código se ha ejecutado con tamaños de malla de 100, 200, 300, 400, 500, 600, 700, 800, 900 y 1000 puntos. También se ha probado con distintos números de autovalores a computar: 50, 100 y 150 autovalores. Además, tampoco se ha tenido en cuenta el tiempo de procesamiento para dibujar las figuras resultantes.

Para llevar a cabo todas la ejecuciones, en el script que se lanzó al clúster se incluyó la instrucción que se muestra a continuación:

```
mpiexec -n <procesos>./chladniPetsc2 -st_pc_factor_mat_solver_type mumps
```

El comando *mpiexec* indica al sistema que se quiere ejecutar un programa MPI. La opción `-n` permite indicar con cuantos procesos queremos que se ejecute el código (*procesos*). Por otro lado, `chladniPetsc2` es el nombre del archivo con la compilación del nuevo código desarrollado, y la última opción, como se indicó en el anterior capítulo, sirve para indicar que queremos emplear la librería MUMPS para poder utilizar matrices dispersas en el *solver* de autovalores EPS.

Por último, previamente a realizar todas estas pruebas, se compilaron las librerias PETSc y SLEPc para que funcionaran en optimizado. Para ello, se ejecuto un script incluido en la librería de PETSc de nombre `configure`, que es con el que se puede compilar dicha librería y pasarle todos los parámetros de configuración. En nuestro caso, se le pa-

só la opción `-with-debugging=no` que indica que se configure el modo optimizado de la librería. En la llamada también se incluyó la opción `-download-mumps`, que indica que se descargue la librería MUMPS mencionada en el anterior capitulo. Finalmente, para configurar SLEPc, tambíen se empleo un script incluido de nombre similar incluido en la carpeta de la librería y de funcionalidad similar. Sin embargo, como heredaba la mayoría de atributos de PETSc al ser una extensión de esta, no hizo falta pasarle ningún parámetro al script.

Además de toda esta configuración mencionada, se tuvieron que crear dos variables de entorno en el sistema para el correcto funcionamiento de SLEPc. Concretamente, se declararon las variables `PETSC_DIR` y `SLEPC_DIR`, y se les asignaron las direcciones de las carpetas de las librerías PETSc y SLEPc respectivamente. Estas variables indican al compilador donde puede encontrar el código fuente de dichas libreras.

6.3 Resultados

6.3.1. Ejecuciones secuenciales

Las pruebas de esta versión del algoritmo han sido ejecutadas en la máquina Kahan, y al igual que la solución en Matlab, solo se ejecutaron con un proceso. Los resultados se recogen en la Tabla 6.1 y también se muestran en las gráficas de la Figura 6.1, para observar que clase de evolución tiene esta nueva implementación del algoritmo.

Tabla 6.1: Medidas de tiempo del algoritmo secuencial en Kahan

n	100	200	300	400	500	600	700	800	900	1000
k=50	0.93	3.999	10.121	15.678	32.096	42.702	63.291	92.085	120.53	168.88
k=100	1.613	7.306	17.698	32.68	54.185	83.195	117.76	203.63	267.42	350.21
k=150	3.327	14.878	35.393	64.434	105.34	157.86	193.82	289.97	370.52	454.74

Como se puede observar, el tiempo de procesamiento sigue aumentando de forma casi cuadrática con respecto al tamaño de malla, sin importar el número de autovalores que se calculen. Esto es igual a lo que ocurría en la solución original en Matlab, a pesar de estar implementado en un lenguaje diferente con otras librerías. Hecho esto, el siguiente paso sería observar cuanto puede mejorar el rendimiento de la nueva solución al emplear paralelismo, y como va evolucionando.

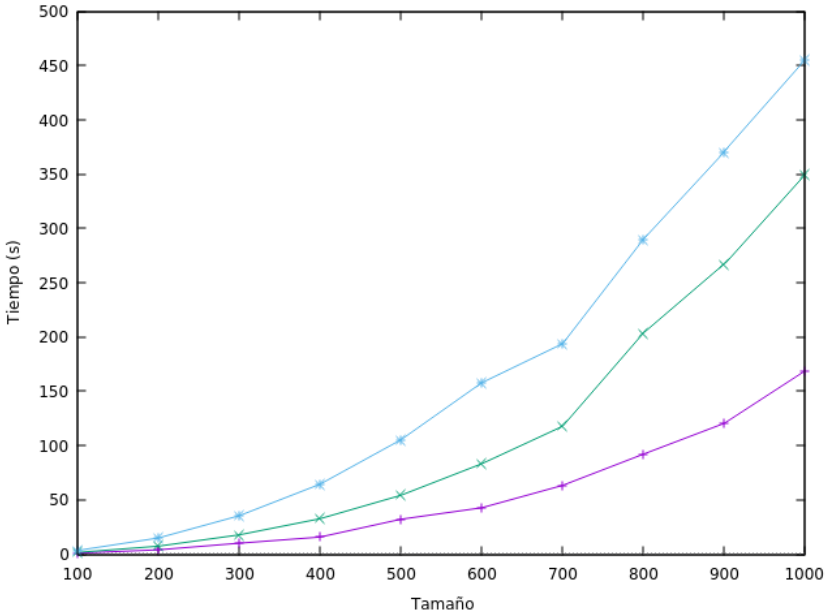


Figura 6.1: Resultados del algoritmo con un único proceso en Kahan. En la línea morada se encuentran los tiempos para calcular 50 autovalores, en la línea verde para calcular 100 autovalores, y en la línea azul para 150 autovalores

6.3.2. Ejecuciones paralelas

En este apartado se va a probar a ejecutar el código empleando paralelismo, con un número creciente de procesos, empezando por **dos**. El objetivo sería comparar el rendimiento del código paralelo con el del código secuencial en Kahan. Por esto, a continuación se muestran tanto los tiempos registrados de las ejecuciones, así como los *speed-ups* alcanzados, en las Tablas 6.1 y 6.2. Además, en la Figura 6.2, se muestra la evolución de las mejoras alcanzadas con respecto a los parámetros.

Tabla 6.2: Medidas de tiempo del algoritmo con dos procesos

n	100	200	300	400	500	600	700	800	900	1000
k=50	0.659	2.295	5.528	12.972	18.316	28.407	47.218	62.549	83.657	124.72
k=100	1.257	5.513	13.026	24.659	40.26	61.537	101.25	139.46	190.93	243.27
k=150	1.969	8.402	19.434	35.966	57.751	85.981	141.52	191.34	293.99	380.64

Tabla 6.3: Speed-ups del algoritmo con dos procesos en comparación al secuencial

n	100	200	300	400	500	600	700	800	900	1000
k=50	1.41	1.742	1.831	1.209	1.752	1.503	1.34	1.472	1.441	1.354
k=100	1.284	1.325	1.359	1.325	1.346	1.352	1.163	1.46	1.401	1.44
k=150	1.69	1.771	1.821	1.792	1.824	1.836	1.37	1.515	1.26	1.195

Como se puede observar, los tiempos se han reducido en comparación a las pruebas secuenciales que se mostraron anteriormente. Sin embargo, no se llegan a alcanzar *speed-ups* de dos, que era lo ideal en este caso al haber empleado dos procesos. Aún así, en todos

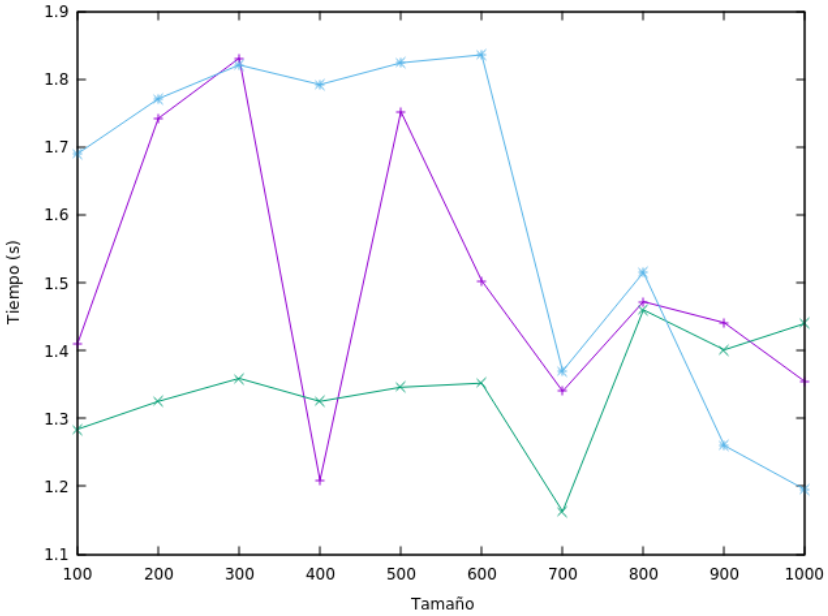


Figura 6.2: Speed-ups alcanzados con dos procesos. En la línea morada se encuentran las mejoras al calcular 50 autovalores, en la línea verde se encuentran las mejoras al calcular 100 autovalores, y en la línea azul se encuentran las mejoras al calcular 150 autovalores

los casos el *speed-up* es mayor que uno, indicando que ha habido mejora con respecto a las ejecuciones originales.

