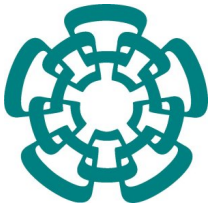

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Paralelización heterogénea del algoritmo de Cuppen, caso de estudio: problema regular de Sturm-Liouville

Tesis que presenta

Alberto Estrella Cruz

para obtener el grado de

Maestro en Ciencias en Computación

Director de Tesis

Dr. Sergio Víctor Chapa Vergara

México, Distrito Federal — Diciembre, 2014

Resumen

El algoritmo de Cuppen (o algoritmo divide y vencerás) es un algoritmo numéricamente estable y eficiente que calcula todos los valores y vectores propios de una matriz tridiagonal simétrica. Los problemas regulares de Sturm-Liouville son ecuaciones diferenciales lineales de segundo orden que tienen una forma especial y cuyos valores en la frontera deben respetar ciertas condiciones. El cálculo numérico de los valores propios de este tipo de problemas es de gran importancia, debido a las distintas áreas de aplicación que tienen, tales como: vibración de cuerdas, propagación de ondas, termodinámica, mecánica y química cuántica. Entre los métodos numéricos para resolver este problema se encuentran, el método de disparo y el método de diagonalización de matrices.

Estos métodos tienen un alto costo computacional, por lo que, en esta tesis se desarrollaron implementaciones paralelas de ellos para reducir su tiempo de ejecución, y se utilizaron en casos específicos del problema regular de valores propios de Sturm-Liouville. En este trabajo se presenta, a nuestro conocimiento, la primera implementación paralela heterogénea en un cluster con múltiples GPUs y CPUs del algoritmo divide y vencerás para valores propios. Una de las principales desventajas de implementar dicho algoritmo en GPUs es que el cómputo esta limitado por la cantidad de memoria del GPU. Para superar este problema nosotros utilizamos múltiples nodos con CPUs y GPUs para resolver subproblemas que encajan en la memoria de un GPU en paralelo y reunimos sus resultados para obtener la solución completa. Experimentos preliminares muestran resultados prometedores. Nuestro enfoque tiene mejor desempeño que el de librerías actuales. Ademas, las soluciones obtenidas muestran un alto grado de exactitud con respecto a la ortogonalidad de los vectores propios y a la calidad de los eigenpares.

Abstract

Cuppen's algorithm (or divide-and-conquer algorithm) is a numerically stable and efficient algorithm that computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix. Regular Sturm-Liouville problems are second order differential equations with special form those boundary values accomplish certain conditions. The considerable importance of the numerical calculation of their eigenvalues laid on the many eigenvalue problems that arise in scientific and engineering applications such as: vibrating strings, propagation of waves, thermodynamics, quantum mechanics and quantum chemistry. Among the variety of methods to solve these problems are shooting methods and matrix diagonalization.

These methods have a high computational cost, for this reason in this thesis parallel implementations of them were developed in order to reduce running times. Also, they were applied to specific cases of the regular Sturm-Liouville eigenvalue problem. In this work we present, to the best of our knowledge, the first heterogeneous parallel implementation on a cluster with multiple GPUs and CPUs of the divide-and-conquer eigenvalue algorithm. A major drawback of implementing such algorithm on GPUs is that computation is limited by the amount of GPU memory. We overcome the issue by using multiple nodes with CPUs and GPUs to solve subproblems, which fit on device memory in parallel, and merging those subproblems to get the whole result. Preliminary experiments show promising results. Our approach has better performance than state-of-the-art libraries. Furthermore, it exhibits a meaningful degree of accuracy with respect to orthogonality of eigenvectors and quality of egeinpairs.

Agradecimientos

A mi madre y a mi hermana por el apoyo incondicional que me han brindado y con el que siempre podré contar

A mi director el Dr. Sergio Víctor Chapa Vergara por aceptarme como su tesista

A mis sinodales el Dr. Oliver Steffen Schütze y el Dr. Amilcar Meneses Viveros por sus valiosos comentarios

Al Departamento de Computación por las todas las experiencias y el aprendizaje que me dejaron

Al Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional (CINVESTAV-IPN) por darme la oportunidad de ser parte de uno de sus posgrados de maestría

Al Consejo Nacional de Ciencia y Tecnología (CONACYT) por el apoyo económico que me brindo durante esta etapa

A los compañeros y amigos que he conocido a lo largo de mi vida, en especial a Hugo y Jorge por creer en mi

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

1.1. Antecedentes y motivación

El problema de valores propios regular de Sturm-Liouville tiene un gran número de aplicaciones, debido a que todos aquellos problemas que puedan ser modelados con una ecuación diferencial que cumpla con las características del problema de valores propios regular de Sturm-Liouville, que se mencionan en el siguiente capítulo, se pueden solucionar mediante las técnicas que se estudian en esta tesis.

Prueba de esto fue la conferencia que tuvo lugar en Suiza en 2003 en conmemoración del 200 aniversario del nacimiento de Charles Francois Sturm, el fundador de la teoría de Sturm-Liouville, a la que asistieron mas de 60 participantes de 16 países. Como resultado de este coloquio se publico una recopilación de artículos presentados en este evento [1], en los que se mencionan las distintas áreas de aplicación que tiene el problema regular de Sturm-Liouville entre las que se encuentran: vibración de cuerdas, propagación de ondas, resonancias, termodinámica, mecánica cuántica, química cuántica.

Entre los métodos numéricos utilizados para resolver el problema de valores propios regular de Sturm-Liouville se encuentran, el método de disparo y el método de diagonalización de matrices simétricas, estos métodos tienen un costo computacional alto, por lo que, emplear programación paralela para su implementación, puede resultar en una mejora significativa en el tiempo de ejecución.

La programación paralela hasta hace algunos años, se había realizado comúnmente con una gran cantidad de nodos con procesadores multinúcleo, recientemente se han aprovechado el poder de cómputo y el bajo consumo energético de las unidades de procesamiento gráfico (GPUs) para realizar cómputo de propósito general y no solo procesamiento de gráficos. Sin embargo, ambos enfoques tienen ventajas y desventajas, por lo que, la tendencia actual es utilizar las fortalezas individuales de cada uno, en un enfoque híbrido.

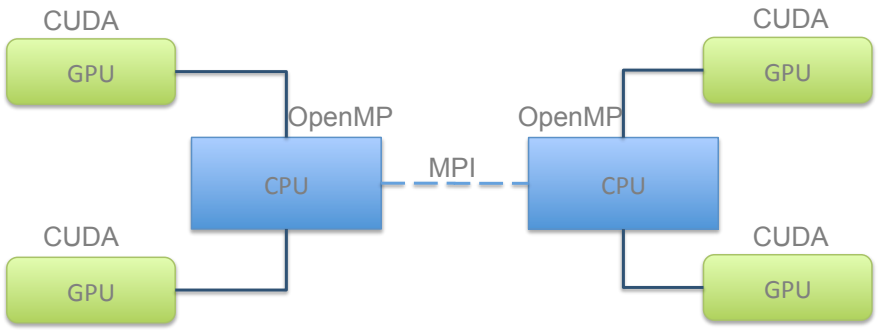

Figura 1.1: Arquitectura paralela heterogénea

Un punto que es importante mencionar es que la cantidad de memoria máxima con la que cuenta una GPU hoy en día es limitada, aproximadamente 8GB, por lo que el contar con una implementación que utilice varios nodos puede ayudar a disminuir esta restricción.

En esta tesis se estudia cual es el beneficio obtenido al utilizar este enfoque híbrido y cual método es más conveniente, al resolver el problema de valores propios regular de Sturm-Liouville, en términos de tiempo y exactitud.

1.2. Estado del arte

Entre las soluciones existentes para resolver el problema de valores propios destacan, las bibliotecas LAPACK (Linear Algebra PACKage) [2] y MAGMA (Matrix Algebra on GPU and Multicore Architectures) [3].

LAPACK es una biblioteca secuencial que es la base para otras bibliotecas de álgebra lineal, de hecho prácticamente todas las aplicaciones que requieren operaciones de álgebra lineal la utilizan. Existen version optimizadas de LAPACK, la que mejores tiempos reporta es la MKL de Intel, esta implementación puede ejecutarse de forma secuencial o utilizar los diferentes núcleos de un procesador.

MAGMA es la sucesora de LAPACK, el propósito de esta biblioteca es utilizar los procesadores multi-núcleo y las unidades de procesamiento gráfico GPUs. Sin embargo, no utiliza múltiples nodos, esto limita el tamaño de las matrices que pueden resolverse en especial en aquellas funciones que utilizan una GPU, ya que estas tienen una memoria mas limitada que la que puede acceder un CPU. Por ejemplo una targeta de un GB de memoria puede resolver un problema de $n = 9318$, pero al aumentar una unidad este número se presenta un error por memoria insuficiente.

1.3. Planteamiento del problema

El problema de valores propios regular de Sturm-Liouville tiene muchas aplicaciones, entre las cuales existen problemas para los que no se conoce un solución analítica, por lo tanto, se requieren métodos numéricos.

Entre los métodos más utilizados están la diagonalización de matrices simétricas y el método de disparo. A continuación, se da una breve descripción de los métodos numéricos que se estudian en esta tesis.

Método de disparo

El método de disparo aplicado al problema de Sturm-Liouville, consiste en, tomar un rango de posibles valores propios, resolver la ecuación diferencial correspondiente para cada uno de ellos y seleccionar aquellos valores para los cuales se cumplen las condiciones a la frontera del problema.

Para obtener el conjunto de valores propios y las aproximaciones de la funciones solución y de su derivada se siguen los siguientes pasos:

1. Se parte el intervalo en el cual se buscan las soluciones en N intervalos, usualmente igualmente espaciados.
2. Se utiliza un algoritmo para resolver problemas de valores iniciales como Runge-Kutta para generar las posibles funciones solución y sus derivadas.
3. Se filtran las soluciones que cumplan las condiciones en la frontera y se agregan al conjunto de soluciones finales.

Una mejor explicación de este método puede consultarse en [4].

Diagonalización de matrices tridiagonales simétricas

En este método matricial la ecuación diferencial se ve como un operador diferencial (el operador de Sturm-Liouville), y se representa con diferencias finitas en una forma matricial, $Lx = \lambda x$. Con lo que se obtiene un problema de valores propios simétrico matricial que se resuelve mediante diagonalización de matrices. Es decir, encontrar una matriz diagonal D y una matriz ortogonal P tal que $L = PDP^{-1}$.

Para obtener los vectores y valores propios de una matriz tridiagonal T simétrica real se siguen los siguientes tres pasos [5]:

1. Los subeigenproblemas $T_1 = Q_1\Lambda_1 Q_1^T$ y $T_2 = Q_2\Lambda_2 Q_2^T$ son resueltos para dos matrices que son, salvo una modificación de rango uno, los bloques diagonales de la matriz original. Este paso puede hacerse utilizando cualquier método.
2. Usando los eigenpares de T_1, T_2 todos los valores propios de T son calculados mediante la solución de la ecuación secular.

$$0 = 1 + \rho \sum_{i=1}^{n} \frac{v_i^2}{d_i - \lambda}$$

3. Una vez que los valores propios de T son conocidos, los vectores propios de T se calculan en dos fases, un escalamiento diagonal apropiado seguido de una multiplicación de matriz por matriz.

Una explicación más completa de este método puede encontrarse en [6]. Los principales problemas que se presentan al implementar este algoritmo son el cálculo de la ecuación secular [7] y el conservar la ortogonalidad de los vectores propios [8].

Las soluciones numéricas tienen un error de aproximación en la practica y son costosas computacionalmente, sobre todo, la diagonalización de matrices, debido a esto se hace necesario el uso de programación paralela para disminuir el tiempo de cómputo.

En esta tesis se estudia cuál es el beneficio que se obtiene al implementar métodos para solucionar el problema de valores propios regular de Sturm-Liouvile en arquitecturas paralelas híbridas (GPUs + CPUs), en términos de exactitud y tiempo.

En particular los métodos de disparo y diagonalización de matrices, ya que se quiere averiguar cuál de ellos es el mas conveniente para resolver este problema y que tanto se pueden adecuar a un enfoque híbrido.

No existe una implementación del algoritmo de Cuppen [9, 6] para diagonalizar matrices que utilice múltiples nodos con CPUs de múltiples núcleos y múltiples GPUs [10], por lo que no se sabe cuál es el beneficio que se obtendrá al utilizar varios nodos, ya que este beneficio podría disminuir principalmente por el costo de comunicación entre los nodos.

En esta tesis se pretende contestar las interrogantes antes mencionadas. Con el uso de las arquitecturas híbridas se busca lograr acelerar la ejecución de las implementaciones de los métodos para solucionar el problema de valores propios regular de Sturm-Liouville y aprovechar al máximo los recursos disponibles. El contar con tales implementaciones puede ayudar a resolver una gran cantidad de problemas en los que se utilizan problemas regulares de Sturm-Liouville. Ademas, la implementación del algoritmo de Cuppen en múltiples nodos puede disminuir las limitaciones de memoria que tienen las unidades de procesamiento gráfico, a la vez que, su bajo consumo de energía, reduce los costos de ejecución.

1.4. Objetivos

Esta tesis tiene como objetivo general implementar el algoritmo para diagonalizar matrices simétricas tridiagonales conocido como algoritmo divide y vencerás, de forma paralela heterogénea (GPUs + CPUs), en múltiples nodos y utilizarlo para solucionar el problema de valores propios regular de Sturm-Liouville, con el fin de, averiguar que tan adecuado es aplicar este método en este problema, en términos de exactitud y tiempo.

Para paralelizar el algoritmo se utilizan unidades de procesamiento gráfico empleando la plataforma CUDA de Nvidia, procesadores de múltiples núcleos de Intel usando OpenMP, y múltiples nodos mediante paso de mensajes con MPI.

Para cumplir con el objetivo general se definieron los siguientes objetivos particulares:

- Investigación sobre las mejoras al algoritmo divide y vencerás para diagonalizar matrices simétricas, en particular la solución de la ecuación secular.

- Paralelizar el algoritmo de disparo para el problema de valores propios regular de Sturm-Liouville, de forma híbrida (GPU + CPU) en un nodo.
- Aumentar la escalabilidad y la independencia de la cantidad de memoria de la GPU, de la implementación del algoritmo divide y vencerás.
- Reducir el tiempo de cómputo de la diagonalización de matrices simétricas tridiagonales.
- Mostrar la ventaja de aprovechar el poder de procesamiento de los GPUs y CPUs en conjunto, y de utilizar varios nodos a la vez, en este problema en particular.
- Averiguar que método es más conveniente para solucionar el problema de valores propios regular de Sturm-Liouville, el método de disparo paralelo o el método de Cuppen paralelo, en términos de exactitud y tiempo.
- Resolver un problema de Sturm-Liouville del que no se conozca la solución exacta.
- Comparar los métodos implementados con los existentes, en términos de tiempo y exactitud.

1.5. Contribuciones y resultados esperados

Las contribuciones y resultados que se esperan obtener de esta tesis son:

- Tener un estudio de los métodos para solucionar el problema de valores propios regular de Sturm-Liouville en CUDA para múltiples tarjetas.
- Implementación híbrida escalable en clusters de GPUs del método divide y vencerás para diagonalizar matrices tridiagonales simétricas.
- Cálculo del error práctico del problema de valores propios simétrico tridiagonal con métodos matriciales.

1.6. Organización de la tesis

El resto del documento esta organizado en seis capítulos, a continuación se presenta una breve descripción de cada uno de ellos.

- **Capítulo 2. Marco teórico:** En este capítulo se explican los conceptos y algoritmos que se utilizaran durante el documento, principalmente sobre el problema regular de Sturm-Liouville, solución de ecuaciones diferenciales de segundo grado, y diagonalización de matrices. También, se muestra los diferentes paradigmas de programación paralela, la definición de algoritmo híbrido y heterogéneo, y se describen las tecnologías que se utilizan para implementar los algoritmos.
- **Capítulo 3. Algoritmo divide y vencerás:** En este capítulo se describe el algoritmo divide y vencerás para diagonalizar matrices tridiagonales simétricas, su justificación teórica y los elementos que se deben tomar en cuenta para su correcta implementación.

- **Capítulo 4. Diseño e implementación:** En este capítulo se presentan las estrategias de paralelización, el diseño y la implementación del algoritmo de disparo y de diagonalización de matrices simétricas tridiagonales en su version secuencial y paralela.
- **Capítulo 5. Pruebas, resultados y discusión:** En este capítulo se describen los casos de prueba y la arquitectura de computo sobre la que se trabaja, se presentan los resultados obtenidos, se comparan con otras implementaciones, y se discute que relevancia tienen los resultados obtenidos.
- **Capítulo 6. Conclusiones y trabajo a futuro:** En este capítulo se presentan las principales conclusiones de esta tesis y se discuten algunos aspectos que nos gustaría mejorar en una investigación futura.

Capítulo 2

Marco teórico

En este capítulo inicia en la sección 2.1 con una revisión formal del problema regular de Sturm-Liouville, se describen dos problemas sobre mecánica cuántica, ya que los casos de prueba se basan en estos problemas, enseguida en las secciones 2.2 y 2.3 (en las págs. 10 y 12) se explican los métodos de disparo y de diferencias finitas, después en la sección 2.4 (en la pág. 12) se define el problema algebraico de valores propios y se presentan algunos métodos que son utilizados comúnmente en su resolución. Por último, en la sección 2.5 (en la pág. 22) se describen brevemente las tecnologías que utilizamos para implementar los algoritmos.

2.1. Problema regular de Sturm-Liouville

Un problema de valores a la frontera que consiste de la ecuación diferencial lineal de segundo orden

$$\left\{-\frac{d}{dx}\left[p(x)\frac{d}{dx}\right]+q(x)\right\}y(x)=\lambda y(x) \qquad x\in[a,b]$$

y las condiciones en la frontera

$$\alpha_1 y(a)+\alpha_2 y'(a)=0, \qquad \alpha_1^2+\alpha_2^2>0,$$

$$\beta_1 y(b)+\beta_2 y'(b)=0, \qquad \beta_1^2+\beta_2^2>0$$

donde $p(x)>0, q(x)$ son funciones reales continuas en el intervalo (a,b)

Es llamado problema de Sturm-Liouville, se dice que un problema de Sturm-Liouville es regular si el intervalo $[a,b]$ es finito y la función $q(x)$ es integrable en este intervalo [11], resolver este problema significa encontrar los valores de λ, llamados valores propios, y las correspondientes soluciones no triviales $y_\lambda(x)$, conocidas como funciones propias. El conjunto de todos valores propios de un problema regular es llamado espectro [12].

Uno de los resultados importantes en la teoría de Sturm-Liouville, es el teorema de oscilación, cuya demostración se puede encontrar en [11]. Las soluciones numéricas encontradas en esta tesis respetan este teorema, como podrá verse mas adelante.

Teorema 2.1 (Teorema de oscilación) *Existe una secuencia creciente indefinida de valores propios $\lambda_0, \lambda_1, \ldots, \lambda_n, \ldots$ del problema regular de Sturm-Liouville, y la función propia correspondiente al valor propio λ_m, tiene precisamente m ceros en el intervalo (a, b).*

Ejemplos de problemas de Sturm-Liouville son las ecuaciones de Schödinger y Dirac que se utilizan para describir sistemas de mecánica cuántica relativista y no relativista. La localización de una partícula en estos sistemas esta determinada por la función $\psi(x)$ en el sentido de que $|\psi(x)|/||\psi||^2$ en una medida de su densidad de probabilidad en $\mathbb{R}$ [13].

2.1.1. Ecuación de Schrödinger

Erwin Schrödinger publicó esta ecuación de onda en 1926 en [14], la cual se expresa de la siguiente forma

$$i\hbar\frac{\partial}{\partial t}\psi(\mathbf{x},t) = -\frac{\hbar}{2m}\nabla^2\psi(\mathbf{x},t) + V(\mathbf{x},t)\psi(\mathbf{x},t)$$

donde $\mathbf{x} \in \mathbb{R}^n$ y n es la dimensión del espacio de configuración o espacio posición [15]. Si consideramos una dimensión en el espacio de configuración, la ecuación anterior se escribe como

$$i\hbar\frac{d}{dt}\psi(x,t) = -\frac{\hbar}{2m}\psi(x,t) + V(x,t)\psi(x,t))$$

consideremos el caso en que el potencial V sólo depende de x, la ecuación se resuelve con respecto a x tal que la solución $\psi(x,t)$ se puede escribir como

$$\psi(x,t) = e^{-iEt/\hbar}\psi(x,t_0) = e^{-iEt/\hbar}\phi(x)$$

que es conocida como la ecuación para el caso estacionario, podemos escribirla en notación de una matriz de dimensión 2×2, y si consideramos a la constante $\hbar$ como menos uno y a la masa m de la partícula como uno, obtenemos la siguiente representación

$$\frac{dZ(x)}{dx} = \begin{pmatrix} 0 & E - V(x) \\ 1 & 0 \end{pmatrix} Z(x)$$

que es equivalente al par de ecuaciones diferenciales acopladas

$$\frac{dy(x)}{dx} = z(x),$$

$$\frac{dz(x)}{dx} = [E - V(x)]\, y(x)$$

El problema del pozo de potencial infinito es un problema clásico en mecánica cuántica, se asume que el potencial es infinito fuera de la región $(-a, a)$ y cero en esta región. Este potencial indica que las partículas dentro de la región están atrapadas y no pueden salir, así que la probabilidad de encontrarlas en esta región es 1. Entonces, el potencial se describe como

$$V(x) = \begin{cases} 0 & x \in (-a, a) \\ \infty & \text{en otro caso} \end{cases}$$

Los valores propios de este problema están dados por

$$\lambda_c = -2 + 2\cos\left(\frac{c\pi}{n+1}\right), \qquad c = 1, \cdots, n$$

2.1.2. Ecuación de Dirac

La ecuación de Dirac es una ecuación de onda para la mecánica cuántica relativista. Paul Dirac publicó esta ecuación de onda en 1928 [16]. A diferencia de la ecuación de Schrödinger, la ecuación de Dirac es un sistema de ecuaciones diferenciales acopladas. La ecuación de Dirac se escribe a partir del hamiltoniano

$$E\Psi = H\Psi,$$

$$H = i\sigma_2\frac{d}{dx} + m_0c^2\sigma_3 + V(x)\mathbf{I},$$

$$\Psi = \begin{pmatrix}\psi_1\\ \psi_2\end{pmatrix}$$

donde $E \in \mathbb{R}$ y representa la energía; H es el operador hamiltoniano (u operador de energía); Ψ es el vector de las funciones de onda solución del sistema, y se le llama espinor; $\mathbf{I}$ es la matriz unitaria; y $\sigma_1, \sigma_2, \sigma_3$ son las matrices de Dirac

$$\sigma_1 = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}, \quad \sigma_2 = \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}, \quad \sigma_3 = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

En una dimensión, la ecuación de Dirac para el caso estacionario donde el potencial es independiente del tiempo, puede escribirse en forma matricial como

$$\frac{d\Psi(x)}{dx} = \begin{pmatrix}0 & m_0 - [V(x) - E]\\ m_0 + [V(x) - E] & 0\end{pmatrix}\Psi(x)$$

donde m_0 es la masa en reposo y $V(x)$ el potencial independiente del tiempo [15]. La ecuación anterior es equivalente al par de ecuaciones diferenciales acopladas

$$\frac{dy(x)}{dx} = \{m_0 + [E - V(x)]\}\, z(x),$$

$$\frac{dz(x)}{dx} = \{m_0 - [E - V(x)]\}\, y(x)$$

Dentro de los enfoques con los que se cuenta para resolver el problema regular de Sturm-Liouville están, el enfoque de problema de valores a la frontera y el de representación como operador matricial.