Tras esto, vamos a seguir añadiendo procesos para ver como evoluciona el rendimiento del código. Por ello, ahora procedemos a utilizar **cuatro procesos**. A continuación se muestran los resultados en las Tablas 6.4 y 6.5, junto con las gráficas de la Figura 6.3, al igual que en el anterior caso.

Tabla 6.4: Medidas de tiempo del algoritmo con cuatro procesos

n	100	200	300	400	500	600	700	800	900	1000
k=50	0.572	2.876	5.276	10.238	16.662	23.038	39.624	53.287	84.843	110.97
k=100	0.928	4.298	9.415	18.062	28.737	39.496	62.404	92.23	152.55	226.36
k=150	1.684	7.231	15.497	29.758	47.062	65.544	110.23	155.34	247.16	320.34

Tabla 6.5: Speed-ups del algoritmo con cuatro procesos en comparación al secuencial

n	100	200	300	400	500	600	700	800	900	1000
k=50	1.625	1.39	1.919	1.531	1.926	1.854	1.597	1.728	1.421	1.522
k=100	1.739	1.7	1.88	1.809	1.886	2.106	1.887	2.208	1.753	1.547
k=150	1.976	2.058	2.284	2.165	2.238	2.408	1.758	1.867	1.499	1.42

Al usar cuatro procesos, se puede apreciar una mejora con respecto a anteriores ejecuciones. Aunque la ninguno de los casos no alcanzan un *speed-up* de cuatro, todos siguen teniendo una mejora positiva, y en algunos casos, se ha visto una mejora del 100 %. Además, se puede ver que la gran mayoría de casos tienen un mejor rendimiento que cuando

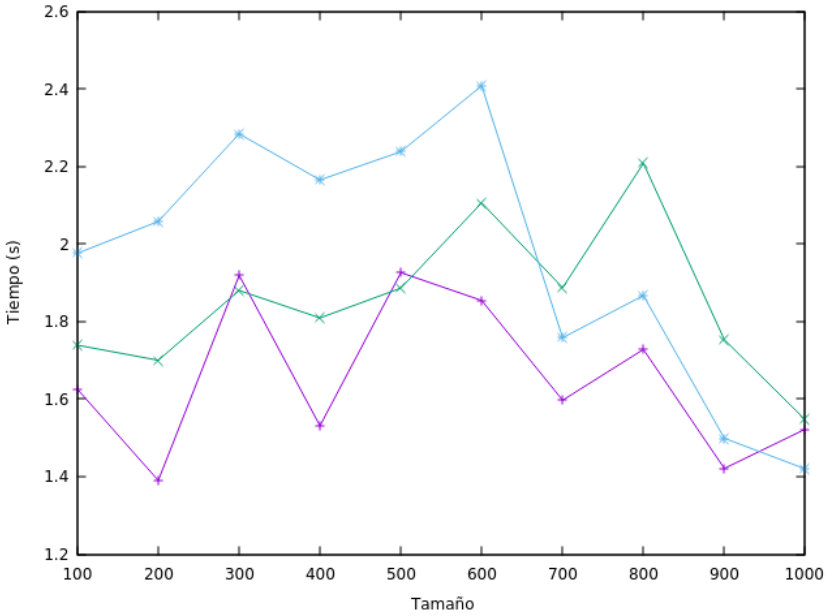


Figura 6.3: Speed-ups alcanzados con cuatro procesos. En la línea morada se encuentran las mejoras al calcular 50 autovalores, en la línea verde se encuentran las mejoras al calcular 100 autovalores, y en la línea azul se encuentran las mejoras al calcular 150 autovalores

se realizaron las ejecuciones con dos procesos. Esto indica que el código sigue mejorando en rendimiento a medida que aumentamos los procesos.

Ahora procedemos a ejecutar todos los casos con **ocho procesos**. Los resultados de estas ejecuciones se muestran en las Tablas 6.6, y los *speed-ups* de cada caso y su evolución se muestran en la Tabla 6.7, y Figura 6.4, respectivamente.

Tabla 6.6: Medidas de tiempo del algoritmo con ocho procesos

n	100	200	300	400	500	600	700	800	900	1000
k=50	0.444	1.763	4.544	8.167	13.618	20.693	29.167	39.043	55.877	65.257
k=100	0.84	4.562	10.849	19.602	32.086	46.333	66.047	87.878	122.36	147.64
k=150	1.393	6.594	15.354	27.622	46.083	68.084	94.76	124.28	168.29	210.19

Tabla 6.7: Speed-ups del algoritmo con ocho procesos en comparación al secuencial

n	100	200	300	400	500	600	700	800	900	1000
k=50	2.093	2.268	2.227	1.92	2.357	2.064	2.17	2.359	2.157	2.588
k=100	1.92	1.602	1.631	1.667	1.689	1.796	1.783	2.317	2.186	2.372
k=150	2.389	2.256	2.305	2.333	2.286	2.319	2.045	2.333	2.202	2.163

En esta ocasión, el algoritmo logra alcanzar *speed-ups* de dos en la mayoría de casos, y en el resto de casos alcanza una mejora de al menos un 50 %. Esto nos indica además que con este número de procesos, el código aun sigue experimentando mejora. Por otro lado, en las gráficas podemos apreciar, especialmente con los casos de 100 autovalores, que las mejoras alcanzadas van aumentando a medida que va aumentando el tamaño de malla.

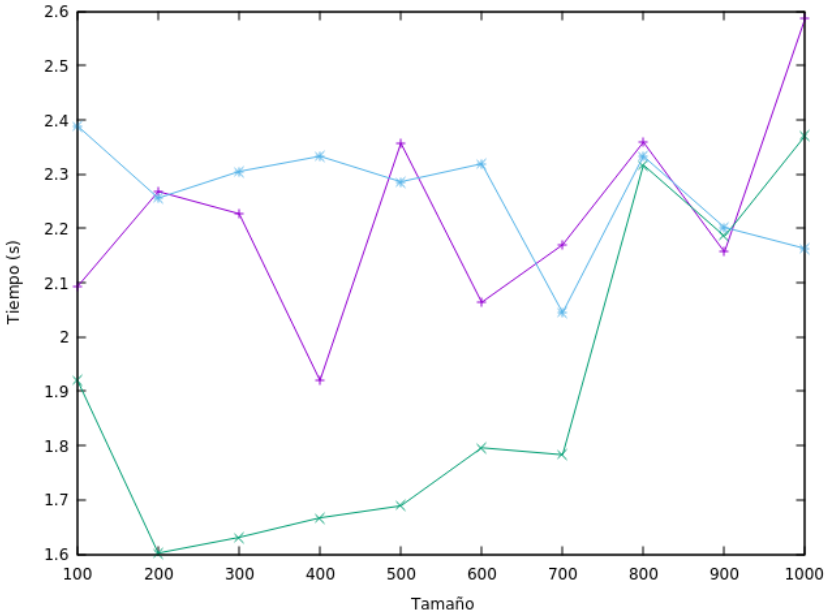


Figura 6.4: Speed-ups alcanzados con ocho procesos. En la línea morada se encuentran las mejoras al calcular 50 autovalores, en la línea verde se encuentran las mejoras al calcular 100 autovalores, y en la línea azul se encuentran las mejoras al calcular 150 autovalores

Con esto, podemos concluir que con ocho procesos, el nuevo código sigue mejorando con respecto a las ejecuciones anteriores. Por ello, probaremos con **dieciséis procesos**, y comprobaremos si los *speed-ups* son mayores, analizando los resultados de las Tablas 6.8 y 6.9, y observando las gráficas de la Figura 6.5 para comprobar la evolución.

Tabla 6.8: Medidas de tiempo del algoritmo con dieciséis procesos

n	100	200	300	400	500	600	700	800	900	1000
k=50	0.383	1.648	3.41	5.984	9.654	13.918	19.916	25.99	35.794	45.248
k=100	0.603	2.628	6.347	11.066	17.654	25.616	34.934	46.823	62.331	79.55
k=150	0.975	4.343	10.292	17.372	27.893	40.277	54.52	73.568	97.058	123.09

Tabla 6.9: Speed-ups del algoritmo con dieciséis procesos en comparación al secuencial

n	100	200	300	400	500	600	700	800	900	1000
k=50	2.429	2.426	2.969	2.62	3.325	3.068	3.178	3.543	3.367	3.732
k=100	2.677	2.78	2.788	2.953	3.069	3.248	3.371	4.349	4.29	4.402
k=150	3.412	3.425	3.439	3.709	3.777	3.919	3.555	3.942	3.818	3.694

Aunque las mejoras no son las más óptimas (*speed-ups* de dieciséis), siguen siendo muy elevadas y han aumentando con respecto a las anteriores. En la mayoría de casos hemos alcanzado *speed-ups* mayores de tres, e incluso de cuatro, con respecto a las ejecuciones secuenciales. También podemos seguir observando que, independientemente del número de autovalores a calcular, el *speed-up* es mayor a medida que aumenta el tamaño de nuestra malla.

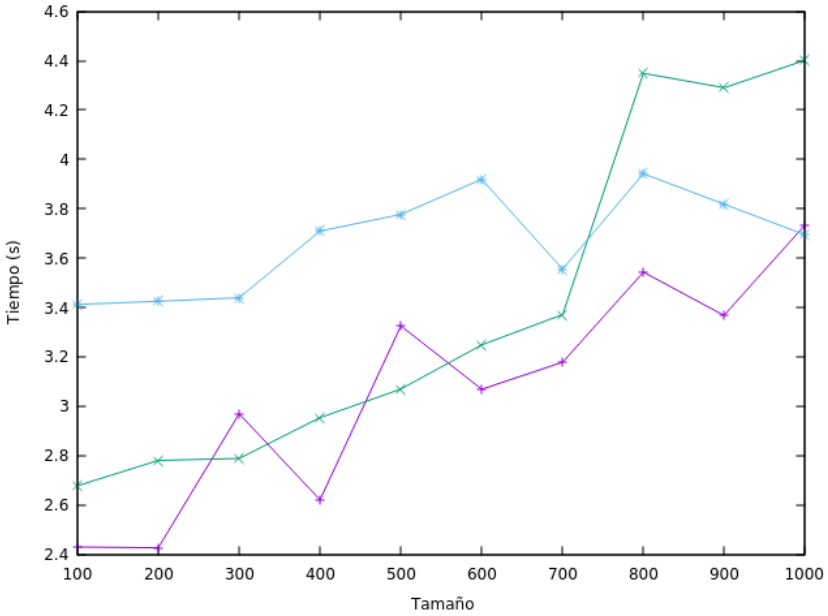


Figura 6.5: Speed-ups alcanzados con dieciséis procesos. En la línea morada se encuentran las mejoras al calcular 50 autovalores, en la línea verde se encuentran las mejoras al calcular 100 autovalores, y en la línea azul se encuentran las mejoras al calcular 150 autovalores

Con esto, podemos concluir que emplear la nueva versión del código con dieciséis procesos es mas eficiente que emplear el código original. Finalmente, comprobaremos el rendimiento del código con treinta y dos procesos, con los resultados que se muestran en la Tablas 6.10, para ver si siguen mejorando, y que el *speed-up* sigue aumentando viendo la Tabla 6.11 y las gráficas de la Figura 6.6.

Tabla 6.10: Medidas de tiempo del algoritmo con treinta y dos procesos

n	100	200	300	400	500	600	700	800	900	1000
k=50	0.389	1.293	2.882	4.951	8.004	11.602	16.518	21.55	28.059	38.928
k=100	0.577	2.082	4.833	8.475	13.516	19.571	27.604	36.06	44.392	56.656
k=150	0.861	3.126	7.346	12.829	20.059	28.962	39.933	52.244	67.385	80.954

Tabla 6.11: Speed-ups del algoritmo con treinta y dos procesos en comparación al secuencial

n	100	200	300	400	500	600	700	800	900	1000
k=50	2.391	3.093	3.512	3.167	4.01	3.68	3.832	4.273	4.296	4.338
k=100	2.795	3.51	3.662	3.856	4.009	4.251	4.266	5.647	6.024	6.181
k=150	3.864	4.76	4.818	5.023	5.252	5.45	4.854	5.55	5.499	5.617

Finalmente, podemos observar que el rendimiento del código sigue aumentando, esta vez incluso alcanzando *speed-ups* de cinco y seis en algunos casos. También se puede ver que de igual forma que antes, a medida que se va aumentando el tamaño de malla, la mejora del código va aumentando con ello.

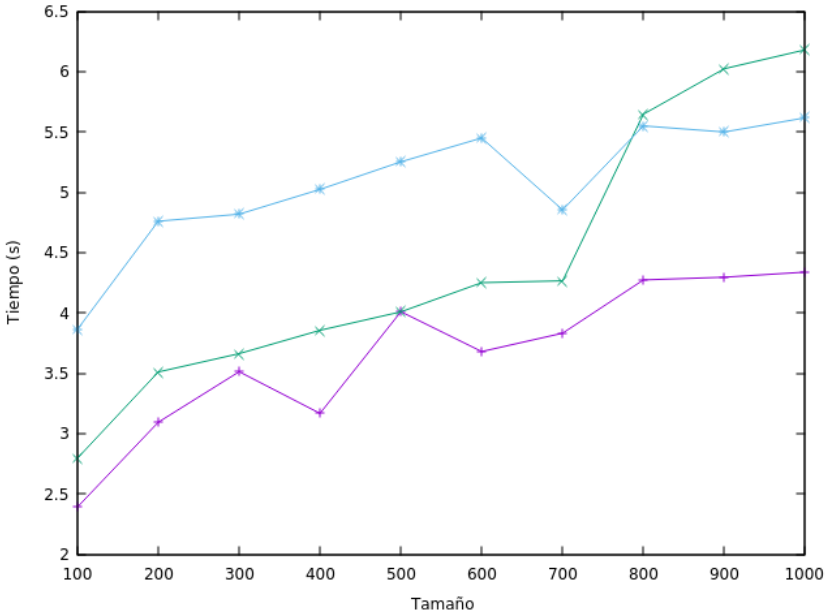


Figura 6.6: Speed-ups alcanzados con treinta y dos procesos. En la línea morada se encuentran las mejoras al calcular 50 autovalores, en la línea verde se encuentran las mejoras al calcular 100 autovalores, y en la línea azul se encuentran las mejoras al calcular 150 autovalores

6.3.3. Figuras

Para terminar, podemos comprobar que las figuras resultantes (Figura 6.7) siguen siendo similares a las figuras originales, a pesar de estar usando un algoritmo distinto en un nuevo lenguaje, y estamos generándolas con un entorno gráfico distinto. Esto indica que el nuevo código aparte de ser eficiente, es una alternativa para resolver el problema de las figuras de Chladni, el cual era nuestro objetivo principal.

En la imagen se muestran cuatro de las figuras más representativas, equivalentes a las que se muestran en el capítulo 4. Por otro lado, en el apéndice B se pueden observar un mayor número de muestras obtenidas por el algoritmo, que se pueden comparar a las mostradas en la Figura 4.2 del artículo original de Gander y Kwok sobre la solución en Matlab [2].

6.4 Conclusiones del análisis

Como se ha podido observar, el código no alcanza la máxima escalabilidad teórica. Cuando se ejecutaba con treinta y dos procesos, la mejora máxima que se alcanzaba era de trece. A pesar de esto, el rendimiento del código seguía mejorando a medida que se han ido aumentando los procesos, lo cual nos indica que su rendimiento podría seguir incrementándose en caso de que fueramos añadiendo más procesos a la ejecución.