2.2. Método de disparo

Este método de valores en la frontera consiste en transformar el problema en uno de valores iniciales. En este método se seleccionan los valores de las variables dependientes en uno de los limites de la frontera, usualmente los valores iniciales, enseguida se integra la ecuación diferencial ordinaria, con métodos de valores iniciales (como Runge-Kutta), llegando hasta el otro limite tratando de satisfacer las condiciones en la frontera en el punto final. En general se encuentran discrepancias con el valor deseado en el punto final, por lo que, se ajustan los parámetros iniciales hasta que ambas condiciones, las finales y las iniciales, se satisfacen. Esto es análogo a intentar varios disparos de integración con condiciones iniciales tratando de atinar en las condiciones finales, por lo que, se le conoce como método de disparo.

Algoritmo 2.1 Disparo para el problema de Sturm-Liouville

Entrada: Puntos finales a, b; condiciones en la frontera α, β; rango de λ, $[l, r]$; numero de subintervalos N

Salida: Conjunto Λ de valores propios y S de aproximaciones w_i de $y(x_i)$ y w'_i de $y'(x_i)$ para cada $i = 0, 1, ..., N$

1: **para** los puntos λ que surgen de partir $[l, r]$ en N intervalos **hacer**
2: Usar un algoritmo de valores iniciales como Runge-Kutta para generar w_i, w'_i
3: **si** w_i, w'_i cumplen con las condiciones en la frontera α, β **entonces**
4: Agregar w_i, w'_i a S y λ a Λ
5: **devolver** (Λ, S)

Una mejor explicación de este método puede consultarse en [4].

2.2.1. Método de Runge-Kutta

Para aproximar la solución al problema de valores iniciales

$$f' = F(x, f), \quad a \leq x \leq b, \quad F(a) = \alpha_0$$

en $(N+1)$ números igualmente espaciados en el intervalo $[a, b]$; usamos el siguiente algoritmo

Algoritmo 2.2 Runge-Kutta de cuarto orden

Entrada: Puntos finales a, b; numero de subintervalos N; condición inicial α_0

Salida: Aproximaciones f_i de $y(x_i)$ para $i = 0, 1, ..., N$

1: $h \leftarrow (b - a)/N$
2: $x_0 \leftarrow a$
3: $f_0 \leftarrow \alpha_0$
4: **para** $i = 0$ **a** N **hacer**
5: $k_1 \leftarrow F(x_i, f_i)$
6: $k_2 \leftarrow F(x_i + \frac{1}{2}h, f_i + \frac{1}{2}k_1 h)$
7: $k_3 \leftarrow F(x_i + \frac{1}{2}h, f_i + \frac{1}{2}k_2 h)$
8: $k_4 \leftarrow F(x_i + h, f_i + k_3 h)$
9: $f_{i+1} \leftarrow f_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k4)h$
10: $x_{i+1} \leftarrow a + ih$
11: **devolver** f

Para aproximar la solución al sistema de problemas de valores iniciales

$$y' = z = F(x, y, z), \quad a \leq x \leq b, \quad F(a) = \alpha_0,$$

$$z' = y'' = G(x, y, z), \quad a \leq x \leq b, \quad G(a) = \alpha_1$$

en $(N+1)$ números igualmente espaciados en el intervalo $[a, b]$; usamos el siguiente algoritmo

Algoritmo 2.3 Runge-Kutta para sistemas de ecuaciones

Entrada: Puntos finales a, b; numero de subintervalos N; condiciones iniciales α_0, α_1
Salida: Aproximaciones y_i de $y(x_i)$ y z_i de $y'(x_i)$ para $i = 0, 1, ..., N$

1: $h \leftarrow (b-a)/N$
2: $x_0 \leftarrow a$
3: $y_0 \leftarrow \alpha_0$
4: $z_0 \leftarrow \alpha_1$
5: **para** $i = 0$ **a** N **hacer**
6: $\quad k_1 \leftarrow F(x_i, y_i, z_i)$
7: $\quad l_1 \leftarrow G(x_i, y_i, z_i)$
8: $\quad k_2 \leftarrow F(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h, z_i + \frac{1}{2}l_1 h)$
9: $\quad l_2 \leftarrow G(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h, z_i + \frac{1}{2}l_1 h)$
10: $\quad k_3 \leftarrow F(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h, z_i + \frac{1}{2}l_2 h)$
11: $\quad l_3 \leftarrow G(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h, z_i + \frac{1}{2}l_2 h)$
12: $\quad k_4 \leftarrow F(x_i + h, y_i + k_3 h, z_i + l_3 h)$
13: $\quad l_4 \leftarrow G(x_i + h, y_i + k_3 h, z_i + l_3 h)$
14: $\quad y_{i+1} \leftarrow y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k4)h$
15: $\quad z_{i+1} \leftarrow z_i + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l4)h$
16: $\quad x_{i+1} \leftarrow a + ih$
17: **devolver** (y, z)

Por ejemplo, para utilizar el método de Runge-Kutta para el problema de Schrödinger F y G están definidos por

$$F(x_i, y_i, z_i) = z_i,$$

$$G(x_i, y_i, z_i) = [E - V(x_i)]\, y_i$$

y para el problema de Dirac por

$$F(x_i, y_i, z_i) = \{m_0 + [E - V(x_i)]\}\, z_i,$$

$$G(x_i, y_i, z_i) = \{m_0 - [E - V(x_i)]\}\, y_i$$

2.3. Método de diferencias finitas

Los métodos de diferencias finitas se basan en sustituir las derivadas en las ecuaciones diferenciales por sus análogas discretas, las diferencias finitas. Por ejemplo, para resolver la ecuación de Schrödinger, sabemos que las diferencias finitas centradas para la derivada de segundo orden son

$$\frac{d^2y}{dx^2} \approx \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2}, \qquad h = \Delta x$$

aplicando diferencias finitas a la ecuación independiente del tiempo de Schrödinger

$$\frac{d^2y}{dx^2} = [E - V(x)]\, y$$

obtenemos la siguiente aproximación

$$\frac{y_{i-1} - 2y_i + y_{i+1}}{h^2} \approx [E - V(x)]\, y_i$$

reacomodando los términos obtenemos

$$\frac{1}{h^2}\left\{y_{i-1} + \left[h^2V(x_i) - 2\right] y_i + y_{i+1}\right\} \approx Ey_i$$

y finalmente con la expresión anterior podemos construir la siguiente matriz

$$\frac{1}{h^2}\begin{pmatrix} h^2V(x_0)-2 & 1 & & & & 0 \\ 1 & h^2V(x_1)-2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & h^2V(x_{n-2})-2 & 1 & \\ 0 & & & 1 & h^2V(x_{n-1})-2 \end{pmatrix}\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{pmatrix} = E\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{pmatrix}$$

que representa un problema de valores propios.

2.4. Problema algebraico de valores propios

Los valores y vectores propios pertenecen a los temas de mayor utilidad del álgebra lineal. Se usan en varias aéreas de las matemáticas, física, mecánica, ingeniería eléctrica y nuclear, hidrodinámica y aerodinámica. De hecho es raro encontrar un una área de la ciencia aplicada donde nunca se hayan usado.

La transformación $L : \mathbb{R}^n \to \mathbb{R}^n$ definida por $L(x) = Ax$ para x en $\mathbb{R}^n$, es una transformación lineal. Una cuestión de gran importancia en una gran variedad de problemas de aplicación es la determinación de vectores x, si los hay tales que x y Ax son paralelos. Tal cuestión aparece en todas las aplicaciones relacionadas con las vibraciones [17]: en elasticidad, mecánica nuclear, ingeniería química, biología, ecuaciones diferenciales, etc.

Existe un gran número de problemas matemáticos que no corresponden necesariamente al campo del álgebra lineal cuya representación puede llevarse a un problema algebraico de

valores propios, esto ocurre debido a que las soluciones de dichos problemas se llevan a cabo mediante la utilización de operadores lineales. Un operador lineal puede representarse por una matriz de transformación, esta matriz puede ser utilizada para reducir el problema inicial a un problema de valores propios.

Por ejemplo, el problema general de valores propios puede referirse a ecuaciones diferenciales con coeficientes constantes. El conjunto general de ecuaciones homogéneas de primer orden con n incógnitas $y_1, y_2, \ldots, y_n$ puede ser escrito en la forma

$$B\frac{d}{dt}(y) = Cy$$

suponiendo que B es no singular esta ecuación puede ser escrita en la forma

$$\begin{aligned} \frac{d}{dt}(y) &= Cy, \\ A &= B^{-1}C \end{aligned}$$

asumiendo que la ecuación tiene una solución de la forma

$$y = xe^{\lambda t}$$

donde x es un vector independiente de t. Se debe tener

$$\lambda x e^{\lambda t} = Axe^{\lambda t}$$

y por lo tanto

$$\lambda x = Ax$$

se tiene una solución si λ es un valor propio de A y x es el vector propio correspondiente.

2.4.1. Definición

El problema fundamental de los valores propios es la determinación de aquellos valores de λ para los que el conjunto de n ecuaciones lineales homogéneas con n incógnitas

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

tiene una solución no trivial [18]. La ecuación (2.1) puede ser escrita en la forma

$$(A - \lambda I)x = 0 \tag{2.2}$$

La teoría general de ecuaciones algebraicas lineales simultáneas nos muestra que existe una solución no trivial si, y solo si, la matriz $A - \lambda x$ es singular, esto es

$$det(A - \lambda x) = 0$$

El determinante en el lado izquierdo de la ecuación anterior puede ser desarrollado para obtener la ecuación polinomio explícita

$$\alpha_0 + \alpha_1\lambda + \cdots + \alpha_{n-1}\lambda^{n-1} + (-1)^n\lambda^n = 0$$

La ecuación anterior es llamada *ecuación característica* de la ecuación de la matriz A y el polinomio en el lado izquierdo de la ecuación es llamado el polinomio característico. Como el coeficiente de λ^n no es cero y se asume que se trabaja en el campo de los números complejos, la ecuación siempre tiene n raíces. En general, las raíces pueden ser complejas, incluso si la matriz A es real, y puede haber raíces de cualquier multiplicidad hasta n. Las n raíces son llamadas los *valores propios* de la matriz A.

Correspondiendo con cualquier valor propio λ, el conjunto de ecuaciones (2.2) tiene al menos una solución no trivial x. Cada solución es llamada un *vector propio* correspondiente a este valor propio. Si la matriz $A - \lambda x$ es de rango menor que $n-1$, entonces habrá más de un vector independiente que satisface la ecuación (2.2). Es evidente que si x es una solución, entonces kx también es una solución para cualquier valor de k, de forma que incluso cuando $(A - \lambda x)$ es de rango $n-1$, el vector propio correspondiente a λ es libre de usar un multiplicador constante.

Es conveniente escoger este multiplicador de forma que el vector propio tenga algunas propiedades numéricas que sean deseables, dichos vectores son llamados vectores *normalizados*.

2.4.2. Transformaciones similares

Las transformaciones de la forma $H^{-1}AH$ de la matriz A, donde H es no singular son de fundamental importancia tanto para el punto de vista teórico como para el práctico, y son conocidas como transformaciones similares; mientras que A y $H^{-1}AH$ se dicen ser *similares* o semejantes. Obviamente $H^{-1}AH$ es también una transformación similar de A. Cuando los valores propios de A son distintos, entonces hay una transformación similar que reduce A a la forma diagonal, es decir, A se puede *diagonalizar*, y la matriz de transformación tiene sus columnas iguales a los vectores propios de A.

Si se tiene

$$H^{-1}AH = D$$

entonces

$$AH = HD$$

La última ecuación implica que d_i son los valores propios de A en el mismo orden en que la i-esima columna de H es un vector propio correspondiente a d_i.

Los valores propios de una matriz son invariantes sobre una transformación similar.

Porque si

$$Ax = \lambda x$$

entonces

$$H^{-1}Ax = \lambda H^{-1}x$$

se obtiene

$$H^{-1}A(HH^{-1})x = \lambda H^{-1}x$$
$$(H^{-1}AH)H^{-1}x = \lambda H^{-1}x$$

Los valores propios son por lo tanto preservados y los valores propios son multiplicados por H^{-1}.

Una forma más común de demostrar que los valores propios de las matrices similares se preservan es mostrando que tienen el mismo polinomio característico.

Si A es semejante a B

$$B = H^{-1}AH$$

entonces

$$\begin{aligned}\det &= \det(H^{-1}AH - \lambda I)\\ &= \det(H^{-1}(A - \lambda I)H)\\ &= \det(H^{-1})\det(A - \lambda I)\det(H)\\ &= \det(A - \lambda I)\end{aligned}$$

ya que el polinomio característico de A y B es igual, tienen los mismos valores propios.

Muchos de los métodos numéricos para encontrar los valores y vectores propios de una matriz consisten esencialmente en la determinación de una transformación similar que reduce una matriz A de la forma general a una matriz B de una forma especial, para la que el problema de valores y vectores propios puede ser resuelto de una forma más simple.

La semejanza es una propiedad transitiva, porque si

$$B = H_1^{-1}AH_1 \qquad C = H_2^{-1}AH_2$$

entonces

$$C = H_2^{-1}H_1^{-1}AH_1H_2 = (H_1H_2)^{-1}A(H_1H_2)$$

En general la reducción de una matriz general a una forma diagonal se realizara mediante una secuencia de transformaciones similares.

2.4.3. Diagonalización de matrices

Como mencionamos anteriormente, para encontrar los valores y vectores propios de una matriz, con frecuencia, se utilizan transformaciones similares para obtener una matriz similar a la original para la que es mas fácil encontrar los valores y vectores propios. Por lo que es lógico preguntarnos cual es la matriz mas sencilla que podemos obtener en términos de calcular sus valores y vectores propios, la respuesta es, la matriz diagonal.

Una matriz se dice ser diagonal si todos los elementos fuera de su diagonal principal son iguales a cero, y su valores propios son precisamente los valores en su diagonal principal, por lo que se pueden obtener por simple inspección.

Por lo mencionado en los dos párrafos anteriores al proceso de encontrar un transformación similar de la forma

$$A = PDP^{-1}$$

se le conoce como diagonalización.

Existen diversos tipos de algoritmos para calcular los valores y vectores propios de una matriz, entre los que destacan aquellos que se basan en la diagonalización de una matriz a partir de sucesivas transformaciones similares. Para llegar a una matriz diagonal, es común hacer una transformación para llegar a una matriz tridiagonal y después realizar otra para finalmente obtener la matriz diagonal

Este escrito se centra en este enfoque, especialmente cuando se aplica a matrices simétricas. Un punto importante de mencionar, es que incluso para resolver problemas con valores complejos y problemas generales de valores propios, por lo general, se termina resolviendo una matriz tridiagonal simétrica. Que es el caso especifico que abordaremos en esta tesis.

A continuación se mencionan de manera breve algunos algoritmos que son de gran utilidad en la determinación de los valores y vectores propios de una matriz.

2.4.4. Algoritmo QR

Si A es una matriz cuadrada con columnas linealmente independientes, entonces A puede factorizarse en la forma

$$A = QR$$

En la que Q es una matriz con columnas ortonormales y R es una matriz triangular superior.

Para encontrar estas matrices es común utilizar el proceso de ortonormalización de Gram-Schmidt, no obstante, se pueden emplear otros algoritmos de factorización para este propósito. La descomposición QR se puede utilizar para aproximar los valores propios de una matriz cuadrada, al algoritmo resultante se le llama algoritmo QR, este algoritmo determina todos los valores propios de una matriz.

La idea principal de este algoritmo es construir una secuencia de matrices similares, $A_1, A_2, \ldots, A_k$,donde A_k converge a una matriz triangular superior semejante a la matriz A conforme k aumenta. Por consiguiente los elementos diagonales de dicha matriz son los valores propios de A, ya que los valores propios de una matriz triangular son los elementos de su diagonal.

Se comienza calculando la factorización QR de $A_0 = Q_0R_0$, y a continuación se forma la matriz $A_1 = R_0Q_0$, se puede observar que A_0 y A_1 son semejantes, porque

$$A_1 = R_0Q_0 = (Q_0^{-1}A_0)Q_0$$

Por lo tanto, tienen los mismos valores propios. Se continua con la factorización QR de $A_1 = Q_1R_1$, y se forma la matriz $A_2 = R_1Q_1$ que tiene los mismos valores propios de A. Como se había mencionado, se itera hasta obtener una matriz triangular superior.

Se puede ver que todas las matrices en la secuencia son similares, debido a que

$$A_k = R_{k-1}Q_{k-1} = (Q_{k-1}^{-1}A_{k-1})Q_{k-1}$$

Entonces, todas las matrices en la secuencia tienen los mismos valores propios, incluyendo la matriz triangular resultante del proceso iterativo.

Una de las características más notables del algoritmo QR es que al ser aplicado a una matriz simétrica produce una matriz diagonal, esto significa que el algoritmo QR nos permite diagonalizar una matriz simétrica.

El costo computacional del algoritmo QR es igual al número de iteraciones necesarias para obtener la matriz triangular por la complejidad del algoritmo utilizado para calcular la factorización QR.

Los pasos del algoritmo pueden observarse de forma resumida en el siguiente algoritmo

Algoritmo 2.4 Método iterativo QR

Entrada: Matriz real simétrica $A \in \mathbb{R}^{n\times n}$

Salida: Matriz $P \in \mathbb{R}^{n\times n}$ cuyas columnas son los vectores propios de A; Matriz $\Lambda \in \mathbb{R}^{n\times n}$ diagonal cuyos elementos son los valores propios de A

1: $\Lambda \leftarrow A$
2: $P \leftarrow \mathbb{I}$
3: **mientras** $\Lambda \neq$ diagonal **hacer**
4: $\quad QR \leftarrow \Lambda$
5: $\quad P \leftarrow PQ$
6: $\quad \Lambda \leftarrow RQ$
7: **devolver** (Λ, P)

2.4.5. Método de Householder

El método de Householder reduce una matriz simétrica arbitraria a una matriz tridiagonal simétrica que es semejante a la matriz inicial. El algoritmo se basa en utilizar transformaciones

de Householder, también conocidas como reflexiones elementales, que sirven para eliminar de manera selectiva bloques de elementos en una forma extremadamente estable con respecto al redondeo. El algoritmo de Householder utiliza $n-2$ transformaciones para llevar una matriz simétrica a una matriz tridiagonal. Para cada una de estas transformaciones se calcula una matriz P_k de la forma

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & p & p & p & \cdots & p \\ 0 & p & p & p & \cdots & p \\ 0 & p & p & p & \cdots & p \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & p & p & p & \cdots & p \end{pmatrix}_{k=1}, \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & p & p & \cdots & p \\ 0 & 0 & p & p & \cdots & p \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & p & p & \cdots & p \end{pmatrix}_{k=2}, \cdots, \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & p & p \\ 0 & 0 & 0 & \cdots & p & p \end{pmatrix}_{k=n-2}$$

más formalmente P_k esta determinada por

$$P_k = I - \beta v v^T, \qquad \beta = \frac{2}{v^T v}$$

para calcular β y v se puede utilizar el siguiente algoritmo [19]

Algoritmo 2.5 Obtención del vector de Householder

Entrada: Vector $x \in \mathbb{R}^n$

Salida: Vector $v \in \mathbb{R}^n$ con $v(1) = 1$ y $\beta \in \mathbb{R}$ tal que $P = I_n - \beta v v^T$ es ortogonal y $Px = ||x||_2 e_1$

$\sigma \leftarrow x(2:n)^T x(2:n), v \leftarrow \begin{pmatrix} 1 \\ x(2:n) \end{pmatrix}$
si $\sigma = 0$ **y** $x(1) \geq 0$ **entonces**
    $\beta \leftarrow 0$
si no
    si $\sigma = 0$ **y** $x(1) \leq 0$ **entonces**
        $\beta \leftarrow -2$
    si no
        $\mu \leftarrow \sqrt{x(1)^2 + \sigma}$
        si $x(1) \leq$ **entonces**
            $v(1) \leftarrow x(1) - \mu$
        si no
            $v(1) \leftarrow -\sigma/(x(1) + \mu)$
        $\beta \leftarrow 2v(1)^2/(\sigma + v(1)^2)$
        $v \leftarrow v/v(1)$
devolver (v, β)

para cada iteración $k = 1, 2, \cdots, n-2$ se calcula la nueva A_{k+1} de la siguiente manera

$$A_{k+1} = P_k A_k P_k$$

para realizar estas iteraciones se pude utilizar el siguiente algoritmo [19]

Algoritmo 2.6 Tridiagonalización de Householder

Entrada: Matriz simétrica $A \in \mathbb{R}^{n\times n}$

Salida: Matriz modificada $A \in \mathbb{R}^{n\times n}$ con $T = Q^T AQ$, donde T es tridiagonal y $Q = H_1, H_2, \cdots, H_{n-2}$ es el producto de transformaciones de Householder

1: **para** $k = 1$ **a** $n-2$ **hacer**
2: Utilizar el algoritmo 2.5 con $A(k+1:n,k)$ como entrada y v, β como salida
3: $p \leftarrow \beta A(k+1:n, k+1:n)v$
4: $w \leftarrow p - (\beta p^T v/2)v$
5: $A(k+1,k) \leftarrow \|A(k+1:n,k)\|_2$; $A(k,k+1) \leftarrow A(k+1,k)$
6: $A(k+1:n, k+1:n) \leftarrow A(k+1:n, k+1:n) - vw^T - wv^T$
7: **devolver** A

Después de realizar todas las iteraciones, se obtiene en A_{n-1} una matriz tridiagonal y simétrica. Ya que se tiene la matriz tridiagonal se suele utilizar un algoritmo como QR para calcular los valores propios de la matriz. Una de las propiedades de mayor utilidad de las proyecciones de Householder es que al ser ortogonales y simétricas evitan realizar cálculos computacionales adicionales para obtener sus correspondientes matrices inversas y transpuestas.

2.4.6. Método de Givens

El algoritmo de Givens es un caso especial del algoritmo de Jacobi. Al igual que el método de Householder, este método anula ciertos elementos de una matriz, pero mediante la aplicación de rotaciones sucesivas de Givens. Estas transformaciones mantienen los mismos valores propios que la matriz original.

Se puede utilizar una serie de rotaciones de Givens sobre una matriz cuadrada para anular sus entradas bajo su diagonal principal, transformando la matriz original en una matriz triangular superior, a este procedimiento se le llama reducción de Givens.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{41} & a_{42} & a_{43} & \cdots & a_{nn} \end{pmatrix} \rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ & a_{22} & a_{23} & \cdots & a_{2n} \\ & & a_{33} & \cdots & a_{3n} \\ & & & \ddots & \vdots \\ 0 & & & & a_{nn} \end{pmatrix}$$

Para convertir una matriz densa en una matriz triangular superior, se deben anular todos los elementos que se encuentran debajo de la diagonal principal de izquierda a derecha y de abajo hacia arriba. Cada una de las rotaciones que anulan los distintos elementos de la matriz original solo altera la fila donde se encuentra el elemento a eliminar y la fila anterior a esta.

$$\begin{pmatrix} u_1 & u_2 & u_3 & u_4 \\ v_1 & v_2 & v_3 & v_4 \end{pmatrix}$$

Para logar anular un elemento en particular, el primer paso es calcular la matriz de rotación de Givens esta se obtiene con las siguientes formulas.

$$g = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

donde

$$c = \frac{u_1}{\sqrt{u_1^2 + v_1^2}} \qquad \text{y} \qquad s = \frac{v_1}{\sqrt{u_1^2 + v_1^2}} \tag{2.3}$$

el elemento que se desea anular es v_1 y el elemento que se encuentra en la misma columna pero en la fila inmediatamente superior a v_1, es decir arriba de él, es u_1.