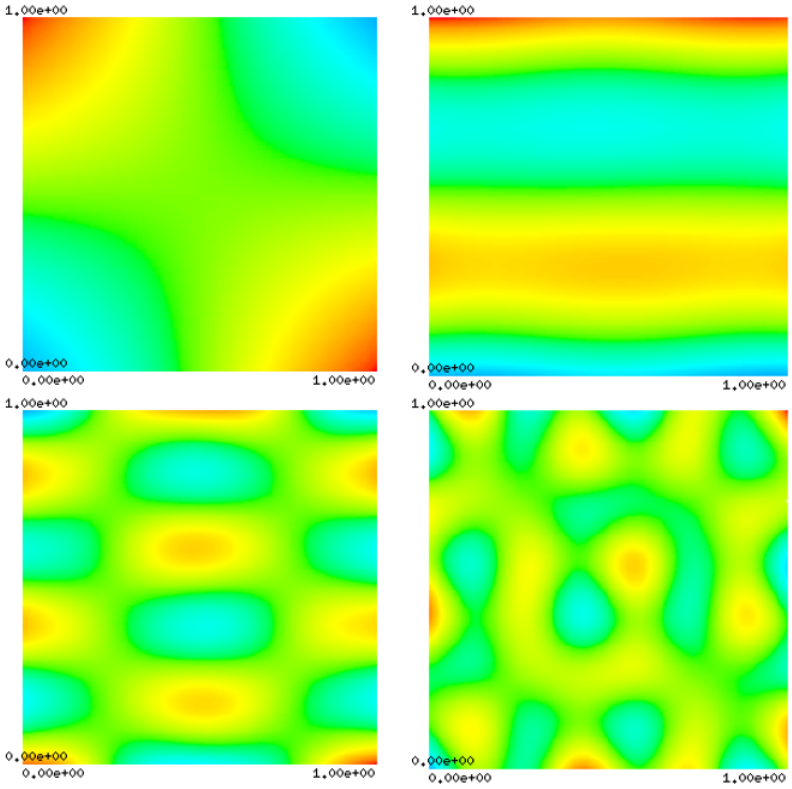


Figura 6.7: Figuras correspondientes a los autovalores 4 (izquierda superior), 10 (derecha superior, 30 (izquierda inferior) y 50 (derecha inferior).

Por otro lado, se ha visto que cuando se han añadido suficientes procesos, a medida que el tamaño de malla iba aumentando, también lo hacían las mejoras alcanzadas, con lo cual se podría concluir que cuanto más grande sea la malla de nuestro caso de estudio, mayor mejora se alcanzara. Aunque se puede ver en las gráficas que este suceso es más apreciable con los casos de 50 autovalores (en los de 150 hay cierta caída con los mayores tamaños), esto es aceptable ya que de normal, en caso de querer calcular un determinado número de figuras, se querrán calcular principalmente las primeras, que son las que tienen los menores autovalores asociados, ya que son las más precisas.

CAPÍTULO 7

Conclusiones

En conclusión, emplear nuestra versión del algoritmo para formar figuras de PETSc es una alternativa viable con respecto al algoritmo original desarrollado en Matlab, al proporcionar resultados prácticamente idénticos y poder alcanzar un buen rendimiento, lo cual era el objetivo principal de este trabajo. Además, aplicando paralelismo y empleando un número de procesos creciente, se ha logrado conseguir una mejora cada vez mayor. Y como a día de hoy, la tecnología ha avanzado lo suficiente y la mayoría de máquinas tienen los suficiente recursos para tener un gran número de procesos ejecutándose al mismo tiempo, alcanzar estas mejoras es completamente viable.

Además, en este estudio se ha demostrado la eficacia de la computación paralela: Matlab es un lenguaje matemático numérico muy potente, y es de los lenguajes más utilizados en universidades y entidades públicas para llevar a cabo simulaciones. Por otro lado, el lenguaje C es mucho menos potente en este aspecto, y tiene más inconvenientes, como mayor complejidad de administración de memoria. Sin embargo, gracias a la implementación de las librerías PETSc y SLEPc y como hacen el máximo uso de la computación paralela, se ha logrado optimizar el código en C, y mejorar su rendimiento considerablemente, lo cual nos hace reflexionar sobre la eficacia de estas técnicas, y los resultados que se podrían obtener con un lenguaje igual de robusto a Matlab en simulaciones, si además este fuera ejecutado empleando paralelismo (Matlab cuenta con librerías de computación paralela, pero la función responsable del calculo de autovalores aun no es paralelizable).

Para la realización de este trabajo, todas las asignaturas de mi carrera universitaria han tenido algún tipo de influencia, ya que todas han servido para formarme como ingeniero y realizar este trabajo. A pesar de esto, hay dos asignaturas que han tenido un especial significado en este estudio. La primera es la asignatura de *Computación Paralela*, en la cual se nos enseñó la base del paralelismo, los principios de diseño para coordinar

procesos y el acceso controlado de memoria, además de otros contenidos más específicos, como el uso del estándar MPI y el concepto de comunicadores entre procesos. Todo esto tuvo importancia a la hora de desarrollar la solución paralela.

Sin embargo, la asignatura de mayor relevancia en este proyecto ha sido la asignatura de *Computación científica*. En esta se nos enseñó como se podían simular y representar problemas científicos empleando la computación, así como algunos técnicas matemáticas como el de la discretización de ecuaciones de derivadas parciales, o los métodos de Krylov para sistemas lineales. Además. en esta asignatura se nos enseñaron las principales funcionalidades de PETSc y SLEPc, y como se podían utilizar para realizar dichas simulaciones. Estas enseñanzas tuvieron una gran importancia a la hora de desarrollar el código paralelo. Además, en esta asignatura revisamos como llevar a cabo un estudio completo y detallado de rendimiento de código, que también ha servido para los apartados de análisis de este trabajo.

Por otro lado, para este trabajo ha sido necesario aprender sobre varias tecnologías. Las más obvias y directas han sido las ya mencionadas librerías de PETSc y SLEPc, que aunque se explicaron en la asignatura de *Computación científica*, igualmente requirieron de un estudio más profundo de sus funcionalidades para poder implementar el código. Las funciones que ofrecían no son *cajas negras* de programación, sino que requieren de cierto conocimiento por parte del programador, no solo de que es lo que hace cada método, sino del trasfondo matemático y científico de las operaciones que realiza. Es por ello por lo que utilizar dichas librerías ha requerido trabajo y esfuerzo. Sin embargo, gracias a la extensa documentación y a los distintos ejemplos ofrecidos por ambas librerías, todas las operaciones que se han intentado hacer han sido posible después de dedicarle suficiente tiempo.

Otra herramienta importante en la realización de este trabajo ha sido el uso de la aplicación gráfica de Matlab, para poder estudiar correctamente el código, depurarlo para revisar su funcionamiento y para poder comprobar correctamente que hacía cada versión del algoritmo antes de realizar las pruebas de rendimiento. Además, a la hora de usar la máquina *Kahan*, ha sido necesario aprender las funcionalidades de las colas *SLURM* (Simple Linux Utility for Resource Management), y saber como escribir scripts que ejecutaran los algoritmos de la forma correcta en el clúster.

A pesar de esto, el mayor reto han sido los distintos problemas que han surgido en relación al uso de memoria del algoritmo. Por ejemplo, al principio del desarrollo del algoritmo en PETSc, hubo problemas con los vectores b y g, que al principio eran punteros

a vectores. Esto se inicializaban con la función `createVec`, que en las primeras versiones trabajaba con punteros de punteros a vectores. Sin embargo, cuando estos vectores se pasaban como parámetros a las funciones de PETSc, no eran leidos correctamente, y los resultados no eran los esperados. Para arreglarlo, se cambio el código a la versión actual, en la que que `b` y `g` son vectores y `createVec` funciona con punteros a vectores.

Otro problema de mayor significado fue el mencionado en el capítulo 5, sobre la falta de memoria al intentar resolver el sistema lineal mediante el *solver* `KSP`. Sin embargo, este problema se pudo solucionar aplicando conocimientos matemáticos, de forma que se pudo emplear otras operaciones que llevaban al mismo resultado que se requería, pero que consumían una cantidad menor de recursos de memoria.

Es por esto por lo que he aprendido varias cosas en la realización de este trabajo. Por un lado, he aprendido a como llevar a cabo un estudio extenso y detallado de un algoritmo, a desarrollar código empleando herramientas muy relevantes en el contexto de computación científica y a como conseguir y obtener información sobre un problema poco conocido, moviendo por las distintas fuentes disponibles. Pero además, una enseñanza que me ha dado este proyecto, comparando los distintos códigos estudiados y a la resolución de los distintos problemas, es que emplear las tecnologías punteras y más novedosas siempre es importante y puede aportar grandes beneficios en el desarrollo de soluciones informáticas. Sin embargo, la mejor herramienta que tenemos como informáticos a la hora de resolver los problemas que se nos presentar es la práctica de buenas técnicas de programación, como ya sea la aplicación de computación paralela o el análisis y desarrollo de los algoritmos más eficientes, lo cual al final es la principal función de un informático y la mayor contribución que podemos dar al avance de la tecnología.

7.1 Trabajos futuros

Una de las fortalezas de este proyecto es que lo que se ha aplicado en este problema se puede emplear en toda clase de problemas científicos diferentes, que podrían haber sido contenido para nuevos trabajos. Empleando las técnicas de computación científica y las librerías de PETSc y SLEPc se podría, por ejemplo, llevar a cabo eficientemente simulaciones de calor sobre una placa metálica, de flujo de radiación o incluso de fenómenos como la subducción de placas tectónicas en un espacio tridimensional. Únicamente sería necesario conocer las fórmulas matemáticas que nos permitirían determinar dichos fenómenos de forma precisa, y a partir de ahí elaborar la solución que los calcule y represente.

Aparte, hay ciertos aspectos de este trabajo que debido a la falta de tiempo no se han podido desarrollar, pero que podrían haber sido de interés. Una idea al principio fue desarrollar una versión en el lenguaje Python, con el fin de poder comparar la nueva solución con más de una solución en secuencial y poder lograr resultados más precisos. Sin embargo, cuando se implementó, se descubrió que el código tenía un rendimiento considerablemente menor que las otras versiones, principalmente porque el *solver* de autovalores del lenguaje tenía un rendimiento inferior. Debido a esto, esta versión tuvo que descargarse del trabajo, pero cabría la posibilidad de que en un futuro se investigara como hacer que funcione más eficientemente. Además, las librerías de PETSc y SLEPc también se pueden utilizar en otros lenguajes como Python, por lo que la versión también se podría implementar de forma paralela y más adelante se podrían comparar ambas versiones para analizar la mejora. Por supuesto, todas las simulaciones que se han mencionado anteriormente también se podrían implementar en Python, no solo en C.

También se consideró hacer pruebas del código empleando una unidad de procesamiento gráfico o GPU, para compararlo con el rendimiento de uno CPU. Esto se podía hacer empleando la librería CUDA (Compute Unified Device Architecture), la cual es soportada por PETSc y SLEPc, y haciendo algunas modificaciones al código. Esto se podría intentar aplicar en un futuro. Por último, como se menciona en este trabajo, el *solver* KSP del código solo funciona con matrices densas, lo cual complicó el desarrollo de la solución y empeoró su eficacia. Un posible proyecto futuro podría ser implementar la clase para que también funcione con matrices dispersas.

Bibliografía

[1] Rossing T. D. Chladni's Law for Vibrating Plates. *Diario Americano de Física*, 50:271–274, marzo, 1982.

[2] Martin J. Gander, Felix Kwok. Chladni Figures and the Tacoma Bridge: Motivating PDE Eigenvalue Problems via Vibrating Plates. *SIAM Review*, 50:3:573–596, agosto, 2012.

[3] Steve Cunningham, Angela B. Shiflet. Computer graphics in undergraduate computational science education. *ACM SIGCSE Bulletin*, 35:1:372–375, enero, 2003.

[4] Ananth Grama, Anshul Gupta, George Karypis, Vipin Kuma. Parallel Programming for Multicore and Cluster Systems. *Addison Wesley*, enero, 2003.

[5] Jonathan Weinberg. Job Scheduling on Parallel Systems. *University of California*, enero, 2006.

[6] David A. Patterson, John L Hennessy. Computer Architecture. A Quantitative Approach (Fifth Edition) *Morgan Kaufmann*, octubre, 2011.

[7] Ananth Grama, Anshul Gupta, George Karypis, Vipin Kuma. Introduction to Parallel Computing, Second Edition. *Addison Wesley*, enero, 2003.

[8] Stewart, G. W. Matrix Algorithms. Volume II: Eigensystems. *SIAM*, agosto, 2001.

[9] Zhongxiao Jia, Yong Zhang. A refined shift-and-invert arnoldi algorithm for large unsymmetric generalized eigenproblemss. *Computers and Mathematics with Applications*, 44:8-9:1117–1127, octubre-noviembre, 2002.

[10] Henk Van der Vorst. Krylov Subspace Iteration. *Computing in Science and Engineering*, 2:1:32–37, febrero, 2000.

[11] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, Junchao Zhang. PETSc/TAO Users Manual. ANL-21/39 - Revision 3.17, Argonne National Laboratory, 2022.

[12] Página Web PETSc. Consultado en https://petsc.org/. Última visita: 08/07/2022

[13] J. E. Roman, C. Campos, L. Dalcin, E. Romero, A. Tomas. SLEPc Users Manual. Tech. Rep. DSIC-II/24/02 - Revision 3.17, Universitat Politècnica de València, marzo, 2022.

[14] V. Hernández, J. E. Roman, A. Tomas, V. Vidal. Krylov-Schur Methods in SLEPc. Tech. Rep.STR-7, Universitat Politècnica de València, junio, 2007.

[15] MPI: A Message-Passing Interface Standard Version 4.0. Consultado en https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. Última visita: 10/06/2022

[16] MATLAB. 9.12.0.1884302 (R2022a). *The MathWorks Inc.*, 2022.

[17] Trevis Rothwell, James Youngman. The GNU C Reference Manual. *Free Software Foundation, Inc.*, 2015.

[18] Fuzhen Zhang. The Schur Complement and Its Applications. *Numerical Methods and Algorithms*, 4:1:1016–1311, diciembre, 2010.

APÉNDICE A

Configuración hardware

Para la realización de este trabajo y llevar a cabo las distintas pruebas de rendimiento, se utilizaron dos máquinas proporcionadas por la UPV. Sus especificaciones se muestran a continuación.

A.1 Clúster Kahan

Este clúster esta formado por 4 nodos comunicados a través de una red Ethernet 10/25Gb. Cada nodo consta de:

- 1 procesador AMD EPYC 7551P de 32 núcleos físicos (64 virtuales)
- 64GB de memoria
- Disco SSD de 240GB

A.2 Servidor GPU

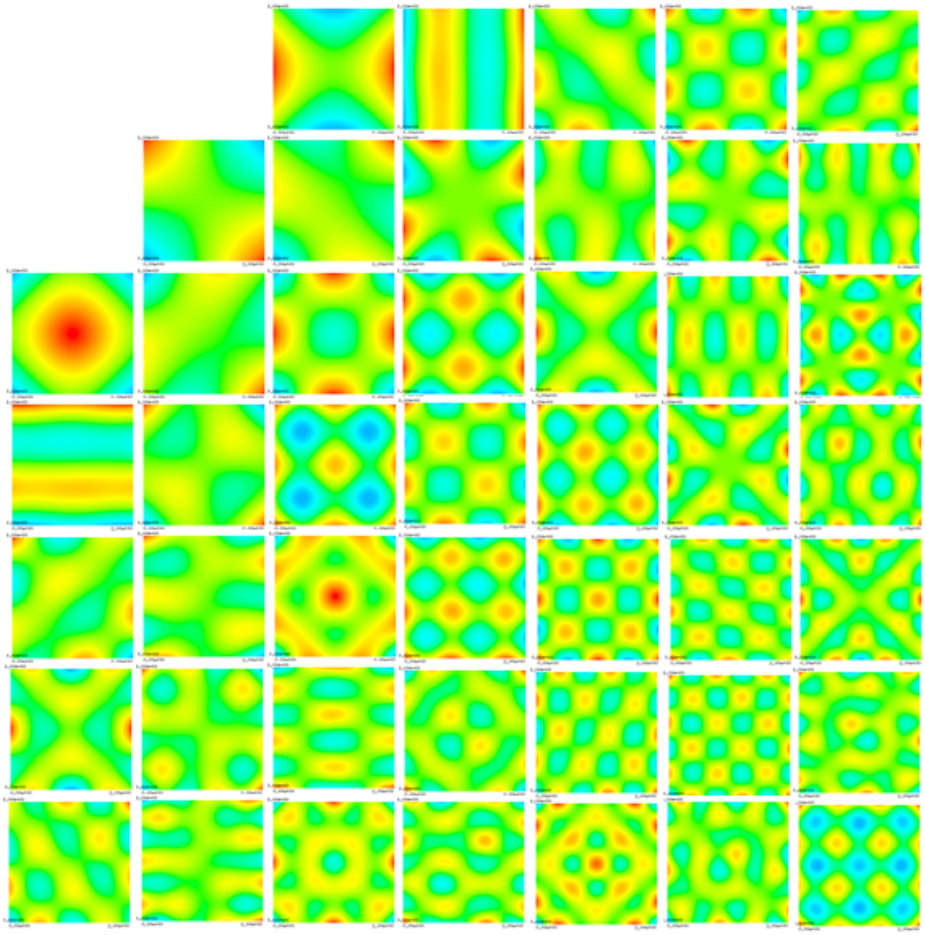
Esta máquina cuenta con un único nodo, cuyas especificaciones son:

- 1 procesador Intel Core i9-7960X de 16 nucleos físicos (32 virtuales)
- 64GB de memoria
- Disco SSD de 240GB

APÉNDICE B

Muestras de figuras

En este apéndice se muestran las primeras 46 figuras generadas por el algoritmo implementado con PETSc y SLEPc.



APÉNDICE C

Código PETSc

```
PetscInt rank, ps, Istart, Iend;

//Inserta los datos en un array a partir de los parametros que le introducimos
//
//@res puntero al vector donde queremos almacenar los datos
//@size tamanyo del vector entre 4
//@start1 primer valor del array
//@start2 valor del array al comienzo del segundo cuarto del array
//@start3 valor del array a la mitad
//@start4 valor del array al comienzo del ultimo cuarto del array
void createVec (PetscInt *res, PetscInt start1, PetscInt start2, PetscInt start3,
    PetscInt start4, PetscInt size) {
    PetscInt val, add, start, end;

    //Aqui introducimos los valores que correspondian a gl y bl en el codigo de matlab
    add = 1;
    val = start1;
    start = 0;
    end = size;
    for (PetscInt i=start; i<end; i++) {
        res[i] = val;
        val = val+add;
    }

    //Aqui introducimos los valores que correspondian a gr y br en el codigo de matlab
    add=1;
    val = start2;
    start = start+(size);
    end = end+(size);
    for (PetscInt i=start; i<end; i++) {
        res[i] = val;
        val = val+add;
    }

    //Aqui introducimos los valores que correspondian a gt y bt en el codigo de matlab
```

```
35      add=size+2;
36      val = start3;
37      start = start+(size);
38      end = end+(size);
39      for (PetscInt i=start; i<end; i++) {
40          res[i] = val;
41          val = val+add;
42      }
43
44      //Aqui introducimos los valores que correspondian a gb y bb en el codigo de matlab
45      add=size+2;
46      val = start4;
47      start = start+(size);
48      end = end+(size);
49      for (PetscInt i=start; i<end; i++) {
50          res[i] = val;
51          val = val+add;
52      }
53  }
54
55  //Algoritmo principal
56  //
57  //@n  tamanyo de la malla
58  //@M  numero de autovalores a calcular
59  //@method  version a usar del problema
60  PetscErrorCode chladni(PetscInt n, PetscInt M, char method[])
61  {
62      PetscInt sM=n*n, sr=(n-2)*4, sr2, sp, II, i, j, v, k, nconv;
63      PetscScalar t, mu = 0.225, h = 2.0/(n-3.0);
64      PetscBool bon;
65      PetscInt b[sr], g[sr];
66      Mat L, N, A, B, A0, A1, A2, A3, A4, B0, F;
67      IS phys, ghost;
68      Vec x;
69      EPS eps;
70      ST st;
71      KSP ksp;
72      PC pc;
73
74      PetscCall(MatCreate(PETSC_COMM_WORLD,&L));
75      PetscCall(MatSetSizes(L,PETSC_DECIDE,PETSC_DECIDE,sM,sM));
76      PetscCall(MatSetFromOptions(L));
77      PetscCall(MatSetUp(L));
78
79      PetscCall(MatCreate(PETSC_COMM_WORLD,&N));
80      PetscCall(MatSetSizes(N,PETSC_DECIDE,PETSC_DECIDE,sM,sM));
81      PetscCall(MatSetFromOptions(N));
82      PetscCall(MatSetUp(N));
83
84      PetscCall(MatGetOwnershipRange(L, &Istart, &Iend));
```

```
85      sp = 0;
86      sr2 = 0;
87      for (II=Istart;II<Iend;II++) {
88          bon = PETSC_FALSE;
89        i = II/n; j = II-i*n;
90
91          if(i!=0 && i!=n-1 && j!=0 && j!=n-1) {
92              sp = sp + 1;
93          }
94          else {
95              sr2 = sr2 + 1;
96          }
97
98          if(II==0 || II==n-1 || II==((n-1)*n) || II==(n*n)-1){
99              sr2 = sr2 - 1;
100         }
101
102       if (i>0) {
103             PetscCall(MatSetValue(L,II,II-n,-1.0,INSERT_VALUES));
104             if ((i>1 && i<n-1) && (j==1 || j==n-2)){
105                 PetscCall(MatSetValue(N,II,II-n,-0.5,INSERT_VALUES));
106                 bon = PETSC_TRUE;
107             }
108             else {
109                 PetscCall(MatSetValue(N,II,II-n,-1.0,INSERT_VALUES));
110             }
111         }
112       if (i<n-1) {
113             PetscCall(MatSetValue(L,II,II+n,-1.0,INSERT_VALUES));
114             if ((i>0 && i<n-2) && (j==1 || j==n-2)){
115                 PetscCall(MatSetValue(N,II,II+n,-0.5,INSERT_VALUES));
116                 bon = PETSC_TRUE;
117             }
118             else {
119                 PetscCall(MatSetValue(N,II,II+n,-1.0,INSERT_VALUES));
120             }
121         }
122       if (j>0) {
123             PetscCall(MatSetValue(L,II,II-1,-1.0,INSERT_VALUES));
124             if ((j>1 && j<n-1) && (i==1 || i==n-2)){
125                 PetscCall(MatSetValue(N,II,II-1,-0.5,INSERT_VALUES));
126                 bon = PETSC_TRUE;
127             }
128             else {
129                 PetscCall(MatSetValue(N,II,II-1,-1.0,INSERT_VALUES));
130             }
131         }
132       if (j<n-1) {
133             PetscCall(MatSetValue(L,II,II+1,-1.0,INSERT_VALUES));
134             if ((j>0 && j<n-2) && (i==1 || i==n-2)){
```

```
135                 PetscCall(MatSetValue(N,II,II+1,-0.5,INSERT_VALUES));
136                 bon = PETSC_TRUE;
137             }
138             else {
139                 PetscCall(MatSetValue(N,II,II+1,-1.0,INSERT_VALUES));
140             }
141         }
142       PetscCall(MatSetValue(L,II,II,4.0,INSERT_VALUES));
143         if(II==n+1 || II==(n*2)-2 || II==(n*(n-2))+1 || II==(n*(n-1))-2) {
144             PetscCall(MatSetValue(N,II,II,1.0,INSERT_VALUES));
145         } else if (bon) {
146             PetscCall(MatSetValue(N,II,II,2.0,INSERT_VALUES));
147         }
148         else {
149             PetscCall(MatSetValue(N,II,II,4.0,INSERT_VALUES));
150         }
151     }
152     PetscCall(MatAssemblyBegin(L,MAT_FINAL_ASSEMBLY));
153     PetscCall(MatAssemblyEnd(L,MAT_FINAL_ASSEMBLY));
154
155     PetscCall(MatAssemblyBegin(N,MAT_FINAL_ASSEMBLY));
156     PetscCall(MatAssemblyEnd(N,MAT_FINAL_ASSEMBLY));
157
158     createVec(&b, n+1, ((n-2)*n)+1, n+1, (n*2)-2, n-2);
159     createVec(&g, 1, ((n-1)*n)+1, n, (n*2)-1, n-2);
160
161     k = 0;
162     v = 0;
163     PetscInt p[sp];
164     PetscInt g2[sr2];
165     for(II=Istart;II<Iend;II++){
166         i = II/n;
167         j = II-i*n;
168
169         if(II!=0 && II!=n-1 && II!=((n-1)*n) && II!=(n*n)-1){
170             if(i!=0 && i!=n-1 && j!=0 && j!=n-1) {
171                 p[k] = II;
172                 k = k+1;
173             }
174             else {
175                 g2[v] = II;
176                 v = v+1;
177             }
178         }
179     }
180
181     PetscCall(ISCreateGeneral(PETSC_COMM_WORLD, sp, p, PETSC_COPY_VALUES, &phys));
182
183     PetscCall(ISCreateGeneral(PETSC_COMM_WORLD, sr2, g2, PETSC_COPY_VALUES, &ghost));
184
```

```
185     PetscCall(MatZeroRowsIS(L, ghost, 0.0, NULL, NULL));
186
187     t = (mu-1)/2;
188     PetscScalar mult[2][4] = {{t, -1*t, -1*t, t}, {-1*t, t, t, -1*t}};
189
190     PetscCall(MatAssemblyBegin(L,MAT_FLUSH_ASSEMBLY));
191     PetscCall(MatAssemblyEnd(L,MAT_FLUSH_ASSEMBLY));
192
193     for(i=0;i<(sr/4)-1;i++){
194         v = g[i];
195         if (v>=Istart && v<Iend){
196             PetscInt rows[2] = {v, v+1};
197             PetscInt columns[4] = {v, v+1, v+(2*n), v+(2*n)+1};
198             PetscCall(MatSetValues(L, 2, rows, 4, columns, mult, ADD_VALUES));
199         }
200     }
201
202     for(i=(sr/4);i<(sr/2)-1;i++){
203       v = g[i];
204         if (v>=Istart && v<Iend){
205             PetscInt rows[2] = {v, v+1};
206             PetscInt columns[4] = {v, v+1, v-(2*n), v-(2*n)+1};
207             PetscCall(MatSetValues(L, 2, rows, 4, columns, mult, ADD_VALUES));
208         }
209     }
210
211     PetscScalar mult2[2][4] = {{-1*t, t, t, -1*t}, {t, -1*t, -1*t, t}};
212
213     for(i=(sr/2);i<((sr*3)/4)-1;i++){
214       v = g[i];
215         if (v>=Istart && v<Iend){
216             PetscInt rows[2] = {v, v+n};
217             PetscInt columns[4] = {v+n, v, v+n+2, v+2};
218             PetscCall(MatSetValues(L, 2, rows, 4, columns, mult2, ADD_VALUES));
219         }
220     }
221
222     for(i=((sr*3)/4);i<sr-1;i++){