Una vez que se ha calculado esta matriz se multiplica por las filas que alteran esta rotación, para obtener los nuevos elementos de la matriz rotada los cuales se denotan por R de la siguiente manera.

$$R = \begin{pmatrix} \frac{u_1}{\sqrt{u_1^2 + v_1^2}} & \frac{v_1}{\sqrt{u_1^2 + v_1^2}} \\ -\frac{v_1}{\sqrt{u_1^2 + v_1^2}} & \frac{u_1}{\sqrt{u_1^2 + v_1^2}} \end{pmatrix} \begin{pmatrix} u_1 & u_2 & u_3 & u_4 \\ v_1 & v_2 & v_3 & v_4 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \end{pmatrix}$$

nótese que al efectuar la multiplicación el elemento $r_{21} = 0$, que es el objetivo que se planteaba desde un principio, esto ocurre debido a que

$$r_{21} = -\frac{v_1 u_1}{\sqrt{u_1^2 + v_1^2}} + \frac{v_1 u_1}{\sqrt{u_1^2 + v_1^2}} = 0$$

la matriz original después de aplicar una rotación de Givens para anular, por ejemplo, el elemento a_{41} es la siguiente:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{pmatrix}$$

por la tanto, la reducción de Givens permite determinar la matriz R del algoritmo QR.

Tal vez la distinción más significativa entre el método de Givens y otros métodos, es que al utilizarlo en matrices simétricas, es altamente paralelizable.

Una matriz simétrica requiere $n(n-1)/2$ rotaciones de Givens para convertirse en una matriz triangular superior, más aún, si la matriz es también tridiagonal solo se requieren $n-1$ rotaciones. Tal vez esta es la diferencia de mayor importancia entre el método de Jacobi y el de Givens, ya que el método de Jacobi no específica un número finito de pasos.

En la práctica, hay mejores formas de calcular c y s que la ecuación (2.3) (en la pág 20), por ejemplo, con el siguiente algoritmo [19].

Algoritmo 2.7 Obtención de una rotación de Givens

Entrada: Escalares $a, b \in \mathbb{R}$

Salida: Escalares $c = \cos(\theta), s = \sin(\theta) \in \mathbb{R}$ tales que

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$$

1: **si** $b = 0$ **entonces**
2: $c \leftarrow 1; s \leftarrow 0$
3: **si no**
4: **si** $|b| > |a|$ **entonces**
5: $\tau \leftarrow -a/b;\ s \leftarrow 1/\sqrt{1+\tau^2};\ c \leftarrow s\tau$
6: **si no**
7: $\tau \leftarrow -b/a;\ c \leftarrow 1/\sqrt{1+\tau^2};\ s \leftarrow c\tau$
8: **devolver** (c, s)

Este algoritmo solo requiere cinco flops y una sola raíz cuadrada, tampoco necesitan funciones trigonométricas inversas.

Es importante aprovechar la estructura de la rotación de Givens cuando la utilizamos en una multiplicación de matrices. Por ejemplo, si se multiplica una rotación de Givens por una matriz es conveniente utilizar el siguiente algoritmo [19].

Algoritmo 2.8 Aplicación de una rotación de Givens (pre-multiplicación)

Entrada: Matriz $A \in \mathbb{R}^{n \times n}$, los escalares $c = \cos(\theta), s = \sin(\theta) \in \mathbb{R}$

Salida: Matriz $A \in \mathbb{R}^{n \times n}$, tal que la actualización $A = G(i, k, \theta)^T A$ afecta solo dos filas

$$A([i, k], :) = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T A([i, k], :)$$

1: **para** $j = 1$ **a** n **hacer**
2: $\tau_1 \leftarrow A(i, j)$
3: $\tau_2 \leftarrow A(k, j)$
4: $A(i, j) \leftarrow c\tau_1 - s\tau_2$
5: $A(k, j) \leftarrow s\tau_1 - c\tau_2$
6: **devolver** A

Y si se multiplica una matriz por una rotación de Givens es conveniente utilizar el siguiente algoritmo [19].

Algoritmo 2.9 Aplicación de una rotación de Givens (pos-multiplicación)

Entrada: Matriz $A \in \mathbb{R}^{n \times n}$, los escalares $c = \cos(\theta), s = \sin(\theta) \in \mathbb{R}$
Salida: Matriz $A \in \mathbb{R}^{n \times n}$, tal que la actualización $A = AG(i, k, \theta)$ afecta solo dos columnas

$$A(:, [i, k]) = A(:, [i, k]) \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

1: **para** $j = 1$ **a** n **hacer**
2: $\quad \tau_1 \leftarrow A(j, i)$
3: $\quad \tau_2 \leftarrow A(j, k)$
4: $\quad A(j, i) \leftarrow c\tau_1 - s\tau_2$
5: $\quad A(j, k) \leftarrow s\tau_1 - c\tau_2$
6: **devolver** A

Estos algoritmos tienen complejidades lineales en vez de cubicas, como pasaría en el caso de una multiplicación típica de matrices densas.

2.5. Programación paralela

Los términos computación paralela, procesamiento paralelo, y programación paralela son usados de forma ambigua algunas veces, o no son claramente definidos y diferenciados [20]. Computación paralela es un termino que engloba todas las tecnologías utilizadas en la ejecución simultanea de múltiples tareas en múltiples procesadores. El procesamiento paralelo, o paralelismo, es alcanzado dividiendo una sola tarea en múltiples tareas más pequeñas e independientes. Las tareas pueden ejecutarse simultáneamente cuando uno o más procesadores están disponibles. Si solo un procesador esta disponible, las tareas se ejecutan secuencialmente. En un solo procesador moderno de alta velocidad, las tareas puden parecer ser ejecutadas al mismo tiempo, pero en realidad no pueden ser ejecutadas simultáneamente en un solo procesador.

Programación paralela es la metodología de software usada para implementar el procesamiento paralelo. El programa debe incluir instrucciones para informar al sistema de ejecución que partes de la aplicación pueden ser ejecutadas simultáneamente. El programa entonces se dice estar paralelizado. La programación paralela es una técnica de optimización de desempeño que intenta reducir el "tiempo de reloj"de ejecución de una aplicación permitiendo al programa manejar varias actividades simultáneamente. La programación paralela puede ser implementada utilizando muchas diferentes interfaces de software, o modelos de programación paralela.

El modelo de programación paralela usado en cualquier aplicación depende de la arquitectura de hardware del sistema sobre el cual se espera ejecutar. Específicamente, el desarollador debe distinguir entre un sistema de memoria compartida y un sistema de memoria distribuida. En una arquitectura de memoria compartida, la aplicación puede acceder de manera transparente a cualquier localidad de memoria. Un procesador multinúcleo es un ejemplo de de un sistema de memoria compartida. En un entorno de memoria distribuida, la aplicación solamente puede acceder transparentemente a la memoria de el nodo sobre el que se esta

ejecutando. El acceso a la memoria en otro nodo tiene que ser explicitamente acordado con la aplicación. Clusters y grids son ejemplos de sistemas de memoria distribuida.

2.5.1. Híbrida

Un modelo híbrido combina más de un modelo de programación paralela [20]. Con la llegada de los sistemas multinúcleo, un creciente número de clusters y grids son sistemas paralelos con dos capas. Dentro de un solo nodo, la rápida comunicación a través de la memoria compartida puede ser explotada, y un protocolo de red pude ser utilizado para comunicarse a través de los nodos. Los programas puede tomar ventaja de ambos la memoria compartida y la memoria distribuida.

El modelo de MPI pude ser usado para ejecutar aplicaciones paralelas en clusters de sistemas multinúcleo. Las aplicaciones MPI pueden ejecutarse a través de los nodos también como dentro de cada nodo, por lo que ambas capas de paralelización, compartida y distribuida pueden ser utilizadas con MPI. En ciertas situaciones, sin embargo, la agregación de la paralelización de grano fino ofrecida por un modelo de programación de memoria compartida tal como Pthreads o OpenMP es más eficiente. Típicamente, la ejecución paralela sobre los nodos es alcanzada por medio de MPI. Dentro de un nodo, Pthreads o OpenMP es usado. Cuando dos modelos de programación son usados en una aplicación, la aplicación se dice ser paralelizada con un modelo de programación híbrido.

Actualmente, un ejemplo común de modelo híbrido es la combinación de el modelo paso de mensajes (MPI) con el modelo de hilos (OpenMP). Los hilos realizan funciones computacionalmente intensivas usando datos locales en un nodo. La comunicación entre procesos en diferentes nodos ocurre sobre la red utilizando MPI. Otro ejemplo similar de un modelo híbrido con una creciente popularidad es el uso de MPI con programación en GPU [21]. Las GPUs desarrollan funciones computacionales intensas usando datos locales sobre un nodo. La comunicación entre procesos en diferentes nodos ocurre sobre la red utilizando MPI.

2.5.2. Heterogénea

El computo heterogéneo se refiere a sistemas que usan mas de una clase de procesador [22]. Estos son sistemas multinúclo que ganan desempeño no solo agregando núcleos, si no que también incorporan capacidades especiales de procesamiento para manejar tareas especiales (típicamente CPUs y GPUs) para brindar lo mejor de ambos mundos: el procesamiento en GPU puede realizar computaciones matemáticamente intensivas sobre conjuntos muy grandes de datos, mientras que los CPUs pueden ejecutar el sistema operativo y desempeñar tareas seriales tradicionales. La programación paralela heterogénea se refiere a la programación de este tipo de sistemas para realizar operaciones que requieren grandes cantidades de cómputo.

2.5.3. Tecnologías para implementar algoritmos paralelos

A continuación se describen brevemente las tecnologías que utilizamos para implementar los algoritmos en esta tesis.

CUDA

Es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema [23]. Parte de CUDA es un una extension al lenguaje C que permite al código de GPU ser escrito en C regular. El código es dirigido al procesador host (el CPU) o al procesador device (el GPU). El procesador host genera tareas multihilo (o kernels como son conocidos en CUDA) en el GPU. El GPU tiene su propio planificador que entonces asignará los kernels a cualquiera hardware de GPU que este presente [24].

Un programa consiste de una o mas fases que son ejecutadas en el host o en el device. Las fases que exhiben poco o ningún paralelismo son implementadas en el código de host. Las fases que exhiben una gran cantidad de paralelismo son implementadas en el código del device. Las funciones kernels (o simplemente kernels) típicamente generan un gran número de hilos para explotar el paralelismo [25]. Mas información sobre la programación en CUDA puede encontrarse en los libros [26, 25, 24, 27]

OpenMP

Es una especificación para un conjunto de directivas de compilación, rutinas de biblioteca, y variables de entorno que pueden ser usadas para especificar paralelismo de alto nivel en programas Fortran y C/C++ [28]. OpenMP ofrece un modelo de más alto nivel que el de los hilos POSIX y también provee funcionalidad adicional. En mucho casos, una implementación es construida sobre un modelo nativo de hilos como POSIX threads.

Las directivas de compilador juegan un papel importante en OpenMP, mediante la inserción de directivas en el código fuente, el desarrollador especifica que partes del programa pueden ser ejecutadas en paralelo. El compilador transforma estas partes especificadas de el programa dentro de la infraestructura apropiada, tal como una llamada a una función de una biblioteca multitarea. OpenMP tiene cuatro ventajas principales sobre otros modelos de programación:

- Potabilidad - Un programa que usa OpenMP es portable a otro compilador o entorno OpenMP
- Facilidad de uso - El desarrollador no tiene que crear y manejar hilos al nivel de los hilos de POSIX, por ejemplo. La gestión de hilos es manejada por el compilador y la librería multitarea que utiliza.
- La aplicación puede ser paralelizada paso por paso - El desarrollador especifica las secciones que pueden ser ejecutadas en paralelo, y pueden por lo tanto paralelizar incrementalmete la aplicación como sea necesario.
- La version secuencial de el programa es preservada - Si el programa no es compilado con la opción de OpenMP, las directivas en el código son ignoradas. Este comportamiento efectivamente deshabilita la ejecución para que el programa se ejecute secuencialmente otra vez.

Una buena referencia para el aprendizaje de OpenMP es [29].

MPI

Message-Passing Interface es una especificación de una interfaz de una biblioteca de paso de mensajes [30]. MPI principalmente se enfoca en el modelo de programación paralela de paso de mensajes: los datos son enviados de un espacio de direcciones de un proceso a otra de otro proceso a través de operaciones cooperativas en cada proceso. El modelo de MPI es comúnmente usado para paralelizar aplicaciones para un cluster de computadoras, o una grid. Como OpenMP, esta interfaz es una capa de software adicional por encima de la funcionalidad básica del sistema operativo. MPI es construido por encima de una interfaz de red, tal como sockets, con un protocolo tal como TCP/IP. MPI provee un basto conjunto de rutinas de comunicación, y es ampliamente disponible.

Un programa MPI es un programa secuencial C, C++, o Fortran que se ejecuta sobre un subconjunto de procesadores, o todos los procesadores en un cluster. El programador implementa la distribución de tareas y la comunicación entre ellas, y decide que trabajo es asignado a los diferentes hilos. Para lograr este fin, el programa necesita ser aumentado con llamadas a funciones de la biblioteca de MPI, por ejemplo, enviar y recibir información de otros hilos (o nodos). MPI es un modelo de programación muy explicito. Aunque alguna funcionalidad conveniente es proveída, tal como una operación global de broadcast, el desarrollador tiene que diseñar específicamente la aplicación paralela para este modelo de programación. Muchos detalles de bajo nivel también tienen que ser manejados explicitamente.

La ventaja de MPI es que una aplicación puede ejecutarse sobre cualquier tipo de cluster que tiene el software para soportar el modelo de programación de MPI. Aunque originalmente los programas de MPI se ejecutaron en clusters de un solo procesador, estaciones de trabajo o PCs, ejecutar una aplicación MPI sobre uno o más computadoras de memoria compartida es ahora común. Más información sobre MPI puede consultarse en [31].

Capítulo 3

Algoritmo divide y vencerás

En este capítulo se describe cada una de las fases del algoritmo divide y vencerás para diagonalizar matrices tridiagonales simétricas; en las secciones 3.2 y 3.3 (en las págs. 30 y 33) se presta especial atención a la solución de la ecuación secular y a la ortogonalidad de los vectores propios, ya que son estos puntos en los que se presentan las mayores dificultades al implementar el algoritmo; también, en la sección 3.5 (en la pág. 34), se señalan algunos pasos que son esenciales para una implementación eficiente como es la deflación, la cual tiene dos propósitos: reducir la complejidad del problema y evitar malas condiciones para resolver la ecuación secular; finalmente, en la sección 3.6 (en la pág. 36), se definen las matrices de permutación que se utilizan para ordenar los valores propios de los subproblemas y para reducir el tamaño del problema.

3.1. Idea general

Para utilizar esta estrategia se debe tener como matriz inicial una matriz tridiagonal simétrica de la forma

$$T = \begin{pmatrix} a_1 & b_1 & & & 0 \\ b_1 & a_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-2} & a_{n-1} & b_{n-1} \\ 0 & & & b_{n-1} & a_n \end{pmatrix}$$

que se puede separar de la siguiente manera [9]

$$T = \left(\begin{array}{ccc|ccc} \ddots & \ddots & & & & \\ \ddots & a_{m-1} & b_{m-1} & & & \\ & b_{m-1} & a_m \mp b_m & & & \\ \hline & & & a_{m+1} \mp b_m & b_{m+1} & \\ & & & b_{m+1} & a_{m+2} & \ddots \\ & & & & \ddots & \ddots \end{array}\right) + \left(\begin{array}{cc|ccc} & & & & \\ & & & & \\ & & & & \\ \hline & & b_m & \pm b_m & \\ & & \pm b_m & b_m & \\ & & & & \end{array}\right),$$

$$m \approx \frac{n}{2}$$

con lo que se tienen dos matrices tridiagonales simétricas

$$T_1 = \begin{pmatrix} a_1 & b_1 & & & 0 \\ b_1 & a_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{m-2} & a_{m-1} & b_{m-1} \\ 0 & & & b_{m-1} & a_m \mp b_m \end{pmatrix}$$

y

$$T_2 = \begin{pmatrix} a_{m+1} \mp b_m & b_{m+1} & & & 0 \\ b_{m+1} & a_{m+2} & b_{m+2} & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-2} & a_{n-1} & b_{n-1} \\ 0 & & & b_{n-1} & a_n \end{pmatrix}$$

por lo tanto, la matriz original está dada por

$$T = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \rho \mathbf{u}\mathbf{u}^T$$

donde

$$\mathbf{u} = \begin{matrix} m \to \\ m+1 \to \end{matrix} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \qquad \text{y} \qquad \rho = \pm b_m$$

de esta forma, se tienen dos problemas separados de diagonalización

$$T_1 = Q_1 \Lambda_1 Q_1^T \qquad \text{y} \qquad T_2 = Q_2 \Lambda_2 Q_2^T$$

con

$$\Lambda_1 = \begin{pmatrix} \lambda_1 & & & 0 \\ & \lambda_2 & & \\ & & \ddots & \\ 0 & & & \lambda_m \end{pmatrix} \qquad \text{y} \qquad \Lambda_2 = \begin{pmatrix} \lambda_{m+1} & & & 0 \\ & \lambda_{m+2} & & \\ & & \ddots & \\ 0 & & & \lambda_n \end{pmatrix}$$

y

$$D = \begin{pmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{pmatrix}$$

La matriz original se reescribe como

$$T = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} \left\{ \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} + \rho \mathbf{v}\mathbf{v}^T \right\} \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix}^T$$

con

$$\mathbf{v} = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix}^T \mathbf{u} = \begin{pmatrix} \pm\text{ultima columna de } Q_1^T \\ \text{primera columna de } Q_2^T \end{pmatrix}$$

La matriz M tiene los mismos valores propios que la matriz original T (es similar)

$$M \equiv D + \rho \mathbf{v}\mathbf{v}^T$$

Cuyos valores propios están dados por la ecuación secular

$$0 = 1 + \rho \sum_{i=1}^{n} \frac{v_i^2}{d_i - \lambda}$$

Los vectores propios de M son

$$y_i = (D - \lambda_i I)^{-1} \mathbf{v} \tag{3.1}$$

Los vectores propios de la matriz original están dados por

$$q_i = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} y_i$$

El seguimiento de estos pasos lleva al siguiente algoritmo recursivo [32].

Algoritmo 3.1 Diagonalización de matrices tridiagonales simétricas

Entrada: Una matriz simétrica tridiagonal $T \in \mathbb{R}^{n \times n}$

Salida: La descomposición espectral de $T = Q\Lambda Q^T$, donde la diagonal Λ es la matriz de valores propios y Q es la matriz de vectores propios y es ortogonal

1: **si** T es 1×1 **entonces**
2: **devolver** $(\Lambda = T, Q = 1)$
3: **si no**
4: Separar la matriz en la forma $T = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + \rho \mathbf{u}\mathbf{u}^T$
5: Usar este algoritmo con T_1 como entrada y Q_1, Λ_1 como salida
6: Usar este algoritmo con T_2 como entrada y Q_2, Λ_2 como salida
7: Formar $D + \rho \mathbf{v}\mathbf{v}^T$ a partir de Λ_1, Λ_2, Q_1, Q_2
8: Calcular los valores propios Λ de $D + \rho \mathbf{v}\mathbf{v}^T$ con la ecuación secular $0 = 1 + \rho \sum_{i=1}^{n} \frac{v_i^2}{d_i - \lambda}$
9: Calcular los vectores propios Q' de $D + \rho \mathbf{v}\mathbf{v}^T$ con ${q'}_i = (D - \lambda_i I)^{-1} \mathbf{v}$
10: Formar $Q = \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} \cdot Q'$ que son los vectores propios de T
11: **devolver** (Λ, Q)

3.2. Ecuación secular

El cálculo de la ecuación no es trivial. En [7] se propone una forma estable y eficiente para resolver esta ecuación. El autor propone utilizar una variante de Newton-Raphson que utiliza una ecuación no lineal para aproximar la función.

Considere el valor propio $\lambda_i \in (d_i, d_{i+1})$ donde $1 \leq i \leq n-1$; el caso para $i = n$ es considerado después. λ_i es la raíz de la ecuación secular

$$f(\lambda) \equiv 1 + \sum_{j=1}^{n} \frac{v_j^2}{d_j - \lambda} = 0 \tag{3.2}$$

Primero suponemos que $\lambda_i \in (d_i, \frac{d_i+d_{i+1}}{2})$. Sea $\delta_j = d_j - d_i$ y sea

$$\psi(\mu) \equiv 1 + \sum_{j=1}^{i} \frac{v_j^2}{\delta_j - \mu} \qquad \text{y} \qquad \phi(\mu) \equiv 1 + \sum_{j=i+1}^{n} \frac{v_j^2}{\delta_j - \mu}$$

ya que

$$f(\mu + d_i) = 1 + \psi(\mu) + \phi(\mu) \equiv g(\mu),$$

nosotros buscamos la raíz $\mu_i = \lambda_i - d_i \in (0, \delta_{i+1}/2)$ de $g(\mu) = 0$.

Una propiedad importante de $g(\mu)$ es que cada diferencia $\delta_j - \mu$ puede ser evaluada con una alta exactitud relativa para cualquier $\mu \in (0, \delta_{i+1}/2)$. Debido a esta propiedad cada fracción $v_j^2/(\delta_j - \mu)$ puede ser evaluada con una alta exactitud relativa.

Nosotros ahora suponemos que $\lambda_i \in [\frac{d_i+d_{i+1}}{2}, d_{i+1})$. Sea $\delta_j = d_j - d_{i+1}$ y sea

$$\psi(\mu) \equiv 1 + \sum_{j=1}^{i} \frac{v_j^2}{\delta_j - \mu} \qquad \text{y} \qquad \phi(\mu) \equiv 1 + \sum_{j=i+1}^{n} \frac{v_j^2}{\delta_j - \mu}$$

Nosotros buscamos la raíz $\mu_i = \lambda_i - d_{i+1} \in [\delta_i/2, 0)$ de la ecuación

$$g(\mu) = f(\mu + d_{i+1}) = 1 + \psi(\mu) + \phi(\mu) = 0.$$

Para cada $\mu \in [\delta_i/2, 0)$, cada diferencia $\delta_j - \mu$ y cada relación $v_j^2/(\delta_j - \mu)$ puede ser otra vez computada con alta exactitud relativa.

Por último, consideramos el caso $i = n$. Sea $\delta_j = d_j - d_n$ y sea

$$\psi(\mu) \equiv 1 + \sum_{j=1}^{n} \frac{v_j^2}{\delta_j - \mu} \qquad \text{y} \qquad \phi(\mu) \equiv 0$$

Nosotros buscamos la raíz $\mu_n = \lambda_n - d_n \in (0, \|\mathbf{v}\|)$ de la ecuación

$$g(\mu) = f(\mu + d_n) = 1 + \psi(\mu) + \phi(\mu) = 0.$$

Para cada $\mu \in (0, \|\mathbf{v}\|)$, cada diferencia $\delta_j - \mu$ y cada relación $v_j^2/(\delta_j - \mu)$ puede ser computada con alta exactitud relativa como antes.

En el siguiente algoritmo, el valor de ρ es igual 1, ya que en nuestra implementación se escala la matriz original de forma que siempre se obtenga ese valor. De la misma forma, la matriz diagonal D, se representa con un vector d, porque al implementar el algoritmo esto reduce el uso de memoria. Los elementos del vector λ son calculados como $(d_k - d_I) - (\lambda_i - d_I)$ para tener una buena exactitud con los cálculos en coma flotante.