223       v = g[i];
224         if (v>=Istart && v<Iend){
225             PetscInt rows[2] = {v, v+n};
226             PetscInt columns[4] = {v+n, v, v+n-2, v-2};
227             PetscCall(MatSetValues(L, 2, rows, 4, columns, mult2, ADD_VALUES));
228         }
229     }
230
231     PetscCall(MatSetOption(L, MAT_IGNORE_ZERO_ENTRIES, PETSC_TRUE));
232     PetscCall(MatSetOption(N, MAT_IGNORE_ZERO_ENTRIES, PETSC_TRUE));
233
234     PetscCall(MatAssemblyBegin(L, MAT_FINAL_ASSEMBLY));
```

```
235     PetscCall(MatAssemblyEnd(L, MAT_FINAL_ASSEMBLY));
236
237     PetscCall(MatMatMult(N, L, MAT_INITIAL_MATRIX, PETSC_DEFAULT, &A));
238
239     PetscCall(MatDestroy(&N));
240     PetscCall(MatDestroy(&L));
241
242     PetscCall(MatSetOption(A, MAT_IGNORE_ZERO_ENTRIES, PETSC_TRUE));
243     MatSetOption(A, MAT_NEW_NONZERO_LOCATION_ERR, PETSC_FALSE);
244     PetscCall(MatZeroRowsIS(A, ghost, 0, NULL, NULL));
245
246     PetscCall(MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY));
247     PetscCall(MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY));
248
249     PetscScalar mult3[1][5] = {{2*(1+mu), -1, -1*mu, -1*mu, -1}};
250
251     for(i=0;i<sr/4;i++){
252       v = g[i];
253         if (v>=Istart && v<Iend){
254             PetscInt rows[1] = {v};
255             PetscInt columns[5] = {v+n, v, v+n-1, v+n+1, v+(2*n)};
256             PetscCall(MatSetValues(A, 1, rows, 5, columns, mult3, ADD_VALUES));
257         }
258     }
259
260     for(i=sr/4;i<sr/2;i++){
261         v = g[i];
262         if (v>=Istart && v<Iend){
263             PetscInt rows[1] = {v};
264             PetscInt columns[5] = {v-n, v, v-n-1, v-n+1, v-(2*n)};
265             PetscCall(MatSetValues(A, 1, rows, 5, columns, mult3, ADD_VALUES));
266         }
267     }
268
269     for(i=sr/2;i<(sr*3)/4;i++){
270         v = g[i];
271         if (v>=Istart && v<Iend){
272             PetscInt rows[1] = {v};
273             PetscInt columns[5] = {v+1, v, v+1+n, v+1-n, v+2};
274             PetscCall(MatSetValues(A, 1, rows, 5, columns, mult3, ADD_VALUES));
275         }
276     }
277
278     for(i=(sr*3)/4;i<sr;i++){
279         v = g[i];
280         if (v>=Istart && v<Iend){
281             PetscInt rows[1] = {v};
282             PetscInt columns[5] = {v-1, v, v-1+n, v-1-n, v-2};
283             PetscCall(MatSetValues(A, 1, rows, 5, columns, mult3, ADD_VALUES));
284         }
```

```
285     }
286
287     PetscCall(MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY));
288     PetscCall(MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY));
289
290     PetscCall(MatCreate(PETSC_COMM_WORLD,&B));
291     PetscCall(MatSetSizes(B,PETSC_DECIDE,PETSC_DECIDE,sM,sM));
292     PetscCall(MatSetFromOptions(B));
293     PetscCall(MatSetUp(B));
294
295     PetscCall(MatAssemblyBegin(B, MAT_FINAL_ASSEMBLY));
296     PetscCall(MatAssemblyEnd(B, MAT_FINAL_ASSEMBLY));
297
298     MatZeroEntries(B);
299     MatShift(B, 1);
300     for(i=0;i<sr;i++){
301         v = b[i];
302         if(v>=Istart && v<Iend) {
303             MatSetValue(B, v, v, 0.5, INSERT_VALUES);
304         }
305     }
306
307     v = n+1;
308     if(v>=Istart && v<Iend) {
309         MatSetValue(B, v, v, 0.25, INSERT_VALUES);
310     }
311
312     v = (n*2)-2;
313     if(v>=Istart && v<Iend) {
314         MatSetValue(B, v, v, 0.25, INSERT_VALUES);
315     }
316
317     v = (n*(n-2))+1;
318     if(v>=Istart && v<Iend) {
319         MatSetValue(B, v, v, 0.25, INSERT_VALUES);
320     }
321
322     v = (n*(n-1))-2;
323     if(v>=Istart && v<Iend) {
324         MatSetValue(B, v, v, 0.25, INSERT_VALUES);
325     }
326
327     PetscCall(MatAssemblyBegin(B,MAT_FINAL_ASSEMBLY));
328     PetscCall(MatAssemblyEnd(B,MAT_FINAL_ASSEMBLY));
329
330     PetscCall(MatCreateDense(PETSC_COMM_WORLD, sr2 , sr2 ,PETSC_DECIDE,PETSC_DECIDE,NULL,&A4
            ));
331
332     MatType type;
333     MatGetType(A4,&type);
```

```
334     PetscPrintf(MPI_COMM_WORLD, "%s\n", type);
335
336     PetscCall(MatCreateSubMatrix(B, phys, phys, MAT_INITIAL_MATRIX, &B0));
337     PetscCall(MatCreateSubMatrix(A, ghost, ghost, MAT_INITIAL_MATRIX, &A2));
338     PetscCall(MatCreateSubMatrix(A, ghost, phys, MAT_INITIAL_MATRIX, &A3));
339
340     PetscCall(MatDestroy(&B));
341
342     PetscCall(MatCreate(PETSC_COMM_WORLD,&F));
343     PetscCall(MatSetSizes(F,sr2,sr2,PETSC_DECIDE,PETSC_DECIDE));
344     PetscCall(MatSetUp(F));
345     PetscCall(MatAssemblyBegin(F,MAT_FINAL_ASSEMBLY));
346     PetscCall(MatAssemblyEnd(F, MAT_FINAL_ASSEMBLY));
347
348     PetscCall(MatZeroEntries(F));
349     PetscCall(MatShift(F,1));
350     PetscCall(MatConvert(F, MATDENSE, MAT_INITIAL_MATRIX, &F));
351
352     PetscCall(KSPCreate(PETSC_COMM_WORLD,&ksp));
353     PetscCall(KSPSetOperators(ksp, A2, A2));
354     PetscCall(KSPGetPC(ksp,&pc));
355     PetscCall(KSPSetType(ksp,KSPPREONLY));
356     PetscCall(PCSetType(pc, PCLU));
357     PetscCall(KSPMatSolve(ksp, F, A4));
358
359     PetscCall(MatDestroy(&A2));
360     PetscCall(MatDestroy(&F));
361
362     PetscCall(MatAssemblyBegin(A4,MAT_FINAL_ASSEMBLY));
363     PetscCall(MatAssemblyEnd(A4, MAT_FINAL_ASSEMBLY));
364
365     PetscCall(MatConvert(A4, MATAIJ, MAT_INITIAL_MATRIX, &A4));
366
367     PetscCall(MatMatMult(A4, A3, MAT_INITIAL_MATRIX, PETSC_DECIDE, &A4));
368
369     PetscCall(MatDestroy(&A3));
370
371     PetscCall(MatCreateSubMatrix(A, phys, phys, MAT_INITIAL_MATRIX, &A0));
372     PetscCall(MatCreateSubMatrix(A, phys, ghost, MAT_INITIAL_MATRIX, &A1));
373
374     PetscCall(MatMatMult(A1, A4, MAT_INITIAL_MATRIX, PETSC_DECIDE, &A1));
375
376     PetscCall(MatConvert(A1, MATAIJ, MAT_INITIAL_MATRIX, &A1));
377
378     PetscCall(MatDestroy(&A4));
379
380     PetscCall(MatAXPY(A0, -1, A1, DIFFERENT_NONZERO_PATTERN));
381
382     PetscCall(MatDestroy(&A1));
383
```

```
384     PetscCall(MatScale(A0, 1/(h*h*h*h)));
385
386     PetscCall(EPSCreate(PETSC_COMM_WORLD,&eps));
387     PetscCall(EPSSetOperators(eps,A0,B0));
388     PetscCall(EPSSetProblemType(eps,EPS_GHEP));
389     PetscCall(EPSSetDimensions(eps,M,-2,-2));
390     PetscCall(EPSSetFromOptions(eps));
391     PetscCall(EPSGetST(eps, &st));
392     PetscCall(STSetFromOptions(st));
393     PetscCall(STSetType(st, STSINVERT));
394     PetscCall(EPSSetWhichEigenpairs(eps,EPS_TARGET_REAL));
395     PetscCall(EPSSetTarget(eps, -1e-2));
396     PetscCall(STGetKSP(st, &ksp));
397     PetscCall(KSPGetPC(ksp, &pc));
398     PetscCall(PCSetType(pc, PCLU));
399
400     PetscCall(EPSSolve(eps));
401
402     PetscCall(MatDestroy(&A0));
403     PetscCall(MatDestroy(&B0));
404
405    DM da;
406     PetscDraw draw;
407     PetscViewer viewer;
408
409     PetscViewerDrawOpen(PETSC_COMM_WORLD,0,"Prueba",-1,1,PETSC_DECIDE,PETSC_DECIDE,&
            viewer);
410     PetscViewerPushFormat(viewer, PETSC_VIEWER_DRAW_CONTOUR);
411     PetscViewerDrawGetDraw(viewer,0,&draw);
412     DMDACreate2d(PETSC_COMM_WORLD,DM_BOUNDARY_NONE,DM_BOUNDARY_NONE,DMDA_STENCIL_BOX,(n
            -2),(n-2),PETSC_DECIDE,PETSC_DECIDE,1,1,NULL,NULL,&da);
413     DMSetFromOptions(da);
414     DMSetUp(da);
415     DMCreateGlobalVector(da,&x);
416
417     PetscCall(EPSGetConverged(eps,&nconv));
418     if (nconv>0) {
419         for (i=0;i<nconv;i++) {
420
421             char number[5];
422             sprintf(number, "%d", i);
423             char name[20] = "ondas";
424             strcat(name, number);
425             strcat(name, ".png");
426             PetscDrawSetSave(draw,name);
427
428
429             PetscCall(EPSGetEigenvector(eps,i,x,NULL));
430             VecView(x,viewer);
431             PetscDrawSave(draw);
```

```
432             }
433         }
434         PetscCall(VecDestroy(&x));
435 
436         PetscCall(MatDestroy(&A));
437         PetscCall(EPSDestroy(&eps));
438         PetscCall(ISDestroy(&ghost));
439         PetscCall(ISDestroy(&phys));
440 }
```