Algoritmo 3.2 Encontrar la i-ésima raíz de la ecuación secular

Entrada: Un vector $d \in \mathbb{R}^n$ cuyas entradas estén en orden ascendente, un vector $v \in \mathbb{R}^n$ con $\|v\|_2 = 1$, el índice $i \in \mathbb{Z}^+$ de la raíz que se va a obtener, la precisión de la máquina $\epsilon \in \mathbb{R}$

Salida: La i-ésima raíz de la ecuación secular $\lambda_i \in \mathbb{R}$, el vector $dmlambda = [d_1 - \lambda_i, d_2 - \lambda_i, \ldots, d_n - \lambda_i]^T \in \mathbb{R}^n$

$di \leftarrow d_i$
$\eta \leftarrow 1$
$g(\mu_i) \leftarrow 1$
si $i < n$ **entonces**
    $dip1 \leftarrow d_{i+1}$
    $\lambda_i \leftarrow (di + dip1)/2$
    $f(\lambda_i) \leftarrow 1 + \phi(\lambda_i) + \psi(\lambda_i)$
    si $f(\lambda_i) > 0$ **entonces** ▷ El cero está en la mitad izquierda del intervalo
        para $j = 1$ **a** n **hacer** ▷ Trasladar el origen a d_i
            $\delta_j \leftarrow d_j - di$
        $\mu_i \leftarrow \lambda_i - di$
        $dip1 \leftarrow dip1 - di$
        mientras $|g(\mu_i)| > \epsilon$ **y** $\left|\frac{\eta}{\mu_i}\right| > 8\epsilon$ **hacer** ▷ Encontrar el cero
            $g(\mu_i) \leftarrow 1 + \phi(\mu_i) + \psi(\mu_i)$
            $\Delta_i \leftarrow -\mu_i$ ▷ $\delta_i - \mu_i$, pero $\delta_i = d_i - d_i = 0$
            $\Delta_{i+1} \leftarrow dip1 - \mu_i$ ▷ $\delta_{i+1} - \mu_i$
            $a \leftarrow (\Delta_i + \Delta_{i+1})g(\mu_i) - \Delta_i\Delta_{i+1}(\psi'(\mu_i) + \phi'(\mu_i))$
            $b \leftarrow \Delta_i\Delta_{i+1}g(\mu_i)$
            $c \leftarrow g(\mu_i) - \Delta_i\psi'(\mu_i) - \Delta_{i+1}\phi'(\mu_i)$
            si $a > 0$ **entonces**
                $\eta \leftarrow \dfrac{2b}{a + \sqrt{a^2 - 4bc}}$
            si no
                $\eta \leftarrow \dfrac{a - \sqrt{a^2 - 4bc}}{2c}$
            $\mu_i \leftarrow \mu_i + \eta$
            si $\mu_i = 0$ **entonces** ▷ Prevenir división por cero
                $\mu_i \leftarrow \epsilon$
        $\lambda_i \leftarrow \mu_i + di$ ▷ Regresar el origen a cero

Algoritmo 3.2 Encontrar la i-ésima raíz de la ecuación secular

28: **si no** ▷ El cero está en la mitad derecha del intervalo
29: **para** $j = 1$ **a** n **hacer** ▷ Trasladar el origen a d_{i+1}
30: $\delta_j \leftarrow d_j - dip1$
31: $\mu_i \leftarrow \lambda_i - dip1$
32: $di \leftarrow di - dip1$
33: **mientras** $|g(\mu_i)| > \epsilon$ **y** $\left|\frac{\eta}{\mu_i}\right| > 8\epsilon$ **hacer** ▷ Encontrar el cero
34: $g(\mu_i) \leftarrow 1 + \phi(\mu_i) + \psi(\mu_i)$
35: $\Delta_i \leftarrow di - \mu_i$ ▷ $\delta_i - \mu_i$
36: $\Delta_{i+1} \leftarrow -\mu_i$ ▷ $\delta_{i+1} - \mu_i$, pero $\delta_{i+1} = d_{i+1} - d_{i+1} = 0$
37: $a \leftarrow (\Delta_i + \Delta_{i+1})g(\mu_i) - \Delta_i\Delta_{i+1}(\psi'(\mu_i) + \phi'(\mu_i))$
38: $b \leftarrow \Delta_i\Delta_{i+1}g(\mu_i)$
39: $c \leftarrow g(\mu_i) - \Delta_i\psi'(\mu_i) - \Delta_{i+1}\phi'(\mu_i)$
40: **si** $a > 0$ **entonces**
41: $\eta \leftarrow \dfrac{2b}{a + \sqrt{a^2 - 4bc}}$
42: **si no**
43: $\eta \leftarrow \dfrac{a - \sqrt{a^2 - 4bc}}{2c}$
44: $\mu_i \leftarrow \mu_i + \eta$
45: **si** $\mu_i = 0$ **entonces** ▷ Prevenir división por cero
46: $\mu_i \leftarrow -\epsilon$
47: $\lambda_i \leftarrow \mu_i + dip1$ ▷ Regresar el origen a cero
48: **si no** ▷ $i = n$
49: $dip1 \leftarrow d_n + 1$ ▷ $d_n + \|v\|_2$
50: $\lambda_i \leftarrow d_n + 0.5$ ▷ $(d_n + d_{n+1})/2 = (d_n + d_n + \|v\|_2)/2$
51: $f(\lambda_i) \leftarrow 1 + \phi(\lambda_i) + \psi(\lambda_i)$
52: **si** $f(\lambda_i) > 0$ **entonces** ▷ El cero está en la mitad izquierda del intervalo
53: **para** $j = 1$ **a** n **hacer** ▷ Trasladar el origen a d_n
54: $\delta_j \leftarrow d_j - di$
55: $\mu_i \leftarrow \lambda_i - di$
56: **mientras** $|g(\mu_i)| > \epsilon$ **y** $\left|\frac{\eta}{\mu_i}\right| > 8\epsilon$ **hacer** ▷ Encontrar el cero
57: $g(\mu_i) \leftarrow 1 + \phi(\mu_i) + \psi(\mu_i)$
58: $\Delta_i \leftarrow -\mu_i$ ▷ $\delta_n - \mu_i$, pero $\delta_n = d_n - d_n = 0$
59: $\Delta_{i-1} \leftarrow \delta_{i-1} - \mu_i$ ▷ $d_{n-1} - \lambda_i = (d_{n-1} - d_n) - (\lambda_i - d_n)$
60: $a \leftarrow (\Delta_{i-1} + \Delta_i)g(\mu_i) - \Delta_{i-1}\Delta_i(\psi'(\mu_i) + \phi'(\mu_i))$
61: $b \leftarrow \Delta_{i-1}\Delta_i g(\mu_i)$
62: $c \leftarrow g(\mu_i) - \Delta_{i-1}\psi'(\mu_i) - \Delta_i\phi'(\mu_i)$
63: **si** $a < 0$ **entonces** ▷ Nótese que cambia el operador relacional
64: $\eta \leftarrow \dfrac{2b}{a - \sqrt{a^2 - 4bc}}$ ▷ Nótese que cambia el signo de la raíz
65: **si no**
66: $\eta \leftarrow \dfrac{a + \sqrt{a^2 - 4bc}}{2c}$ ▷ Nótese que cambia el signo de la raíz
67: $\mu_i \leftarrow \mu_i + \eta$

Algoritmo 3.2 Encontrar la i-ésima raíz de la ecuación secular

si $\mu_i = 0$ **entonces** ▷ Prevenir división por cero
  $\mu_i \leftarrow \epsilon$
$\lambda_i \leftarrow \mu_i + di$ ▷ Regresar el origen a cero
si no ▷ El cero está en la mitad derecha del intervalo
  para $j = 1$ **a** n **hacer** ▷ Trasladar el origen a $d_{n+1} = d_n + \|v\|_2$
    $\delta_j \leftarrow d_j - dip1$
  $\mu_i \leftarrow \lambda_i - dip1$
  mientras $|g(\mu_i)| > \epsilon$ **y** $\left|\frac{\eta}{\mu_i}\right| > 8\epsilon$ **hacer** ▷ Encontrar el cero
    $g(\mu_i) \leftarrow 1 + \phi(\mu_i) + \psi(\mu_i)$
    $\Delta_i \leftarrow \delta_i - \mu_i$
    $\Delta_{i-1} \leftarrow \delta_{i-1} - \mu_i$
    $a \leftarrow (\Delta_{i-1} + \Delta_i)g(\mu_i) - \Delta_{i-1}\Delta_i(\psi'(\mu_i) + \phi'(\mu_i))$
    $b \leftarrow \Delta_{i-1}\Delta_i g(\mu_i)$
    $c \leftarrow g(\mu_i) - \Delta_{i-1}\psi'(\mu_i) - \Delta_i\phi'(\mu_i)$
    si $a < 0$ **entonces** ▷ Nótese que cambia el operador relacional
      $\eta \leftarrow \dfrac{2b}{a - \sqrt{a^2 - 4bc}}$ ▷ Nótese que cambia el signo de la raíz
    si no
      $\eta \leftarrow \dfrac{a + \sqrt{a^2 - 4bc}}{2c}$ ▷ Nótese que cambia el signo de la raíz
    $\mu_i \leftarrow \mu_i + \eta$
    si $\mu_i = 0$ **entonces** ▷ Prevenir división por cero
      $\mu_i \leftarrow -\epsilon$
  $\lambda_i \leftarrow \mu_i + dip1$ ▷ Regresar el origen a cero
para $j = 1$ **a** n **hacer**
  $dlambda_j \leftarrow \delta_j - \mu_i$ ▷ $d_j - \lambda_i = (d_j - d_I) - (\lambda_i - d_I), I = i$ ó $i + 1$

Algoritmo 3.3 Solución de la ecuación secular

Entrada: Un vector $d \in \mathbb{R}^n$ cuyas entradas estén en orden ascendente sin valores repetidos, un vector $v \in \mathbb{R}^n$ con $\|v\|_2 = 1$, la precisión de la máquina $\epsilon \in \mathbb{R}$

Salida: Las n raíces de la ecuación secular en el vector $\lambda \in \mathbb{R}^n$, una matriz $dfmlc \in \mathbb{R}^{n \times n}$ cuyo valor en la fila f y la columna c es $d_f - \lambda_c$

para $i = 1$ **a** n **hacer**
  Usar el algoritmo 3.2 con d, v, i, ϵ como entrada y λ_i, la columna i de $dfmlc$ como salida

3.3. Ortogonalidad de los vectores propios

Si se utiliza la ecuación 3.1 para calcular los vectores propios de $D + \mathbf{v}\mathbf{v}^T$ se puede perder la ortogonalidad de los vectores propios. Gu y Eisenstat [8] proponen las siguientes ecuaciones para evitar que esto ocurra:

$$v'_k = \text{sign}\,(v_k)\sqrt{\frac{\prod_{j=1}^{k-1}(d_k - \lambda_j)\prod_{j=k}^{n}(\lambda_j - d_k)}{\prod_{j=1}^{k-1}(d_k - d_j)\prod_{j=k+1}^{n}(d_j - d_k)}} \tag{3.3}$$

$$q_i = \frac{(D - \lambda_i I)^{-1}\mathbf{v'}}{\parallel (D - \lambda_i I)^{-1}\mathbf{v'} \parallel}$$

La solución de Gu y Eisenstat es a la vez ingeniosa y simple, ademas, no depende de la arquitectura de la máquina donde se implemente; básicamente proponen el siguiente problema inverso de valores propios:

Dado D y valores $\lambda_1, \lambda_2, \ldots, \lambda_n$ que satisfacen la propiedad de entrelazado, encontrar un vector $\mathbf{v'} = [v'_1, v'_2, \ldots, v'_n]^T$ tal que la matriz $D + \mathbf{v}\mathbf{v}^T$ tiene los valores propios prescritos $\lambda_1, \lambda_2, \ldots, \lambda_n$. La solución a este problema esta dada por la ecuación 3.3.

Ellos observaron que v'_k en la ecuación 3.3 está determinada por los valores d_i y λ_i con una gran exactitud. Por lo tanto una vez que los valores propios son calculados con un buen grado de exactitud, un vector $\mathbf{v'}$ puede calcularse, tal que los valores λ_i son los valores propios *exactos* de $D + \mathbf{v}\mathbf{v}^T$.

Antes de que se propusiera esta solución, se había utilizado doble precisión para los problemas de valores propios de precisión sencilla y cuadruple precisión (simulada en software) para los problemas de precisión doble, por lo que se tenían implementaciones que eran totalmente dependientes de la máquina.

3.4. El problema de valores propios para $D + \mathbf{v}\mathbf{v}^T$

Como se mencionó anteriormente nosotros escalamos la matriz original T con el escalar $1/(2\rho)$ por lo que el problema que resolvemos es $D + \mathbf{v}\mathbf{v}^T$, en lugar de $D + \rho\mathbf{v}\mathbf{v}^T$; ademas esto facilita tener un vector $\mathbf{v}$ con $\|\mathbf{v}\| = 1$.

Con los puntos estudiados en las secciones anteriores, se puede observar que para resolver este problema necesitamos resolver la ecuación secular con el algoritmo 3.3 y la corrección para restaurar la ortogonalidad de los vectores.

Este problema se conoce como modificación de rango uno y todos los solucionadores de valores propios basados en el algoritmo divide y vencerás tienen que resolverlo, incluso aquellos para solucionar problemas generales de valores propios, por lo que se considera uno de los puntos claves de muchas implementaciones actuales.

3.5. Deflación

La deflación es un proceso mediante el que se reduce el tamaño del problema de valores propios para $D + \mathbf{v}\mathbf{v}^T$. Esta reducción también es necesaria para cubrir las condiciones del algoritmo 3.3 que utilizamos para resolver la ecuación secular; no tener valores propios repetidos, ni elementos cero en el vector $\mathbf{v}$. Existen dos tipos de deflación. El primer tipo

de deflación se aplica cuando hay una elemento en $\mathbf{v}$ igual a cero. Si esto ocurre entonces tenemos que

$$(v_i = 0 \iff \mathbf{v}^T\mathbf{e}_i = 0) \Longrightarrow (D + \mathbf{v}\mathbf{v}^T)\mathbf{e}_i = d_i\mathbf{e}_i$$

Esta es una de ciertas soluciones que se pueden obtener inmediatamente, solo observando cuidadosamente la ecuación. Por lo tanto, si un elemento de v se aproxima a cero podemos tomar el valor propio directamente de la diagonal de D y el correspondiente vector propio es un vector coordenado.

El segundo tipo de deflación se aplica cuando hay entradas idénticas en la diagonal de D, es decir, $d_i = d_j$, con $i < j$. Si esto ocurre nosotros podemos encontrar una rotación de Givens $G(i, j, \varphi)$ tal que esta introduce un cero en la j-ésima posición de $\mathbf{v}$,

$$\begin{matrix} \\ i \to \\ \\ \\ j \to \\ \\ \end{matrix} \begin{pmatrix} \times \\ \vdots \\ \sqrt{v_i^2 + v_j^2} \\ \vdots \\ 0 \\ \vdots \\ \times \end{pmatrix} = G(i, j, \varphi)\mathbf{v} = G\mathbf{v}$$

Nótese que para *cualquier* φ,

$$G(i, j, \varphi)DG(i, j, \varphi)^T = D, \qquad d_i = d_j$$

Por lo tanto, si hay múltiples valores propios en D podemos reducir todos. excepto uno, al introducir ceros en v y después proceder como en el primer tipo de deflación.

En nuestra implementación aplicamos deflación si

$$|v_i| < 8.0\epsilon t \quad \text{o} \quad |(d_i - d_j)cs| < 8.0\epsilon t \quad \text{con} \quad t = \text{máx}\left(\text{máx}(d_i), \text{máx}(v_i)\right)$$

Utilizamos t como tolerancia por ser una aproximación a $\|D + \mathbf{v}\mathbf{v}^T\|$ y ϵ es la precision de la máquina.

Como se puede observar la aplicación primeramente de la deflación para todos los valores v_i iguales a cero y después para todos valores propios repetidos de D nos permite cumplir con las condiciones necesarias para resolver la ecuación secular de forma correcta.

La deflación cambia el problema de valores propios para $D + \mathbf{v}\mathbf{v}^T$ por el de

$$G(D + \mathbf{v}\mathbf{v}^T)G^T = D + G\mathbf{v}\mathbf{v}^TG^T = D + (G\mathbf{v})(G\mathbf{v})^T$$

Donde G es el producto de todas las rotaciones de Givens.

Sin embargo, también debemos cambiar todas las entradas igualadas a cero en v a la parte inferior del vector y guardar esta permutación para poder: 1) utilizarla una vez que

tengamos los valores propios de la ecuación secular y 2) tener los valores correspondientes en su orden original.

Con la permutación el problema se transforma en

$$\begin{pmatrix} D_{def1} + \mathbf{v}_{def}\mathbf{v}_{def}^T & 0 \\ 0 & D_{def2} \end{pmatrix} = P_{def}G(D + \mathbf{v}\mathbf{v}^T)G^T P_{def}^T = \begin{pmatrix} Q & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} \Lambda & 0 \\ 0 & D_{def2} \end{pmatrix} \begin{pmatrix} Q & 0 \\ 0 & I \end{pmatrix}^T$$

Es esta reducción la que permite al algoritmo divide y vencerás ser uno de los más eficientes, incluso en versiones secuenciales. Entre mayor sea la deflación más rápida será la ejecución del algoritmo.

3.6. Matrices de permutación

Una matriz de permutación es una matriz $P \in \mathbb{R}^{nxn}$ que es una permutación de filas (o columnas) de la matriz identidad [33]. Son una clase especial de matriz ortogonal que por medio de multiplicaciones reordena las filas o las columnas de otra matriz. Por ejemplo,

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Si una matriz $A \in \mathbb{R}^{nxm}$ es premultiplicada por P, el resultado es que re-ordena las filas de A:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \end{pmatrix}$$

Si una matriz $B \in \mathbb{R}^{mxn}$ es postmultiplicada por P, el resultado es que re-ordena las columnas de B:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{pmatrix}$$

Podemos revertir la permutación, multiplicando la matriz permutada por la inversa de la matriz de permutación que causó el cambio en primer lugar así:

$$P^T PA = A \qquad \text{y} \qquad APP^T = A$$

Por ejemplo, retomando nuestro primer ejemplo:

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

En la práctica, el uso de matrices de permutación requiere mucha memoria por lo que se usan vectores de permutación. Un vector de permutación $p \in \mathbb{Z}^{+1\times n}$o $\in \mathbb{Z}^{+n\times 1}$ es una

permutación de enteros del 1 a n que se utiliza como índice para colocar las filas o columnas en el índice correcto. Para obtener el vector de permutación a partir de la matriz de permutación, solo se multiplica P por el vector que contiene los números del 1 a n, por ejemplo:

$$p = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}$$

3.7. Algoritmo completo

Después de ver los detalles que deben ser tomados en cuenta para una correcta implementacion, podemos ver que la descomposición que buscamos tiene la forma

$$\begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} P_{sort}^T G^T P_{def}^T P_{def} G P_{sort} \left\{ \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} + \rho \mathbf{v}\mathbf{v}^T \right\} P_{sort}^T G^T P_{def}^T P_{def} G P_{sort} \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix}^T$$

que equivale a

$$\begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} P_{sort}^T G^T P_{def}^T \left(\begin{pmatrix} Q_{r1} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} \Lambda_{r1} & 0 \\ 0 & D_{def2} \end{pmatrix} \begin{pmatrix} Q_{r1} & 0 \\ 0 & I \end{pmatrix}^T \right) P_{def} G P_{sort} \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix}^T$$

por lo que los vectores propios de T son

$$Q = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} P_{sort}^T G^T P_{def}^T \begin{pmatrix} Q_{r1} & 0 \\ 0 & I \end{pmatrix}$$

y los valores propios son

$$\Lambda = \begin{pmatrix} \Lambda_{r1} & 0 \\ 0 & D_{def2} \end{pmatrix}$$

es decir

$$T = Q \Lambda Q^T$$

Capítulo 4

Diseño e implementación

Este capitulo describe el desarrollo principal de la tesis, los problemas que se encontraron al implementar los algoritmos y como se resolvieron.

Primero, se muestra en la sección 4.1, la implementación secuencial del método de disparo, y en la sección 4.2 (en la pág. 40) la implementantación secuencial del algoritmo divide y vencerás. Enseguida, en la seccion 4.3 (en la pág. 45), se explican las estrategias de paralelización que se utilizaron y cual es más adecuada para cada granularidad de datos y herramienta de programación. Después, en la sección 4.4 (en la pág. 46) se describe la implementación paralela del método de disparo, finalmente, en la sección 4.5 (en la pág. 46) se detalla la implementación paralela del algoritmo divide y vencerás, se muestra como se sincronizaron el CPU con el GPU y los mensajes que se intercambian entre los nodos.

Se explica con detalle como se implemento cada uno de los pasos que conforman los algoritmos de cada método.

4.1. Implementación secuencial del método de disparo

La implementación secuencial del método de disparo se realizo únicamente en lenguaje C, por que se conocía mucho mejor la forma de implementarlo y no existía la necesidad de hacer tantas pruebas.

Para la implementación básicamente se desarrollaron las siguientes tareas:

- Runge-Kutta de cuarto orden
- Recorrido de intervalos
- Contar el número de cruces que realiza cada función que resulta de integrar usando Runge-kutta, para saber a que valor propio corresponde
- Filtrar resultados que si cumplen con las condiciones finales e iniciales de las ecuaciones diferenciales
- Comparar errores con respecto a soluciones analíticas, cuando estas existan

Las estructuras de datos que se utilizaron fueron arreglos de longitud $n \times n$ que guardan en cada fila un discretización de las funciones de la ecuación diferencial de segundo grado, en total son dos matrices una para $y = f$ y otra para $z = f'$ que son las funciones desacopladas de la ecuación diferencial.

Se utilizaron tres arreglos de longitud n, uno para guardar la discretización del intervalo de x donde se buscan los valores propios, uno para guardar los valores de energía E, y uno para guardar los cruces de cada disparo que se intento c.

4.2. Implementación secuencial del algoritmo divide y vencerás

Se desarrollaron dos implementaciones secuenciales, una en matlab y otra en lenguaje C. La metodología que se siguió durante la tesis para implementar los algoritmos fue:

Primeramente, codificar en matlab con funciones de álgebra lineal que ya incluye matlab, el objetivo de esta fase fue aprender y comprender los algoritmos utilizados, en especial el algoritmo divide y vencerás, porque era del que menor información se tenía, gracias esta implementación se logró identificar varios errores que ocurrían al seguir de forma estricta los algoritmos utilizados en los artículos en los que nos basamos. Al darnos cuenta de que se necesitaba investigar más los detalles propios de la implementación, hicimos pruebas sobre varias versiones, hasta asegurarnos que la implementación funcionaba en todos los casos, y no solo en ejemplos específicos, esta fue la principal razón para desarrollar funciones generadoras de matrices y funciones que median el error de las soluciones encontradas. Matlab nos permitió crear estas funciones de una forma muy rápida, ademas por el hecho de mantener las variables en el espacio de trabajo nos permitió hacer diferentes experimentos sobre los resultados de cada ejecución.

Después, fuimos modificando la versión que utilizaba instrucciones vectorizadas y funciones de matlab hasta llegar a una que encajara con el tipo de programación del lenguaje c. Aunque eliminar las instrucciones vectorizadas produce una ejecución más lenta, nos proporciona otras ventajas. Estas modificaciones nos permitieron pensar como se paralelizaría cada parte del código, porque identificamos las estructuras de datos que utilizaríamos, los datos que debíamos enviar entre los nodos, y el profiler de matlab nos permitió analizar las tareas que más tiempo consumían. Además, la implementación de matlab con estructura del lenguaje C puede ayudar a otras personas interesadas en implementar el algoritmo divide y vencerás, por ejemplo, futuros estudiantes, al servirles como un buen punto de partida para realizar sus propios desarrollos en otros lenguajes, porque la implemetación esta desarrollada a un bajo nivel, que permite ver los detalles de implementación que son necesarios para obtener buenos resultados y permite hacer experimentos a futuras modificaciones de una forma rápida.

Finalmente, se codificó una version secuencial en lenguaje C, basándonos en la version de matlab, al implementar esta version pensamos en estructurarla de forma que fuera fácil portarla a la version paralela, para hacer esto, tomamos en cuenta que los modelos de ejecución paralela son operaciones de una y dos dimensiones, en los que cada elemento resultante

es calculado por un hilo de ejecución, entonces generamos métodos de la forma `<método>_r` para una dimension, y `<método>_rc` para dos dimensiones. Estos métodos normalmente se encuentran dentro de un ciclo `for` y contiene las operaciones que calculan el resultado del elemento en la fila `r` y columna `c`, así imitamos el comportamiento de los kernels de CUDA o de los `pragma` de OpenMP. Sin embargo, imitar el comportamiento de MPI no se puede lograr de esta forma y es preferible limitarse a mantener estructuras de datos separadas en lo que se convertirá en diferentes procesos sobre nodos distintos.

4.2.1. Almacenamiento de matrices

El método para almacenar matrices densas que utilizamos fue el *colum-major order*, que como su nombre lo indica consiste en almacenar los elementos de una columna en direcciones continuas de memoria, esto difiere al formato tradicional del lenguaje C, pero nos permite partir las matrices de una forma mas sencilla y tener compatibilidad con bibliotecas como cublas en la version paralela. Para tener un buen desempeño cuando recorremos las matrices recorremos todas las filas de una columna y después pasamos a la siguiente columna, porque el acceder a elementos que son continuos en memoria es mas rápido.

4.2.2. Generación de matrices y métodos de cálculo de error

En todas las versiones del código se desarrollaron funciones de generación de matrices, en especifico:

- Matrices tridiagonales T
- Matrices de modificación de rango uno $D + \mathbf{v}\mathbf{v}^T$

También en todas las versiones, se desarrollaron funciones para medir el error de las soluciones encontradas, en particular:

- $\|AQ - Q\Lambda\|_F$ para medir la calidad de los eigenpares de una matriz tridiagonal
- $\|(D + \mathbf{v}\mathbf{v}^T)Q - Q\Lambda\|_F$ para medir la calidad de los eigenpares de una modificación de rango uno
- $\|QQ^T - I\|_F$ para medir la ortogonalidad de los vectores propios

4.2.3. Estructuras de datos

Las estructuras de datos que se emplearon utilizan la menor cantidad posible de memoria:

- Matriz tridiagonal T utiliza dos arreglos; uno de longitud n para la diagonal principal y uno de longitud $n - 1$ para la subdiagonal
- Matriz de modificación de rango uno $D + \mathbf{v}\mathbf{v}^T$ utiliza dos arreglos de longitud n; uno para la diagonal principal de D y uno para los valores del vector v

- Matriz de rotaciones de givens G utiliza cuatro arreglos de longitud igual al máximo número de rotaciones que se pueden aplicar n; dos arreglos con las posiciones i, j de las rotaciones y dos arreglos con los valores s, c de las rotaciones
- Matrices densas cuadradas utilizan un arreglo de longitud $n \times n$

Para medir los tiempos de ejecución se utiliza la función `gettimeofday`, porque utilizar otras funciones como `time` de c, solo miden el uso del CPU y no del GPU. Ésta es la misma función (enmascarada) que utiliza MAGMA para medir sus tiempos.

4.2.4. Funciones principales

A grandes rasgos, las funciones principales que se implementaron para la version secuencial son:

- Merge
- Deflación
- Modificación de rango uno
  - Solución de la ecuación secular
  - Corrección de la ortogonalidad de los vectores propios
  - Calcular vectores propios
- Aplicar e invertir rotaciones de givens.
- Aplicar permutaciones simples y compuestas.

4.2.5. Escalamiento del problema original

Durante la primera fase del código se escala la matriz tridiagonal original, esta operación se realiza en todas las implementaciones que hemos observado, sin embargo, rara vez se comenta en los artículos, ya que esta operación esta directamente relacionada con no perder precisión en las operaciones de coma flotante. El escalamiento no afecta el algoritmo que utilizamos de manera significativa, y en general ningún otro que se utilice por lo siguiente:

Sea A la matriz original y B la matriz escalada

$$B = \alpha A$$
$$A = \frac{1}{\alpha} B$$

si la descomposición en valores propios de B es

$$B = QDQ^T$$

entonces

$$\begin{aligned} A &= \frac{1}{\alpha} Q D Q^T \\ &= Q \frac{1}{\alpha} D Q^T \\ &= Q \Lambda Q^T \end{aligned}$$

por lo tanto, los valores propios de A son $\frac{1}{\alpha} D$, y los vectores propios son los mismos que los de B, es decir, Q.

El escalar α que utilizamos es $\frac{1}{2\rho_o}$, donde ρ_o es el valor b_m de la matriz original; porque nos permite llegar a una modificación de rango uno del tipo $D + \mathbf{v}\mathbf{v}^T$ en lugar de $D + \rho \mathbf{v}\mathbf{v}^T$. La razón por la que esto sucede se describe a continuación.

Primero, el algoritmo que utilizamos para resolver la ecuación secular tiene como condición previa que $\|\mathbf{v}\| = 1$ pero los subproblemas dan como resultado $\mathbf{v_1}$ y $\mathbf{v_2}$, que después se concatenan para formar $\mathbf{v}$, ya que $\|\mathbf{v_1}\| = 1$ y $\|\mathbf{v_2}\| = 1$, tenemos que

$$\begin{aligned} \|\mathbf{v_1}\| &= \sqrt{\sum_{k=1}^{m} v_k^2} = 1 \Rightarrow \sum_{k=1}^{m} v_k^2 = 1, \\ \|\mathbf{v_2}\| &= \sqrt{\sum_{k=m+1}^{n} v_k^2} = 1 \Rightarrow \sum_{k=m+1}^{n} v_k^2 = 1 \end{aligned}$$

entonces

$$\|\mathbf{v}\| = \sqrt{\sum_{k=1}^{n} v_k^2} = \sqrt{\sum_{k=1}^{m} v_k^2 + \sum_{k=m+1}^{n} v_k^2} = \sqrt{1+1} = \sqrt{2}$$

para lograr que la norma sea igual a 1 dividimos $\mathbf{v_1}$ y $\mathbf{v_2}$ (lo que equivale a dividir $\mathbf{v}$) por $\sqrt{2}$, y multiplicamos ρ_o por 2 para no alterar la ecuación, con lo que obtenemos

$$D + 2\rho_o \left(\frac{1}{\sqrt{2}}\mathbf{v}\right)\left(\frac{1}{\sqrt{2}}\mathbf{v}\right)^T = D + 2\rho_o \hat{\mathbf{v}}\hat{\mathbf{v}}^T$$

entonces, tenemos $\rho = 2\rho_o$ y para tener un $\rho = 1$ necesitamos una matriz B con $\rho_e = \frac{1}{2}$, para lograr esto multiplicamos la matriz original A por el escalar $\alpha = \frac{1}{2b_m} = \frac{1}{2\rho_o}$ y obtenemos una matriz B con $\rho_e = \frac{1}{2}$ y por lo tanto un a modificación de rango uno de B con $\rho = 1$, es decir

$$D + 2\rho_e \left(\frac{1}{\sqrt{2}}\mathbf{v}\right)\left(\frac{1}{\sqrt{2}}\mathbf{v}\right)^T = D + 2\rho_e \hat{\mathbf{v}}\hat{\mathbf{v}}^T = D + 2\frac{1}{2}\hat{\mathbf{v}}\hat{\mathbf{v}}^T = D + \hat{\mathbf{v}}\hat{\mathbf{v}}^T$$

que es lo que queríamos obtener.

Por último, como habíamos observado antes, los valores propios de A son los valores propios de B multiplicados por $\frac{1}{\alpha} = 2\rho_o$ y los vectores propios son los mismos que los de la matriz escalada B.

Podríamos pensar que este escalamiento aplicado a la matriz A aumenta el tiempo de computo de nuestra implementación, pero esto no ocurre, ya que en otras implementaciones se escala utilizando como valor de α el máximo valor absoluto de las entradas de la matriz original. Ademas, nosotros evitamos otro tipo de transformaciones que ocurren por tener un valor de $\rho \neq 1$, por ejemplo, valores negativos.

4.2.6. Consideraciones en la ecuación secular

La precisión de la maquina la tomamos de la macro `DBL_EPSILON` de la biblioteca `float.h` del lenguaje C. Pero este valor es un parámetro que se puede ajustar.

En la version secuencial utilizamos Lapack para calcular la multiplicación de matrices. También, utilizamos la función `dstedc` para calcular los subproblemas del algoritmo divide y vencerás, aunque puede utilizarse cualquier otro método como el QR.

Nosotros utilizamos un criterio de parada para resolver la ecuación secular distinto del artículo de Li [7], el criterio que utilizamos es:

$$|g(\mu_i)| > \epsilon \qquad \text{y} \qquad \left|\frac{\eta}{\mu_i}\right| > 8\epsilon$$

Este criterio mostró buenos resultados al realizar pruebas en matlab, el criterio de paro es diferente porque usamos el método al que Li llama *The Middle Way*, y los criterios que usa para este método dependen del tamaño del problema, ya que nosotros manejamos problemas de una gran magnitud, estos criterios se ven demasiado afectados, llevado a malos resultados.

También, evitamos que existieran divisiones entre cero al resolver la ecuación secular, Li menciona que esto puede ocurrir pero de forma poco frecuente, sin embargo, nosotros realizamos pruebas en matlab y este fenómeno se presentaba en muchos casos, debido al tamaño de problema que nosotros estamos manejando.

Otro hecho del que nos percatamos al realizar pruebas en matlab fue que, aunque los casos de la ecuación secular que se toman en cuenta en la teoría son tres en la practica son cuatro, dividendo el tercer caso en λ_n en dos uno para cada mitad del intervalo $(0, \|\mathbf{v}\|)$ en el que se encuentra μ_n.

4.2.7. Consideraciones en las rotaciones de Givens

La transpuesta de G es igual a la multiplicación en orden inverso de cada una de las matrices transpuestas de cada rotación de givens

$$G^{-1} = G^T = (G_{nr} \dots G_2 G_1)^T = G_1^T G_2^T \dots G_{nr}^T$$

Sin embargo, como se puede notar en las estructuras de datos que utilizamos nosotros nunca tenemos almacenada en memoria la matriz G o G^T, porque ocuparían demasiado espacio, lo único que hacemos es aplicar G^T a cada columna que se requiera de Q utilizando los indices i, j y los valores c, s que definen cada rotación. De hecho G nunca se multiplica con ninguna matriz completa, solo se aplica a entradas especificas del vector d y del vector v durante la deflación, y a columnas especificas de lo que sera la matriz de vectores propios.

4.2.8. Permutaciones compuestas

Un punto muy importante en la implementación secuencial que nos trae ventajas en la version paralela es la utilización los vectores de permutación de forma compuesta, esto es, crear una nueva permutación que tenga el mismo efecto que a dos o mas permutciones, de esta forma los movimientos de memoria se reducen y por lo tanto, se obtienen mejores tiempos de ejecución.

Para obtener una nueva permutación a partir de dos permutaciones se usa el valor de la posición que resulta de la aplicación de la primera permutación como nuevo indice para la siguiente permutación, por ejemplo, nosotros utilizamos la siguiente composición:

$$k \leftarrow p_{def}(p_{sort}(i))$$

Para invertir las permutaciones que usamos para, ordenar los valores propios, y reducir el problema utilizando deflación.

4.2.9. Ventajas practicas de la deflación

Debemos destacar que en las pruebas de matlab la deflación reducía el numero de columnas en la multiplicación final del algoritmo, la que se realiza entre los vectores propios de los subproblemas y los vectores de la modificación de rango uno, en un ochenta porciento aproximadamente.

4.3. Estrategias de paralelización

Es importante mencionar que en cualquiera de las estrategias de paralelización que puedan utilizarse, las operaciones que se van a realizar deben ser lo suficientemente pesadas para compensar el overhead (creación de hilos, creación de procesos, paso de mensajes).

Básicamente se usaron dos estrategias de paralelización las cuales se definen a continuación:

4.3.1. Descomposición por datos

En esta estrategia buscamos idealmente partir los datos asociados al problema. Si es posible, partimos los datos en partes iguales y se asignan hilos de procesamiento relacionados con cada dato.

La descomposición por datos, generalmente, se realiza sobre los ciclos de las instrucciones de programación, tipicamente, ciclos `for`.

Por lo que mencionamos anteriormente, CUDA y OpenMP son los más aptos para este tipo de estrategia.

4.3.2. Descomposición por tareas

La descomposición por tareas se realiza generalmente sobre las funciones que conforman un método.

Se busca encontrar la dependencia de las funciones y realizar la mayor cantidad posible de funciones independientes en paralelo. Si estas funciones necesitan información de otras, se requiere utilizar comunicación; pero, en general, se trata de evitar esta comunicación lo más que se pueda. En MPI esta comunicación tiene que hacerse a mano, mientras que en OpenMP es prácticamente automática.

La granularidad que se utiliza es de grano grueso, para realizar una gran cantidad de operaciones en una estructura de datos y evitar comunicación. Por lo que MPI es mas apto para este tipo estrategia.

4.4. Paralelización del método de disparo

Para paralelizar el método de disparo solo se utilizó CUDA en un solo nodo, debido a que este método no utiliza tanta memoria y es más rápido que el divide y vencerás, pero sus resultados tienen errores mucho mas significativos, como veremos en la sección de resultados, por lo que este método es recomendable si se quieren soluciones inexactas y rápidas.

Se paralelizo cada una de las integraciones de runge-kutta, es decir, recorremos todo los rangos de energía posible y lanzamos un kernel de CUDA para cada integración que también cuenta el numero de cruces en paralelo. Las partes restantes del método se hacen de forma secuencial.

4.5. Paralelización del algoritmo divide y vencerás

Primeramente, describiremos el particionamiento general del problema, así como los mensajes que se intercambian, y después nos centraremos en describir, las principales funciones que se paralelizaron.

Como mencionamos anteriormente una de las razones principales para utilizar el algoritmo divide y vencerás es que podemos partir por datos la matriz en dos partes independientes, por lo que en la implementación enviamos la mitad del vector que contiene la diagonal principal y la mitad del vector que contiene la subdiagonal de la matriz original a otro nodo con MPI, resolvemos cada uno de los subproblemas que se generan debido a este particonamiento usando la función `dstedx` de MAGMA, aplicando antes ciertas transformaciones que mencionamos anteriormente (escalamiento y restar ρ) usando CUDA y OpenMP. Después enviamos la solución de regreso al nodo principal para poder aplicar la deflación y calcular la modificación de rango uno, para poder finalmente unir las dos soluciones obtenidas y formar la solución completa del problema original. Para realizar estos pasos se desarrollaron varias funciones que emplean CUDA, CUBLAS y OpenMP. Las mas importantes se describen en las siguientes subsecciones de este capitulo.

Antes de comenzar a describir cada función debemos señalar que es un error común pensar que solo por el hecho de usar una mayor cantidad de procesadores tendremos un mejor tiempo, proporcional al inverso del número de procesadores empleados, lo cual es falso. Esto ocurre debido a que, por una parte, se genera un overhead en la programación paralela derivado de la creación y destrucción de hilos, sincronización, y tiempos de comunicación, entre otras cosas; y por otra parte, existen funciones que tienen una gran dependencia funcional, por lo tanto no pueden ser paralelizadas de forma eficiente. Así, contrario al sentido común, existen versiones paralelas que son mas lentas que las secuenciales.

Por la razón antes mencionada, hay funciones que no paralizamos, por ejemplo, la deflación que es inherentemennte secuencial. En la siguiente figura se muestra las diferentes etapas de la implementación paralela. Las secciones azules representan el CPU y las verdes la GPU.

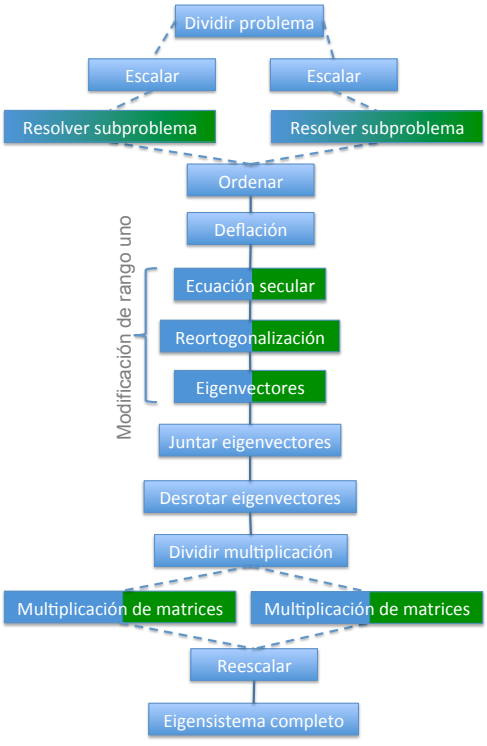

Figura 4.1: Flujo del proceso

4.5.1. Paralelización de la ecuación secular

Para paralelizar la ecuación secular, primero calculamos $\mathbf{v}^2$ en la GPU haciendo una paralelización de una dimensión, porque utilizaremos sus valores después, para cada raíz de la ecuación secular calculamos $d_f - \lambda_c$ y λ_c, ejecutando un kernel de CUDA sobre los intervalos de la ecuación secular. Como vamos a utilizar los resultados encontrados por esta función en GPU mantenemos los datos en la memoria de la GPU para evitar hacer copias de memoria innecesarias y hacer mas rapida la ejecución

En otras implementaciones esto siempre se hace de forma secuencial o con OpenMP. Así que esta implementación es una de las primeras que aplica este enfoque.

4.5.2. Paralelización de la corrección de la ortogonalidad de los vectores propios

Para paralelizar la corrección de la ortogonalidad de los valores propios utilizamos la ecuación 3.3 (en la pág 34). Empleamos los valores de $d_f - \lambda_c$ que forman una matriz de tamaño $n \times n$, aplicamos una paralelización de una dimensión para calcular cada elemento del vector $\mathbf{v}'$.

Otra vez, mantenemos los datos en la memoria de la GPU para evitar hacer copias de memoria innecesarias y hacer mas rapida la ejecución.

4.5.3. Paralelización del cálculo de los vectores propios de la modificación de rango uno

Con los valores corregidos de $\mathbf{v}'$ hacemos una paralelización de una dimension para calcular la norma con la que se divide cada vector columna de la matriz Q_{r1} de vectores propios, utilizamos un kernel de CUDA para lograr esto.

Luego utilizamos una paralelización de dos dimensiones para calcular cada elemento de la matriz Q_{r1} utilizando otro kernel de CUDA.

4.5.4. Paralelización de la aplicación de las rotaciones de Givens

Para paralelizar la aplicación de las rotaciones de Givens utilizamos un ciclo que recorre cada uno de las rotaciones de Givens y lanzamos varios hilos de OpenMP para que se aplique cada rotación en la columna correspondiente de la matriz afectada por las rotaciones.

En un principio pensábamos utilizar CUDA pero el numero de rotaciones que observamos en los experimentos de matlab eran muy pocas y el tiempo de transferencia de las columnas, sería mayor que el de aplicar la rotación.

4.5.5. Paralelización de las pertmutaciones

Para paralelizar la aplicación de las permutaciones usamos una paralelización de una dimension utilizando OpenMP. Cada hilo calcula en nuevo índice de la permutación compuesta

y copia los elementos de una columna a la nueva ubicación que le corresponde.

No utilizamos CUDA porque el tiempo de copiar las matrices a la GPU es mayor que el de permutar las columnas de la matriz.

4.5.6. Paralelización de la multiplicación de matrices

La multiplicación de matrices que se utiliza para producir los vectores propios de la matriz escalada a partir de la multiplicación de los vectores propios de los subproblemas y la modificación de rango uno se paralelizo utilizando CUBLAS ya que es la version mas eficiente que conocemos.

Ya que la multiplicación de matrices es una de las operaciones que mayor tiempo requiere, en una version posterior, se utilizó MPI para partir el problema en dos, cada nodo calcula la mitad de las columnas del resultado final.

4.5.7. Otras paralelizaciónes

Otras paralelizaciones que no son parte de las funciones principales del algoritmo, como son los escalamientos y el cálculo de errores, se paralelizaron identificando los ciclos en los que se podía emplear OpenMP.

Capítulo 5

Pruebas, resultados y discusión

Este capítulo inicia en la sección 5.1 describiendo la infraestructura en la que se realizaron las pruebas. Enseguida, se presentan en la sección 5.2 (en la pág. 52) los resultados del tiempo y en la sección 5.3 (en la pág. 55) del error que se obtuvieron y se comparan con las implementaciones existentes. Después, en las secciones 5.4 (en la pág. 59) y 5.5 (en la pág. 68) se presentan los casos de estudio que se seleccionaron, los cuales son casos específicos de la ecuación de Schrödinger y Dirac, específicamente, el problema del pozo infinito. Se describe la metodología con la que se realizaron las pruebas, por ejemplo, tamaño del problema, intervalo de energía, tolerancia de error. Y por ultimo, en la sección 5.6 (en la pág. 73) se discute la relevancia de los resultados.

5.1. Infraestructura

Durante el desarrollo de este trabajo se emplearon dos equipos de cómputo, un servidor y una laptop. En la laptop se escribió la mayoría del código y se realizaron pequeñas pruebas, sin embargo, todos los resultados presentados en este capitulo fueron ejecutados en el servidor. Cuando nos referimos a los resultados de una implmentación que utiliza una sola tarjeta, nos referimos a la Tesla 2070, y cuando nos referimos a los resultados de una implementación que utiliza multiples tarjetas nos referimos a la Tesla 2070 y a la GeForce GTX 460.

5.1.1. Software

Para el desarrollo de esta tesis se requirió que los equipos de cómputo cumplan con los requerimientos de software de:

- CUDA
- MPI
- OpenMP
- MKL
- MAGMA (requiere CUDA, CPU BLAS, LAPACK)

5.1.2. Hardware

En esta tesis se utilizaron equipos de cómputo que cuentan con tarjetas gráficas que soportan la tecnología CUDA de Nvidia y con procesadores que pueden utilizar la biblioteca MKL de Intel. A continuación se mencionan las características de cada uno:

1. Servidor

   - Procesador doble zócalo Intel Xenon X5675 6 cores a 3.07 GHz con 12288kB de cache
   - Memoria RAM 24GB
   - Tarjeta de vídeo
     - Nvidia Tesla 2070 6GB con 448 CUDA Cores a 1.15 GHz
     - Nvidia GeForce GTX 460 1GB con 336 CUDA Cores a 1.35 GHz

2. Laptop

   - Procesador Intel Core i5 2 cores a 2.5 GHz con 3072kB de cache
   - Memoria RAM 4 GB
   - Tarjeta de vídeo GeForce GT 630m 1GB con 96 CUDA cores a 800 MHz

5.2. Tiempo

Las pruebas de tiempo que se realizaron fueron: la comparación entre las diferentes implementaciones del algoritmo divide y vencerás, la comparación entre la implementación secuencial y la paralela del método de disparo, y por ultimo, una comparación entre las versiones paralelas de disparo y diagonalización que se desarrollaron en esta tesis.

En la figura 5.1 (en la pág. 53), las etiquetas se refieren a las siguientes implementaciones:

dstedc: Rutina divide y vencerás secuencial de MAGMA
cuppen seq: Implementación secuencial desarrollada en esta tesis
dstedx: Rutina divide y vencerás paralela de MAGMA (12 hilos, 1 GPUs)
cuppen hib: Implementación paralela híbrida desarrollada en esta tesis (2 nodos con 6 hilos y 1 GPU cada uno)

Se puede observar que la implementación paralela híbrida es la que tiene menor tiempo de ejecución a partir de un tamaño de matriz de 8000 y que la diferencia con respecto a las otras implementaciones se incrementa conforme el tamaño de la matriz aumenta. También, se puede apreciar que, aunque pequeña, existe una mejora de la version secuencial desarrollada durante la tesis con respecto de la función equivalente de MAGMA. Las matrices para obtener estos resultado se obtuvieron con la función `dlarnv` de LAPACK.