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**			X	
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**		X		
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**			X	
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

ETS Enginyeria Informàtica
Camí de Vera, s/n, 46022, València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

El tema de este trabajo surgió de la curiosidad y el deseo por el conocimiento. Principalmente se pretendía comprobar el funcionamiento de una herramienta y aplicarla a una situación en concreto (en este caso, el calculo y representación de las figuras de Chladni), con el fin de comprobar la calidad de los resultados que proporcionaba. A pesar de esto, lo que se ha logrado con este trabajo tiene relevancia y en las manos adecuadas puede aportar grandes beneficios en muchas áreas y lograr objetivos de gran importancia.

Por ejemplo, el contenido de este trabajo está relacionado en gran medida con el objetivo de la **innovación**. Esto se debe a que, para resolver un determinado problema, no solo se han empleado técnicas líderes de paralelismo y computación científica, sino que también se han llevado a cabo empleando las librerías de software desarrolladas en la UPV PETSc y SLEPc, las cuales son poco conocidas, pero logran hacer el máximo uso de las técnicas mencionadas, aportando mejores resultados que otras soluciones que emplean herramientas más populares.

Esto por supuesto, se puede aplicar a todo tipo de simulaciones de diferentes campos, como la automovilística o la aeronáutica. En general, las herramientas que se han empleado en este trabajo pueden utilizarse en el todo tipo de ámbitos, ya que podemos simular por ejemplo, sistemas de modelado en 3D mediante extrusión de aluminio, pruebas de soldadura por láser, fabricación de piezas mediante fundición a presión, o incluso, simulación de partes anatómicas del cuerpo humano, para la detección de enfermedades o el diseño de prótesis resistentes y eficientes de usar. Por esto, la influencia que los contenidos de este trabajo pueden tener en la **industria** puede ser muy alta si es aplicada correctamente.

Por otro lado, uno de los casos más importantes en los que estas técnicas pueden resultar de ayuda, es en el diseño y la construcción de **infraestructuras**. El trabajo realizado en este proyecto se puede aplicar a simulaciones que, por ejemplo, puedan representar fielmente y de forma eficiente la resistencia de puentes a diferentes entornos y estimulaciones naturales, como tsunamis o fuertes rachas de viento, y con esto asegurar que las infraestructuras que se construyen y que utilizamos en nuestro día a día son seguras y no tiene ningún riesgo, evitando posibles

ETS Enginyeria Informàtica
Camí de Vera, s/n, 46022, València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es

accidentes que puedan ocurrir en el futuro. Es por esto, por lo que este trabajo también tiene relación con el desarrollo de **ciudades y comunidades sostenibles**.

Además, otro objetivo de gran relevancia que pueden favorecerse de las técnicas mencionadas y de el software de PETSc y SLEPc es el de **agua limpia y saneamiento**, ya que realizar simulaciones relacionadas con la dinámica de fluidos puede ayudar considerablemente al desarrollo de sistemas de drenaje que distribuyan agua limpia eficientemente por nuestras ciudades.

Aún así, este trabajo tiene principalmente relación con los aspectos infraestructurales del diseño de poblaciones, pero no tiene tanta relación con los aspectos sociales, relacionales y culturales de una sociedad. Es por esto, por lo que otros objetivos de gran importancia, como el fin de la pobreza, la igualdad de género, o la reducción de desigualdades, no están tan conectados con el estudio realizado en este trabajo. Sin embargo, al mismo tiempo, este trabajo no afecta de forma negativa a ninguno de estos factores.