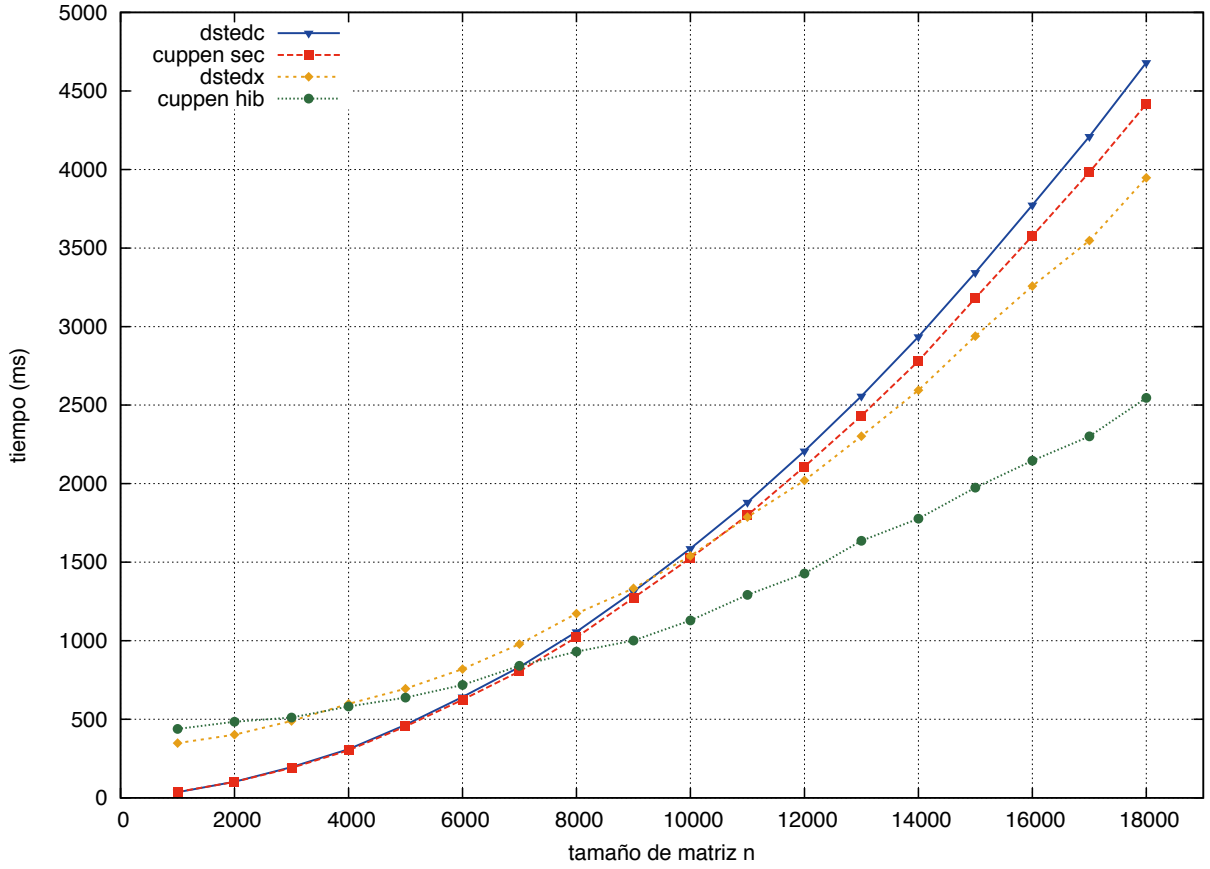

Figura 5.1: Tiempo Eigensistema Simétrico Tridiagonal

n	dstedc	secuencial	dstedx	híbrido
1000	35.31166	37.33702	348.03122	438.65449
2000	102.14143	100.76133	402.26580	484.71602
3000	195.90396	190.97439	489.56982	510.66672
4000	309.85035	302.20978	598.61821	581.94785
5000	464.09548	454.94648	694.29419	637.95799
6000	641.83560	624.25859	819.95210	718.35530
7000	832.40324	804.41708	978.27698	840.65067
8000	1056.75981	1022.35738	1172.71708	931.07416
9000	1312.63305	1272.95771	1333.19804	1000.84300
10000	1586.35967	1525.64633	1539.87303	1128.72409
11000	1881.45098	1798.57443	1786.86850	1292.16257
12000	2207.90081	2108.81810	2020.06893	1427.85470
13000	2557.15771	2431.70690	2302.09985	1636.04010
14000	2935.44659	2779.96399	2595.15100	1776.99729
15000	3343.56440	3181.02741	2939.45126	1974.21070
16000	3773.65241	3577.17584	3257.57760	2146.62823
17000	4210.57625	3983.19193	3547.57581	2300.57378
18000	4680.37327	4416.38142	3946.83685	2545.83336

Tabla 5.1: Tiempo (ms) - Diagonalización

A continuación se muestra los tiempos del método de disparo secuencial y paralelo. Los tiempos de la version paralela son mejores a partir de un tamaño de matriz de 9000 por una diferencia significativa.

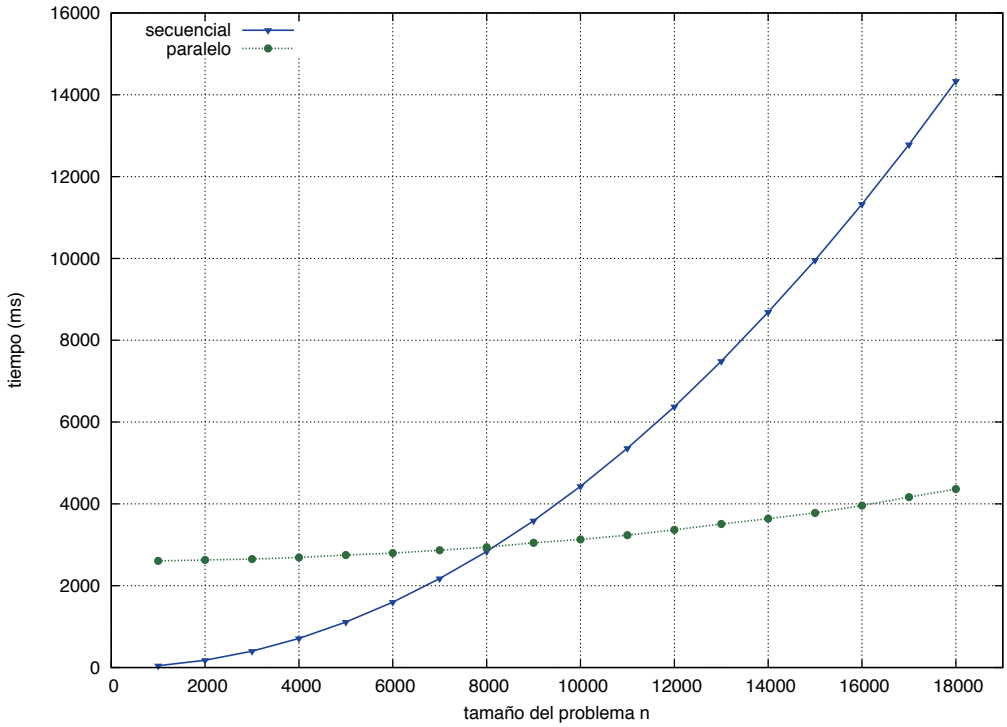

Figura 5.2: Tiempo Disparo

n	secuencial	paralelo
1000	44.64525	2607.34151
2000	178.06841	2628.85657
3000	399.96982	2651.46947
4000	710.29340	2690.00610
5000	1110.25182	2748.36331
6000	1598.70083	2797.48094
7000	2173.66274	2868.15224
8000	2836.87063	2939.17863
9000	3589.17456	3049.15062
10000	4430.07476	3129.69670
11000	5359.71229	3234.42832
12000	6376.33826	3364.79503
13000	7487.94553	3509.53713
14000	8686.60390	3639.79764
15000	9957.13353	3779.28516
16000	11326.13269	3960.45426
17000	12784.51238	4167.36297
18000	14331.73421	4364.26204

Tabla 5.2: Tiempo (ms) - Disparo

El método de disparo muestra un mejor tiempo que el de diagonalización en el problema de Schrödinger, sin embargo, presenta varios inconvenientes, como veremos mas adelante.

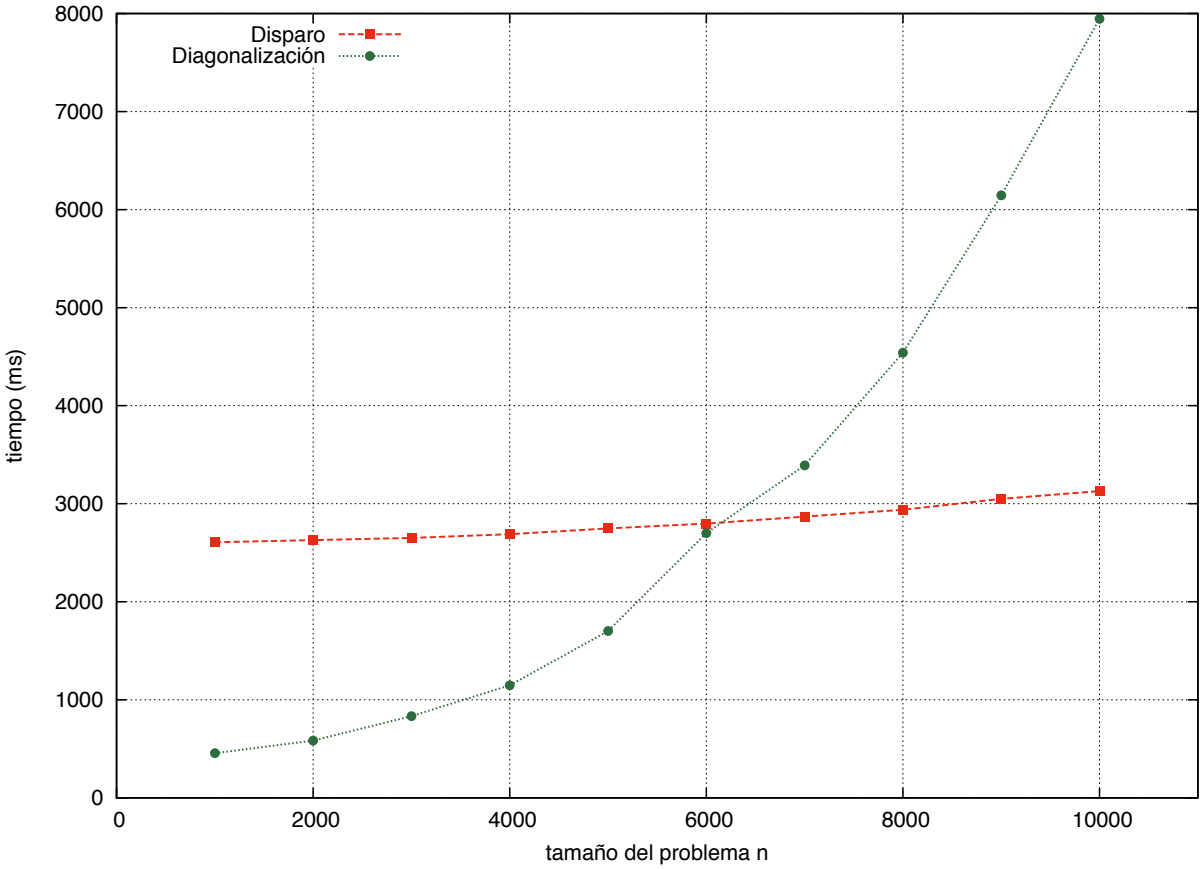

Figura 5.3: Tiempo Diagonalización vs Disparo

n	diagonalización	disparo
1000	455.67820	2607.34151
2000	584.53394	2628.85657
3000	833.77009	2651.46947
4000	1148.19218	2690.00610
5000	1702.04420	2748.36331
6000	2700.62526	2797.48094
7000	3391.34500	2868.15224
8000	4539.96886	2939.17863
9000	6145.23960	3049.15062
10000	7945.36903	3129.69670

Tabla 5.3: Tiempo (ms) - Digonalización vs Disparo

5.3. Error

El error respecto a la solución analítica en el método de disparo va aumentando conforme se incrementa índice del valor propio i en las tablas 5.4 y 5.5 (en las págs 56 y 57).

i	E_{disparo}	$E_{\text{analítica}}$	$\lvert E_{\text{disparo}} - E_{\text{analítica}} \rvert$
0	2.500000000E-01	2.500000000E-01	1.665335000E-16
1	1.000000000E+00	1.000000000E+00	6.661338000E-16
2	2.250000000E+00	2.250000000E+00	1.367795000E-13
3	4.000000000E+00	4.000000000E+00	3.295142000E-13
4	6.250000000E+00	6.250000000E+00	4.218847000E-13
5	9.000000000E+00	9.000000000E+00	4.511946000E-13
6	1.225000000E+01	1.225000000E+01	1.350031000E-12
7	1.600000000E+01	1.600000000E+01	3.428369000E-12
8	2.025000000E+01	2.025000000E+01	1.765699000E-12
9	2.500000000E+01	2.500000000E+01	7.570833000E-12
10	3.025000000E+01	3.025000000E+01	1.399769000E-11
11	3.600000000E+01	3.600000000E+01	6.799894000E-12
12	4.225000000E+01	4.225000000E+01	7.780443000E-12
13	4.900000000E+01	4.900000000E+01	2.349765000E-11
14	5.625000000E+01	5.625000000E+01	4.038014000E-11
15	6.400050000E+01	6.400000000E+01	4.999999000E-04
16	7.225000000E+01	7.225000000E+01	1.907097000E-11
17	8.100000000E+01	8.100000000E+01	2.270895000E-11
18	9.025050000E+01	9.025000000E+01	5.000001000E-04
19	1.000005000E+02	1.000000000E+02	5.000001000E-04
20	1.102510000E+02	1.102500000E+02	1.000000000E-03
21	1.210005000E+02	1.210000000E+02	5.000002000E-04
22	1.322510000E+02	1.322500000E+02	1.000000000E-03
23	1.440010000E+02	1.440000000E+02	1.000000000E-03
24	1.562515000E+02	1.562500000E+02	1.500000000E-03
25	1.690020000E+02	1.690000000E+02	2.000000000E-03
26	1.822525000E+02	1.822500000E+02	2.500001000E-03
27	1.960030000E+02	1.960000000E+02	3.000001000E-03
28	2.102540000E+02	2.102500000E+02	4.000001000E-03
29	2.250045000E+02	2.250000000E+02	4.500001000E-03
30	2.402560000E+02	2.402500000E+02	6.000001000E-03
31	2.560070000E+02	2.560000000E+02	7.000001000E-03
32	2.722580000E+02	2.722500000E+02	8.000000000E-03
33	2.890100000E+02	2.890000000E+02	1.000000000E-02
34	3.062615000E+02	3.062500000E+02	1.150000000E-02
35	3.240140000E+02	3.240000000E+02	1.400000000E-02
36	3.422665000E+02	3.422500000E+02	1.650000000E-02
37	3.610190000E+02	3.610000000E+02	1.900000000E-02
38	3.802725000E+02	3.802500000E+02	2.250000000E-02
39	4.000260000E+02	4.000000000E+02	2.600000000E-02
40	4.202800000E+02	4.202500000E+02	3.000000000E-02
41	4.410345000E+02	4.410000000E+02	3.450000000E-02
42	4.622900000E+02	4.622500000E+02	4.000000000E-02
43	4.840460000E+02	4.840000000E+02	4.600000000E-02

Tabla 5.4: Error Schrodinger - Disparo secuencial

i	E_{disparo}	$E_{\text{analítica}}$	$\lvert E_{\text{disparo}} - E_{\text{analítica}} \rvert$
0	2.500000000E-01	2.500000000E-01	1.665335000E-16
1	1.000000000E+00	1.000000000E+00	6.661338000E-16
2	2.250000000E+00	2.250000000E+00	1.367795000E-13
3	4.000000000E+00	4.000000000E+00	3.295142000E-13
4	6.250000000E+00	6.250000000E+00	4.218847000E-13
5	9.000000000E+00	9.000000000E+00	4.511946000E-13
6	1.225000000E+01	1.225000000E+01	1.350031000E-12
7	1.600000000E+01	1.600000000E+01	3.428369000E-12
8	2.025000000E+01	2.025000000E+01	1.765699000E-12
9	2.500000000E+01	2.500000000E+01	7.570833000E-12
10	3.025000000E+01	3.025000000E+01	1.399769000E-11
11	3.600000000E+01	3.600000000E+01	6.799894000E-12
12	4.225000000E+01	4.225000000E+01	7.780443000E-12
13	4.900000000E+01	4.900000000E+01	2.349765000E-11
14	5.625000000E+01	5.625000000E+01	4.038014000E-11
15	6.400050000E+01	6.400000000E+01	4.999999000E-04
16	7.225000000E+01	7.225000000E+01	1.907097000E-11
17	8.100000000E+01	8.100000000E+01	2.270895000E-11
18	9.025050000E+01	9.025000000E+01	5.000001000E-04
19	1.000005000E+02	1.000000000E+02	5.000001000E-04
20	1.102510000E+02	1.102500000E+02	1.000000000E-03
21	1.210005000E+02	1.210000000E+02	5.000002000E-04
22	1.322510000E+02	1.322500000E+02	1.000000000E-03
23	1.440010000E+02	1.440000000E+02	1.000000000E-03
24	1.562515000E+02	1.562500000E+02	1.500000000E-03
25	1.690020000E+02	1.690000000E+02	2.000000000E-03
26	1.822525000E+02	1.822500000E+02	2.500001000E-03
27	1.960030000E+02	1.960000000E+02	3.000001000E-03
28	2.102540000E+02	2.102500000E+02	4.000001000E-03
29	2.250045000E+02	2.250000000E+02	4.500001000E-03
30	2.402560000E+02	2.402500000E+02	6.000001000E-03
31	2.560070000E+02	2.560000000E+02	7.000001000E-03
32	2.722580000E+02	2.722500000E+02	8.000000000E-03
33	2.890100000E+02	2.890000000E+02	1.000000000E-02
34	3.062615000E+02	3.062500000E+02	1.150000000E-02
35	3.240140000E+02	3.240000000E+02	1.400000000E-02
36	3.422665000E+02	3.422500000E+02	1.650000000E-02
37	3.610190000E+02	3.610000000E+02	1.900000000E-02
38	3.802725000E+02	3.802500000E+02	2.250000000E-02
39	4.000260000E+02	4.000000000E+02	2.600000000E-02
40	4.202800000E+02	4.202500000E+02	3.000000000E-02
41	4.410345000E+02	4.410000000E+02	3.450000000E-02
42	4.622900000E+02	4.622500000E+02	4.000000000E-02
43	4.840460000E+02	4.840000000E+02	4.600000000E-02

Tabla 5.5: Error Schrodinger - Disparo paralelo

Enseguida, se muestran los errores en la ortogonalidad de vectores y calidad de eigenpares de la diagonalización. Los cuales son muy pequeños y competitivos con los de MAGMA.

n	dstedc	secuencial	dstedx	híbrido
1000	3.3723369844E-14	3.5413769356E-14	7.4104475159E-14	7.5036783575E-14
2000	4.9398476990E-14	5.1367406559E-14	1.1052185317E-13	1.1125178030E-13
3000	6.7936238273E-14	6.9484776725E-14	1.1488396569E-13	1.1728491432E-13
4000	6.8277837802E-14	6.8595825769E-14	1.5173536169E-13	1.5310466613E-13
5000	8.3165445355E-14	8.3235126550E-14	1.3989947843E-13	1.4010080491E-13
6000	9.5906644696E-14	9.8137120626E-14	1.6425158788E-13	1.6444466674E-13
7000	8.7656842344E-14	9.0313872848E-14	1.9224298618E-13	1.9240266758E-13
8000	9.8618629484E-14	9.8620096840E-14	2.1786978075E-13	2.1821778045E-13
9000	1.0716251386E-13	1.0721458910E-13	1.7616753669E-13	1.7699042452E-13
10000	1.2034528831E-13	1.1964212195E-13	1.9752980618E-13	1.9713012391E-13
11000	1.2671703344E-13	1.2785366930E-13	2.1270844237E-13	2.1289474806E-13
12000	1.3612085986E-13	1.3623394032E-13	2.3216059187E-13	2.3148186096E-13
13000	1.1750286765E-13	1.1817585790E-13	2.5341161930E-13	2.5405762016E-13
14000	1.2632224702E-13	1.2830361872E-13	2.7310021148E-13	2.7304164991E-13
15000	1.3313258569E-13	1.3269465964E-13	2.9037108885E-13	2.9082110894E-13
16000	1.3996586693E-13	1.3926127194E-13	3.0885916790E-13	3.0903029814E-13
17000	1.4600930335E-13	1.4629183361E-13	2.3706382836E-13	2.3766649328E-13
18000	1.5318494207E-13	1.5377167077E-13	2.5012001619E-13	2.5094679493E-13

Tabla 5.6: Error eigenpares $\|AQ - Q\Lambda\|_F$ - Diagonalización

n	dstedc	secuencial	dstedx	híbrido
1000	4.3802856897E-14	4.3570526479E-14	9.6165845438E-14	9.5292645245E-14
2000	6.0912254622E-14	6.1048348421E-14	1.3357093703E-13	1.3589374259E-13
3000	8.2550838579E-14	8.3427828554E-14	1.4338898864E-13	1.4251964259E-13
4000	8.5860079899E-14	8.5279543160E-14	1.8935550011E-13	1.9042942868E-13
5000	1.0089762655E-13	1.0176485185E-13	1.7020492757E-13	1.6988878976E-13
6000	1.1768677191E-13	1.2004190969E-13	2.0551915435E-13	2.0190836071E-13
7000	1.1042394809E-13	1.0996379415E-13	2.3634190295E-13	2.3741945772E-13
8000	1.2284015421E-13	1.2073745792E-13	2.6884925622E-13	2.6935767979E-13
9000	1.3350431955E-13	1.3330456724E-13	2.1657775931E-13	2.1782623808E-13
10000	1.4502119547E-13	1.4440528649E-13	2.4113627370E-13	2.4067837246E-13
11000	1.5632377426E-13	1.5456816024E-13	2.6280490301E-13	2.6168743190E-13
12000	1.6604734840E-13	1.6539571596E-13	2.8493752577E-13	2.8605796097E-13
13000	1.4759834337E-13	1.4705769933E-13	3.0880873178E-13	3.1185606763E-13
14000	1.5738605695E-13	1.5667447350E-13	3.3201961762E-13	3.3312402366E-13
15000	1.6526583665E-13	1.6303979166E-13	3.5780551225E-13	3.5765971002E-13
16000	1.7370763913E-13	1.7284304094E-13	3.7780342866E-13	3.7999840330E-13
17000	1.8064360251E-13	1.7990722331E-13	2.9124512819E-13	2.9083486608E-13
18000	1.8776952570E-13	1.8834346100E-13	3.1171240631E-13	3.0874162283E-13

Tabla 5.7: Error ortogonalidad $\|QQ^T - I\|_F$ - Diagonalización

A continuación, se muestra la tabla con el error de la diagonalización híbrida de matrices con respecto a la solución analítica para el problema de Schrödinger. El cual, es muy pequeño y mucho menor que el del método de disparo. También se muestra el error de la ortogonalidad de vectores y calidad de eigenpares.

n	$\|AQ - Q\Lambda\|_F$	$\|QQ^T - I\|_F$	$\|E_{\text{diagonalización}} - E_{\text{analítica}}\|$
1000	1.22645524043179E-13	1.17657912596398E-13	1.89045842511042E-15
2000	1.86785904435307E-13	2.02689701523453E-13	2.08486308500503E-15
3000	2.27668436012644E-13	2.92843251204076E-13	1.70298048316227E-15
4000	3.16896071971424E-13	3.76369181625732E-13	1.81455987084739E-15
5000	3.46510503075467E-13	4.68284011291394E-13	2.01528963843077E-15
6000	4.23242278834166E-13	5.59249282620112E-13	1.96988744779653E-15
7000	5.01955777605751E-13	6.57178492262681E-13	2.09725963790725E-15
8000	5.44568727373566E-13	7.22458636075449E-13	1.81238253262755E-15
9000	5.59048336561493E-13	8.16715994459751E-13	2.00809809519925E-15
10000	6.17935928379211E-13	9.09085051267901E-13	1.99061966115400E-15

Tabla 5.8: Error Schrodinger - Diagonalización

5.4. Schrödinger

En las siguientes figuras se muestran los resultados de las primeras diez soluciones de la diagonalización paralela del problema de Schrödinger.

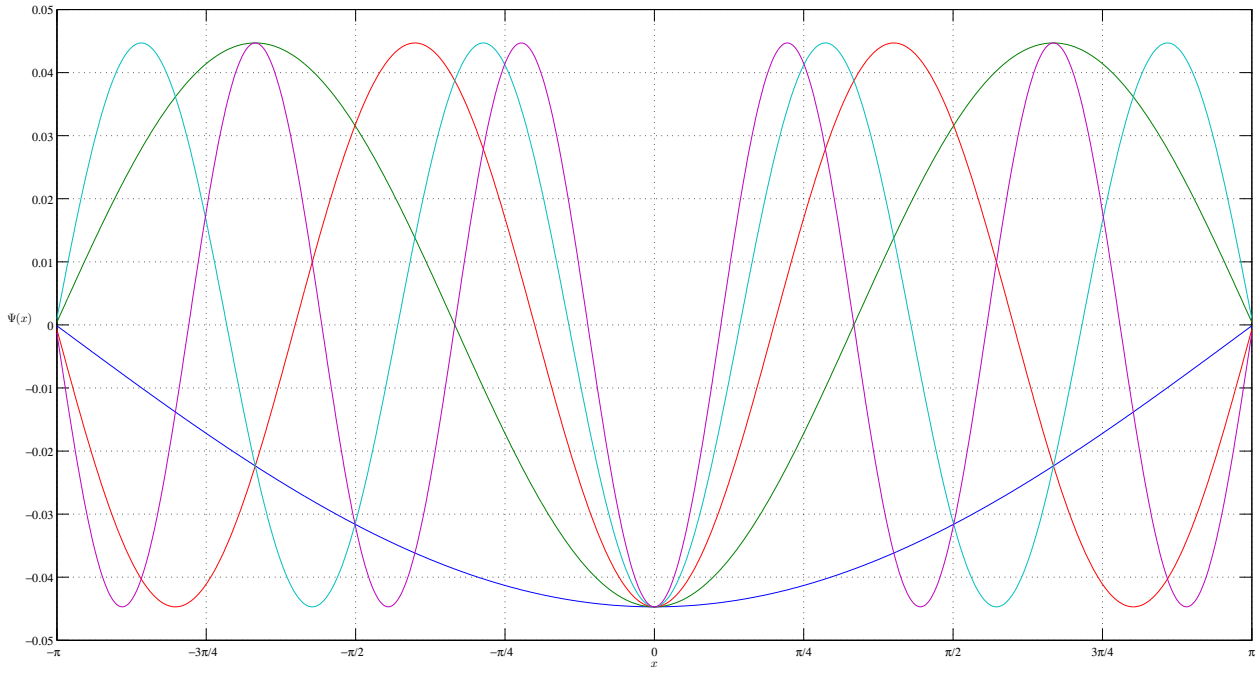

Figura 5.4: Soluciones pares Schrödinger - Diagonalización

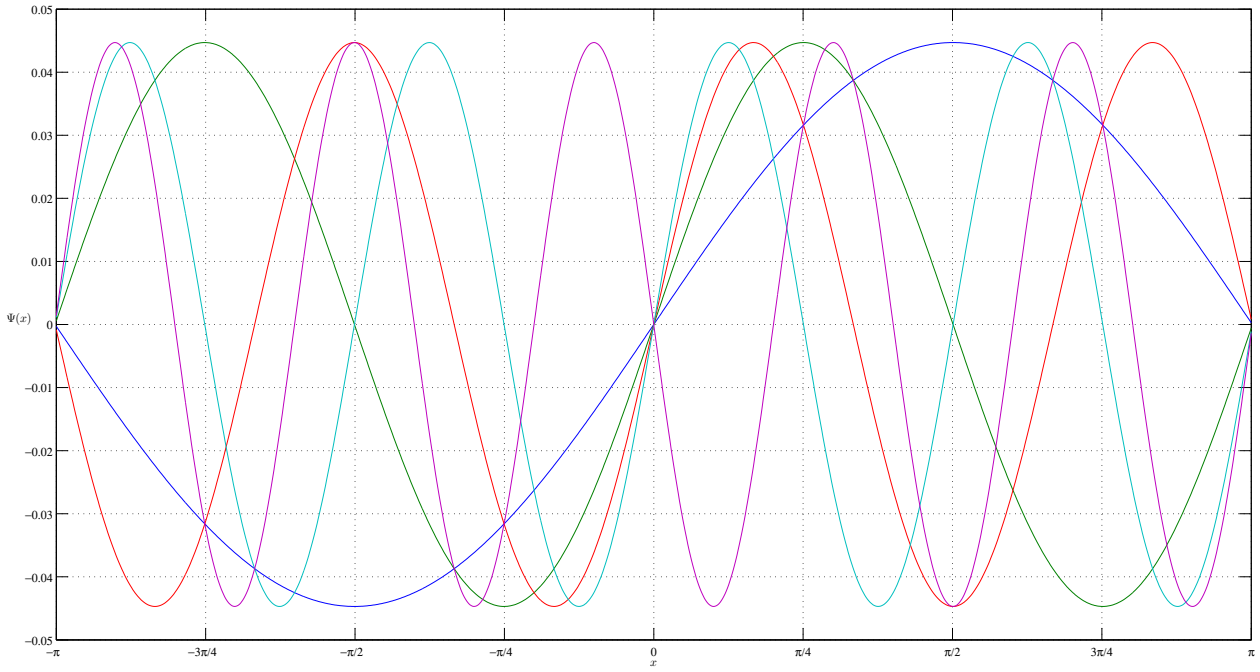

Figura 5.5: Soluciones impares Schrödinger - Diagonalización

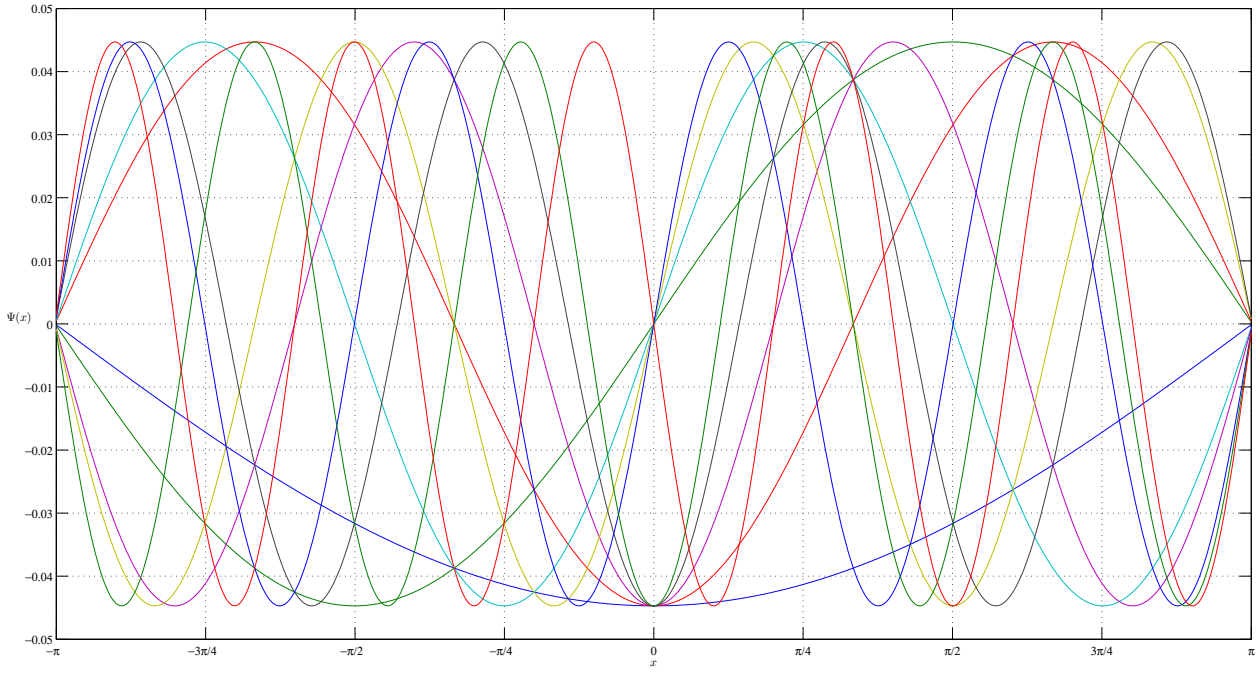

Figura 5.6: Soluciones Schrödinger - Diagonalización

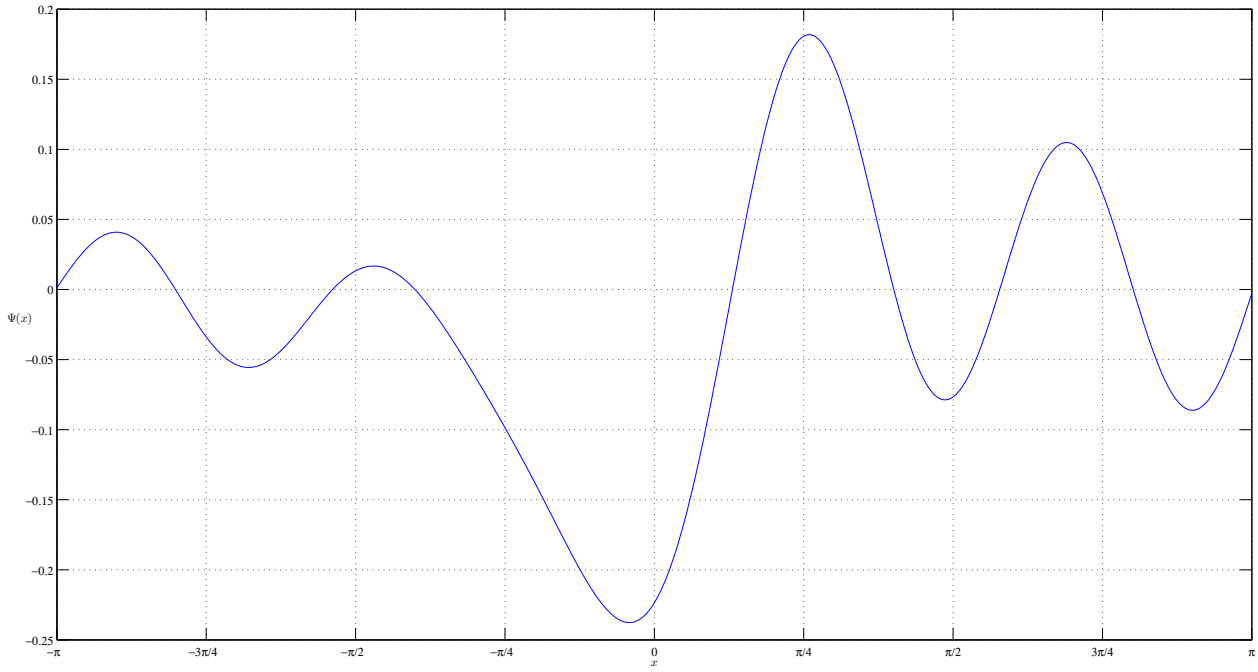

Figura 5.7: Suma soluciones Schrödinger - Diagonalización

A continuación, se presentan las diez primeras soluciones del método de disparo paralelo para el problema de Schrödinger con $y = 0, z = 1$. A partir de aquí y en lo que resta de este capitulo $y = \Psi_0$ y $z = \Psi_0'$, es decir los valores de la función y la derivada de Ψ en el punto inicial del método de disparo.

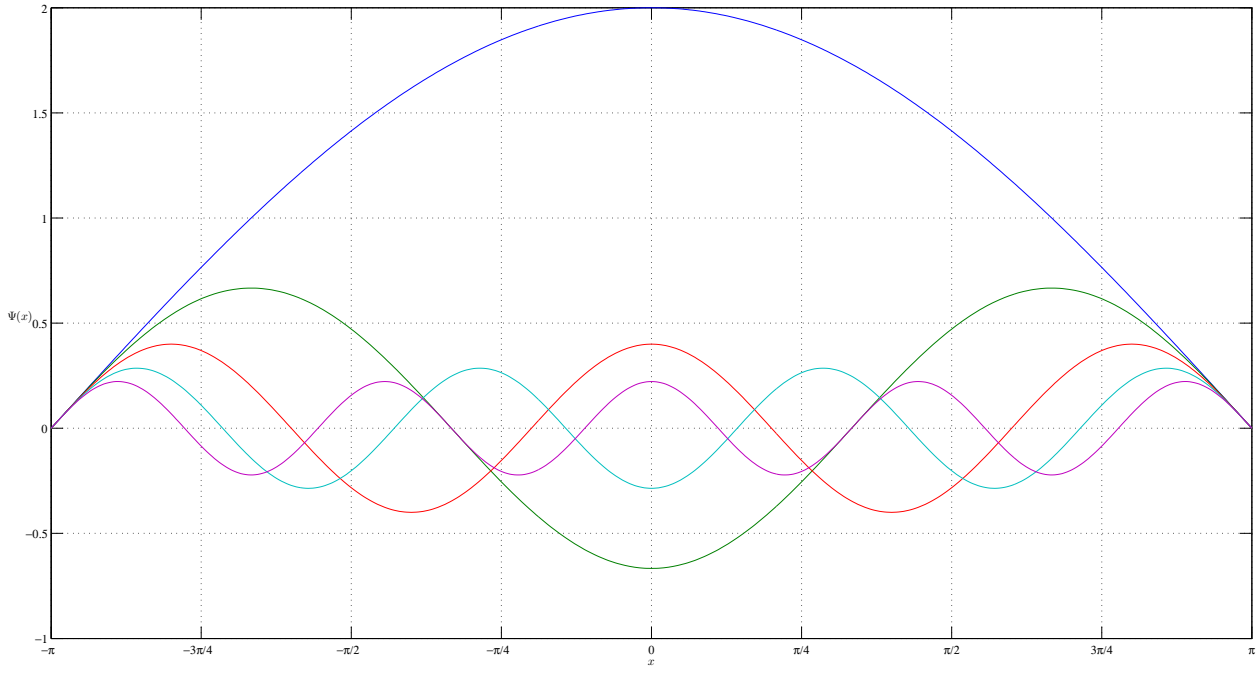

Figura 5.8: Soluciones pares Schrödinger - Disparo ($y = 0, z = 1$)

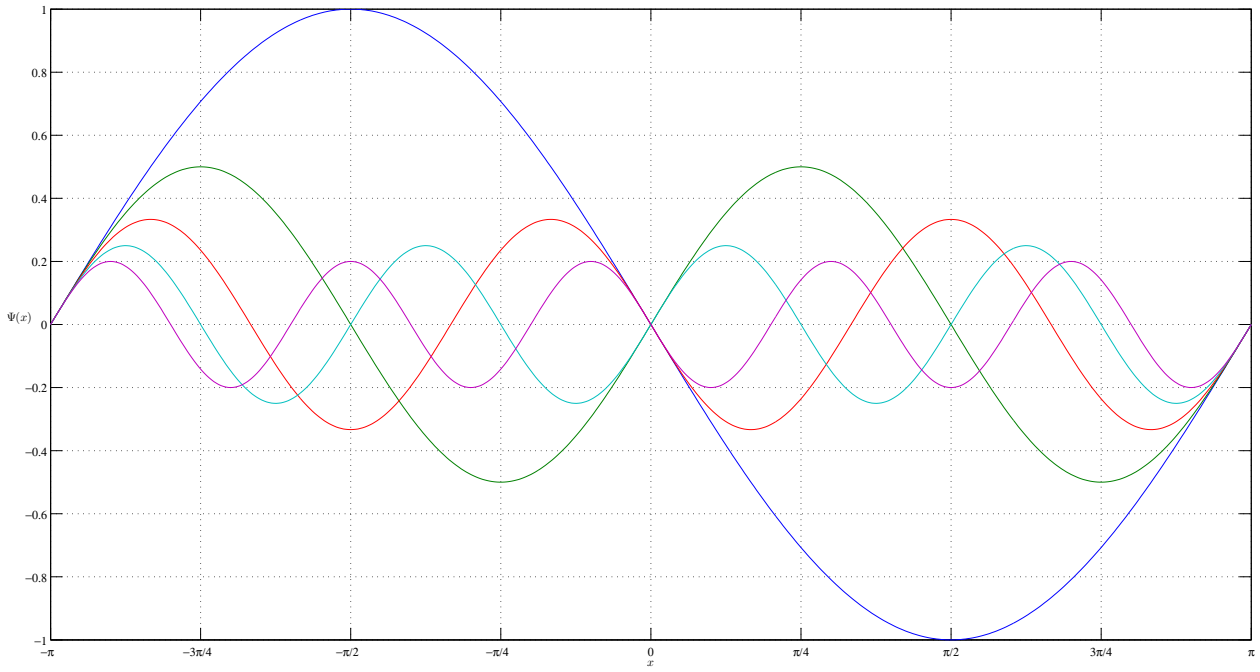

Figura 5.9: Soluciones impares Schrödinger - Disparo $(y = 0, z = 1)$

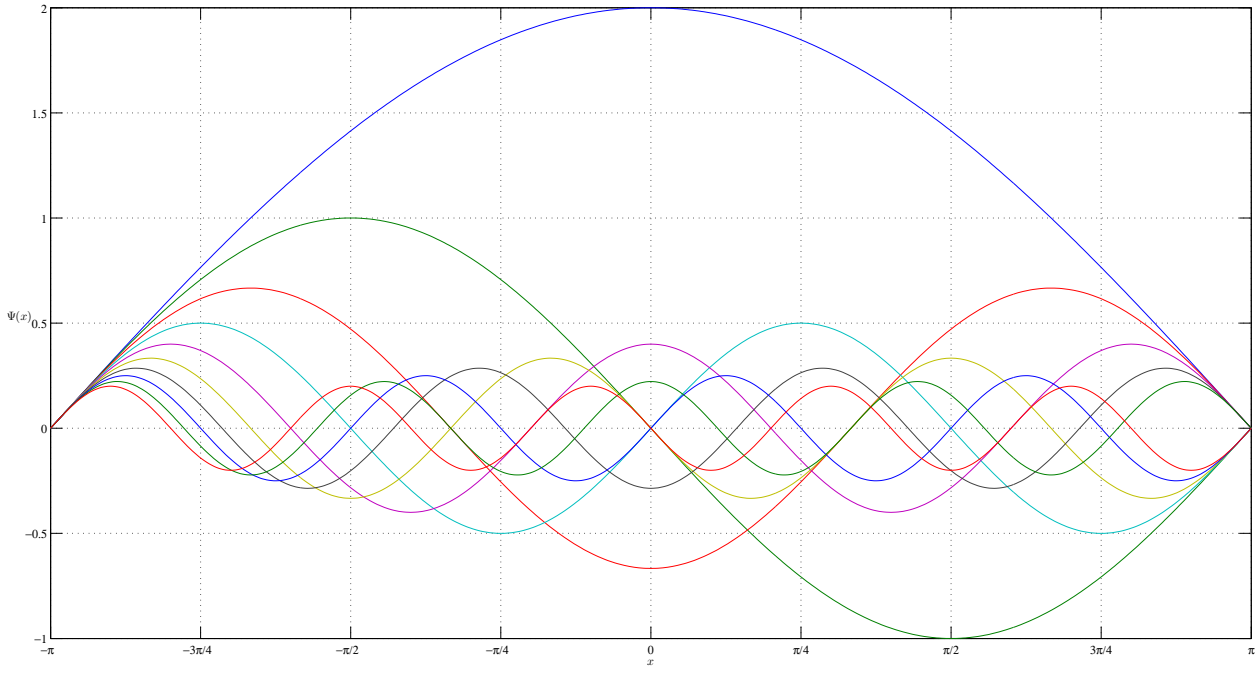

Figura 5.10: Soluciones Schrödinger - Disparo $(y = 0, z = 1)$

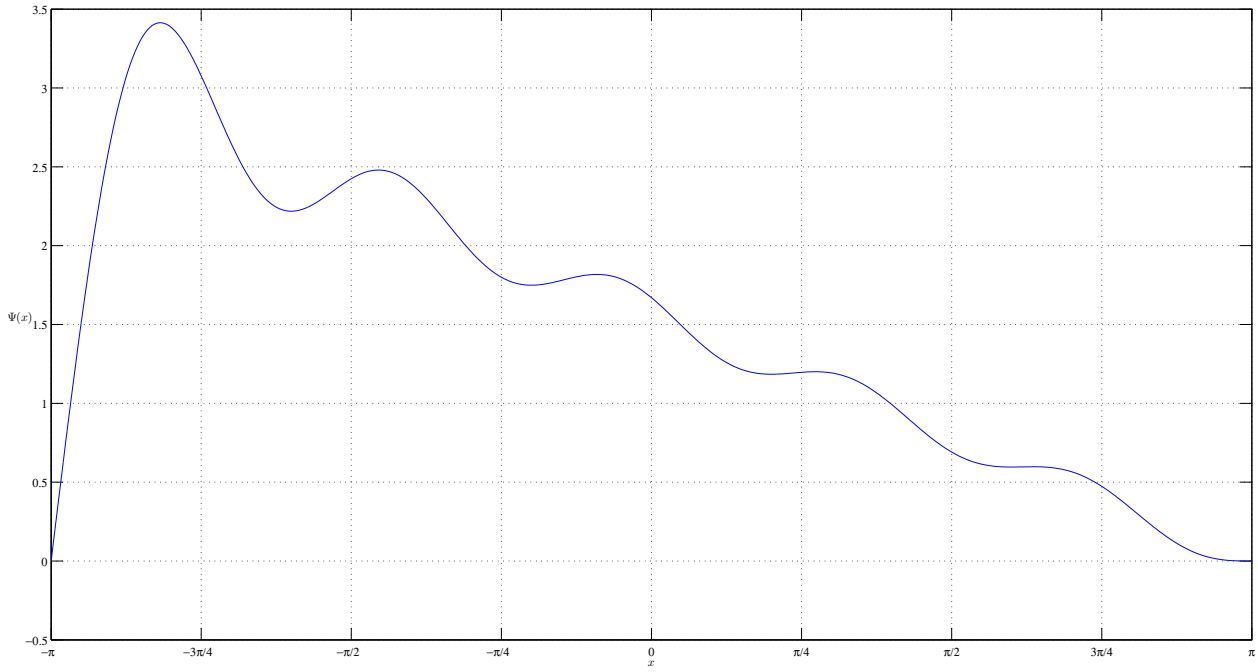

Figura 5.11: Suma Soluciones Shrödinger - Disparo ($y = 0, z = 1$)

A continuación, se presentan las diez primeras soluciones del método de disparo paralelo para el problema de Schrödinger con $y = 0, z = -1$.

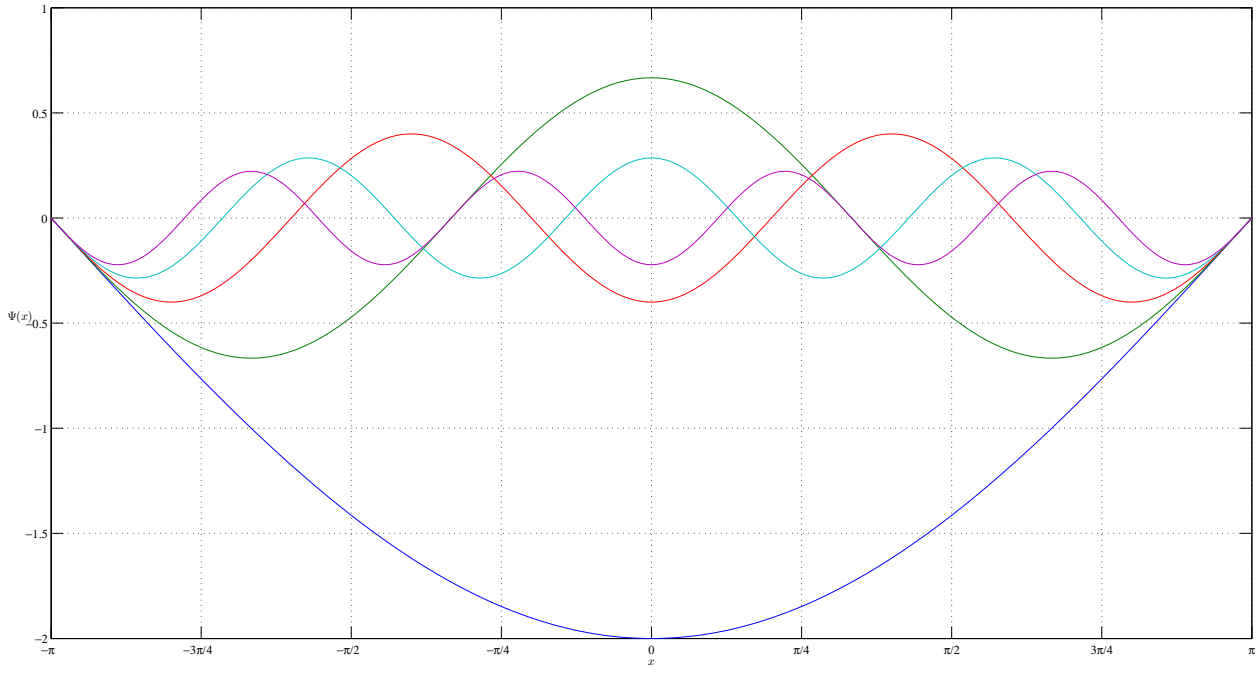

Figura 5.12: Soluciones pares Schrödinger - Disparo ($y = 0, z = -1$)

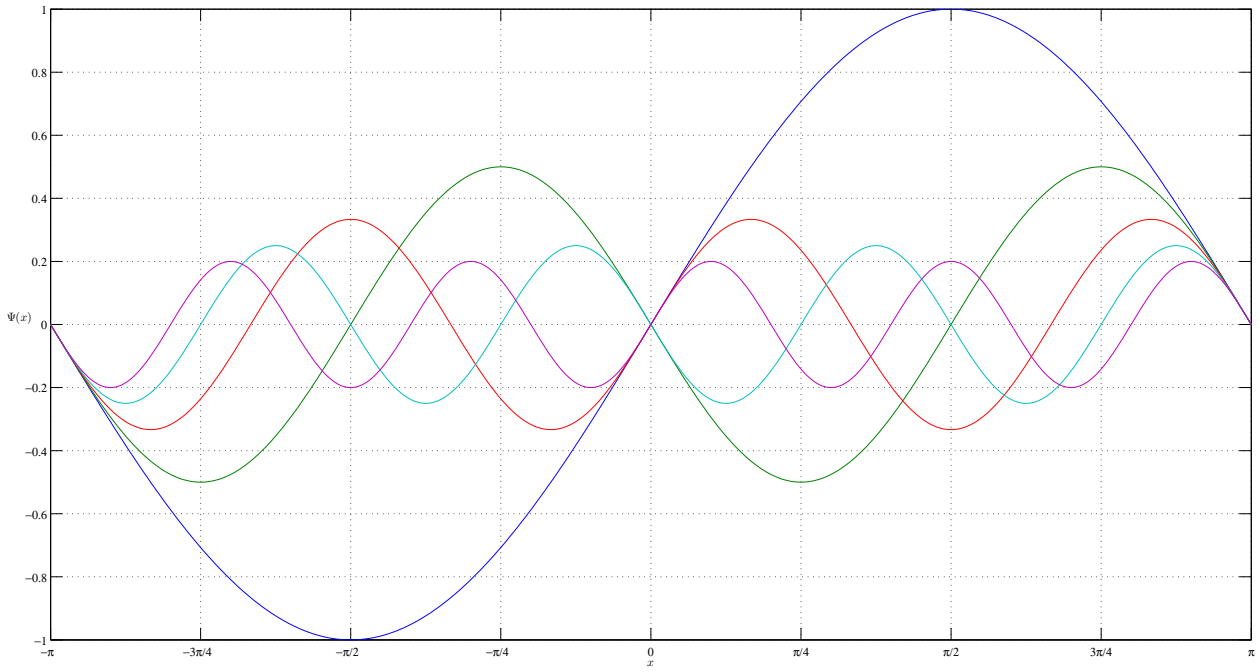

Figura 5.13: Soluciones impares Schrödinger - Disparo ($y = 0, z = -1$)

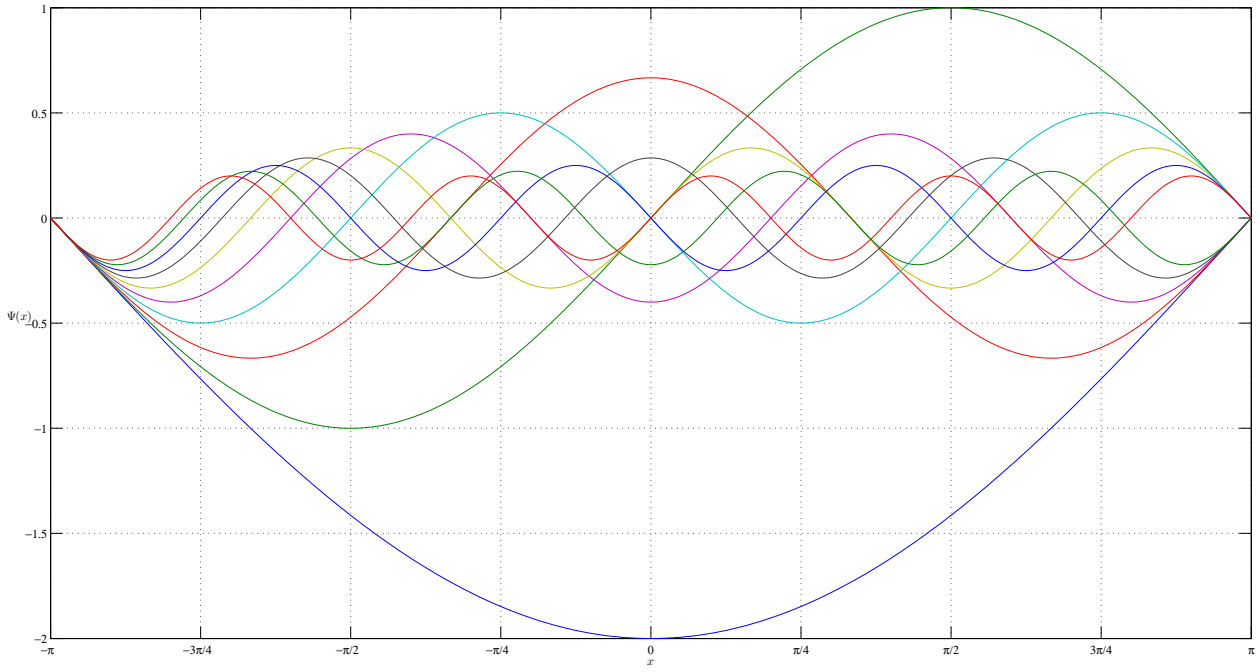

Figura 5.14: Soluciones Schrödinger - Disparo ($y = 0, z = -1$)

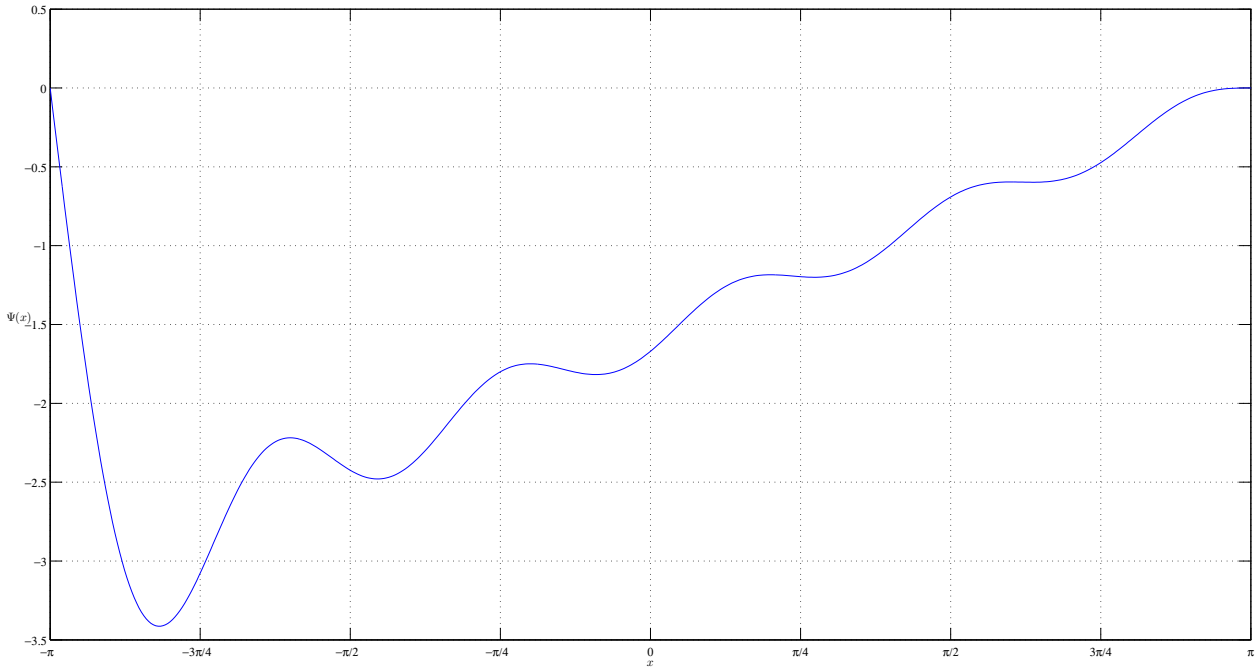

Figura 5.15: Suma Soluciones Shrödinger - Disparo ($y = 0, z = -1$)

Las siguientes tablas muestran el valor de la energía que se obtuvo E_{disparo}, y el valor de la función Ψ en el ultimo punto Ψ_n, del método de disparo para el problema de Shrödinger. En la tabla 5.9 (en la pág. 66) se muestran los resultados de la version secuencial y en la tabla 5.10 (en la pág. 67) se muestran los resultados de la version paralela. Se puede observar que los valores energía tiene un orden ascendente y que preservan la propiedad de entrelazado. Un punto en contra de nuestras implementaciones es que los valores de Ψ_n se alejan de cero conforme se buscan valores propios con un índice mayor. Para obtener estos resultados se utilizaron los siguientes parámetros: $E_i = 0, E_f = 100, \Delta E = 0.001$; $x_i = -\pi, x_f = \pi, \Delta x = 0.0125664$; $\epsilon = 0.0001$

i	Ψ_n	E_{disparo}
0	8.160331000E-11	2.500000000E-01
1	-1.305607000E-09	1.000000000E+00
2	6.609377000E-09	2.250000000E+00
3	-2.088649000E-08	4.000000000E+00
4	5.098509000E-08	6.250000000E+00
5	-1.057066000E-07	9.000000000E+00
6	1.957992000E-07	1.225000000E+01
7	-3.339542000E-07	1.600000000E+01
8	5.348004000E-07	2.025000000E+01
9	-8.149011000E-07	2.500000000E+01
10	1.192743000E-06	3.025000000E+01
11	-1.689942000E-06	3.600000000E+01
12	2.326054000E-06	4.225000000E+01
13	-3.126939000E-06	4.900000000E+01
14	4.118656000E-06	5.625000000E+01
15	1.921388000E-05	6.400050000E+01
16	6.788868000E-06	7.225000000E+01
17	-8.528146000E-06	8.100000000E+01
18	-6.823048000E-06	9.025050000E+01
19	2.723776000E-06	1.000005000E+02
20	-1.272159000E-05	1.102510000E+02
21	-6.005330000E-06	1.210005000E+02
22	-1.087153000E-06	1.322510000E+02
23	-5.039522000E-06	1.440010000E+02
24	1.438843000E-06	1.562515000E+02
25	2.397536000E-07	1.690020000E+02
26	-1.686318000E-07	1.822525000E+02
27	-1.522974000E-06	1.960030000E+02
28	-2.729682000E-06	2.102540000E+02
29	-2.435475000E-06	2.250045000E+02
30	-4.105856000E-06	2.402560000E+02
31	1.558231000E-06	2.560070000E+02
32	2.988212000E-06	2.722580000E+02
33	1.414212000E-06	2.890100000E+02
34	2.392683000E-06	3.062615000E+02
35	1.161219000E-06	3.240140000E+02
36	-1.436178000E-06	3.422665000E+02
37	-1.383529000E-06	3.610190000E+02
38	-1.105241000E-06	3.802725000E+02
39	-4.966827000E-08	4.000260000E+02
40	9.331154000E-07	4.202800000E+02
41	-1.924546000E-06	4.410345000E+02
42	-3.346333000E-08	4.622900000E+02
43	9.443011000E-07	4.840460000E+02

Tabla 5.9: Solución Schrödinger ($y = 0, z = 1$) - Disparo secuencial

i	Ψ_n	E_{disparo}
0	8.160573000E-11	2.500000000E-01
1	-1.305610000E-09	1.000000000E+00
2	6.609186000E-09	2.250000000E+00
3	-2.088623000E-08	4.000000000E+00
4	5.098530000E-08	6.250000000E+00
5	-1.057067000E-07	9.000000000E+00
6	1.957988000E-07	1.225000000E+01
7	-3.339535000E-07	1.600000000E+01
8	5.348006000E-07	2.025000000E+01
9	-8.149020000E-07	2.500000000E+01
10	1.192744000E-06	3.025000000E+01
11	-1.689943000E-06	3.600000000E+01
12	2.326054000E-06	4.225000000E+01
13	-3.126937000E-06	4.900000000E+01
14	4.118654000E-06	5.625000000E+01
15	1.921389000E-05	6.400050000E+01
16	6.788867000E-06	7.225000000E+01
17	-8.528146000E-06	8.100000000E+01
18	-6.823045000E-06	9.025050000E+01
19	2.723772000E-06	1.000005000E+02
20	-1.272158000E-05	1.102510000E+02
21	-6.005335000E-06	1.210005000E+02
22	-1.087147000E-06	1.322510000E+02
23	-5.039529000E-06	1.440010000E+02
24	1.438850000E-06	1.562515000E+02
25	2.397454000E-07	1.690020000E+02
26	-1.686231000E-07	1.822525000E+02
27	-1.522983000E-06	1.960030000E+02
28	-2.729672000E-06	2.102540000E+02
29	-2.435485000E-06	2.250045000E+02
30	-4.105846000E-06	2.402560000E+02
31	1.558220000E-06	2.560070000E+02
32	2.988218000E-06	2.722580000E+02
33	1.414211000E-06	2.890100000E+02
34	2.392679000E-06	3.062615000E+02
35	1.161226000E-06	3.240140000E+02
36	-1.436188000E-06	3.422665000E+02
37	-1.383515000E-06	3.610190000E+02
38	-1.105258000E-06	3.802725000E+02
39	-4.964828000E-08	4.000260000E+02
40	9.330928000E-07	4.202800000E+02
41	-1.924521000E-06	4.410345000E+02
42	-3.349060000E-08	4.622900000E+02
43	9.443304000E-07	4.840460000E+02

Tabla 5.10: Solución Schrödinger ($y = 0, z = 1$) - Disparo paralelo

5.5. Dirac

A continuación, se presentan las diez primeras soluciones del método de disparo paralelo para el problema de Dirac con $y = 0, z = 1$.

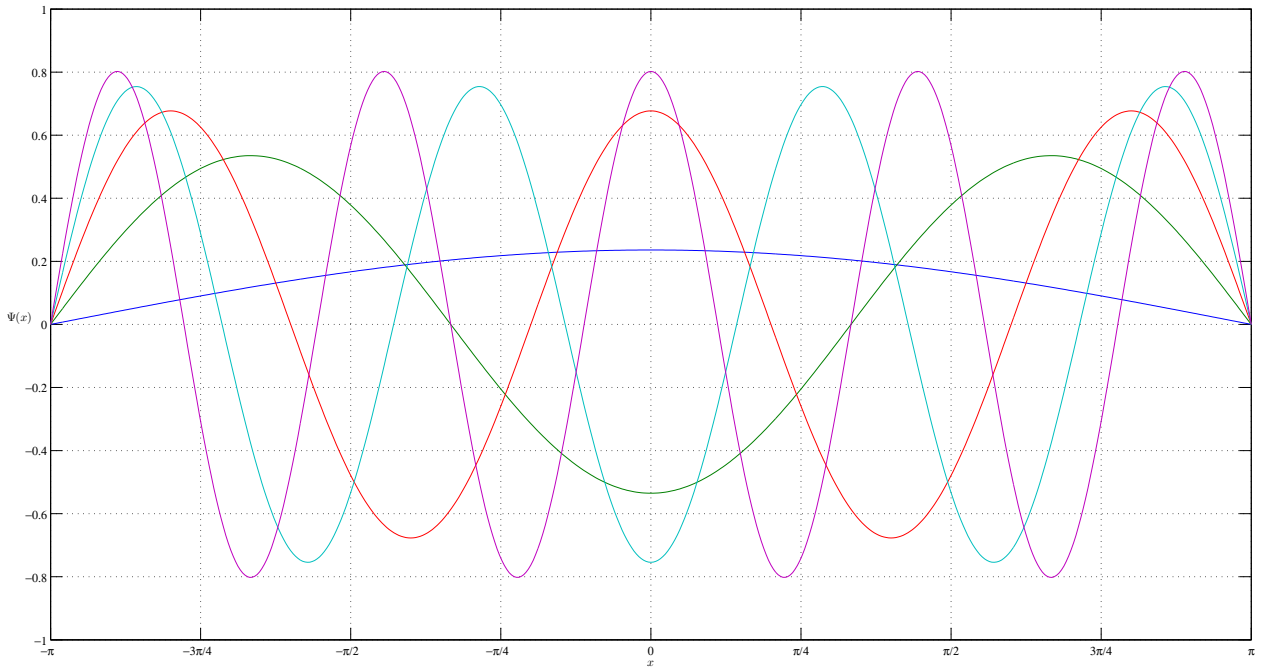

Figura 5.16: Soluciones pares Dirac - Disparo $(y = 0, z = 1)$

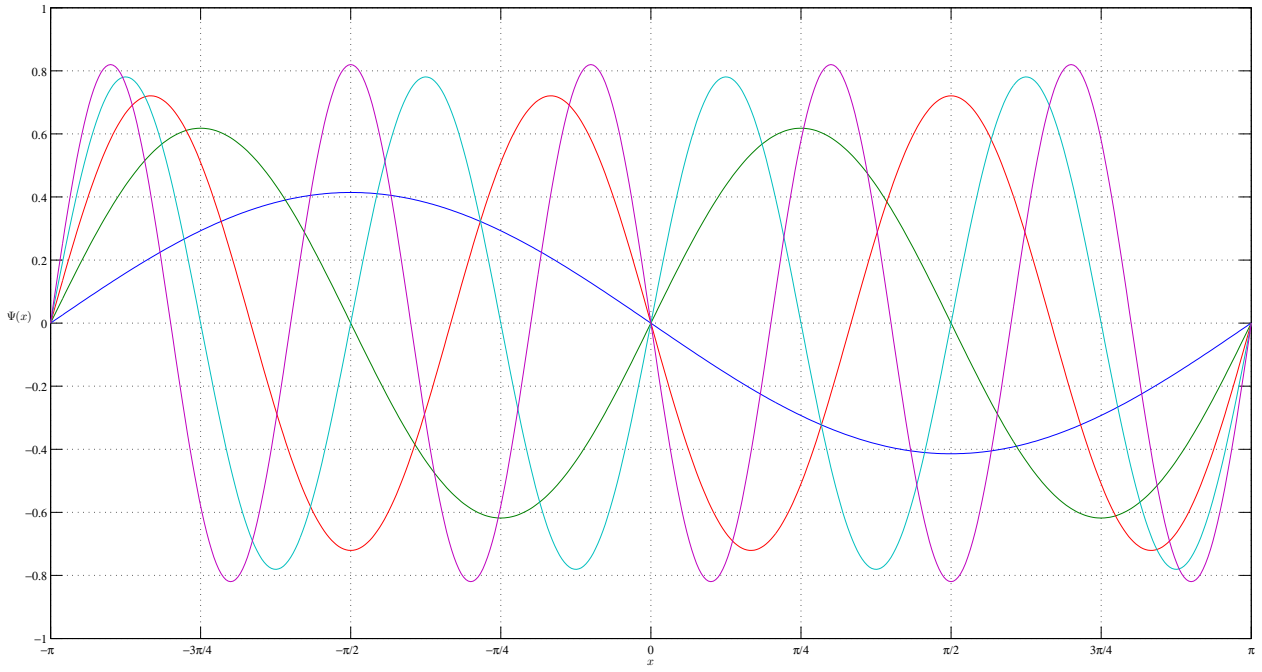

Figura 5.17: Soluciones impares Dirac - Disparo $(y = 0, z = 1)$

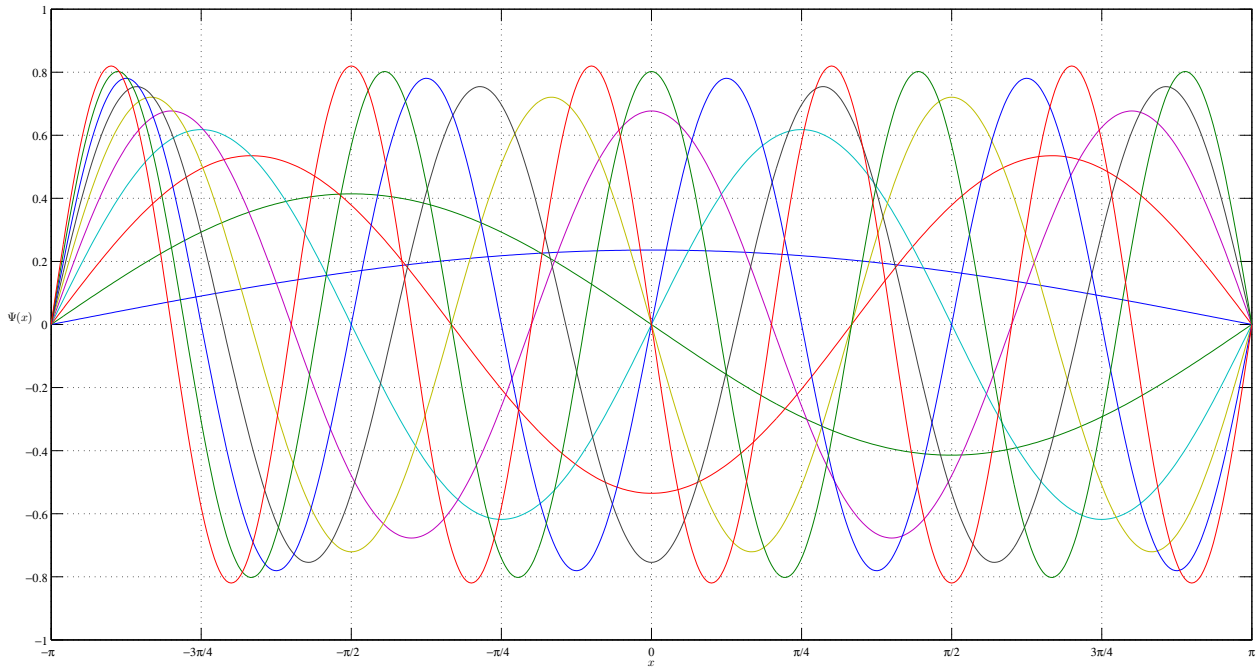

Figura 5.18: Soluciones Dirac - Disparo $(y = 0, z = 1)$

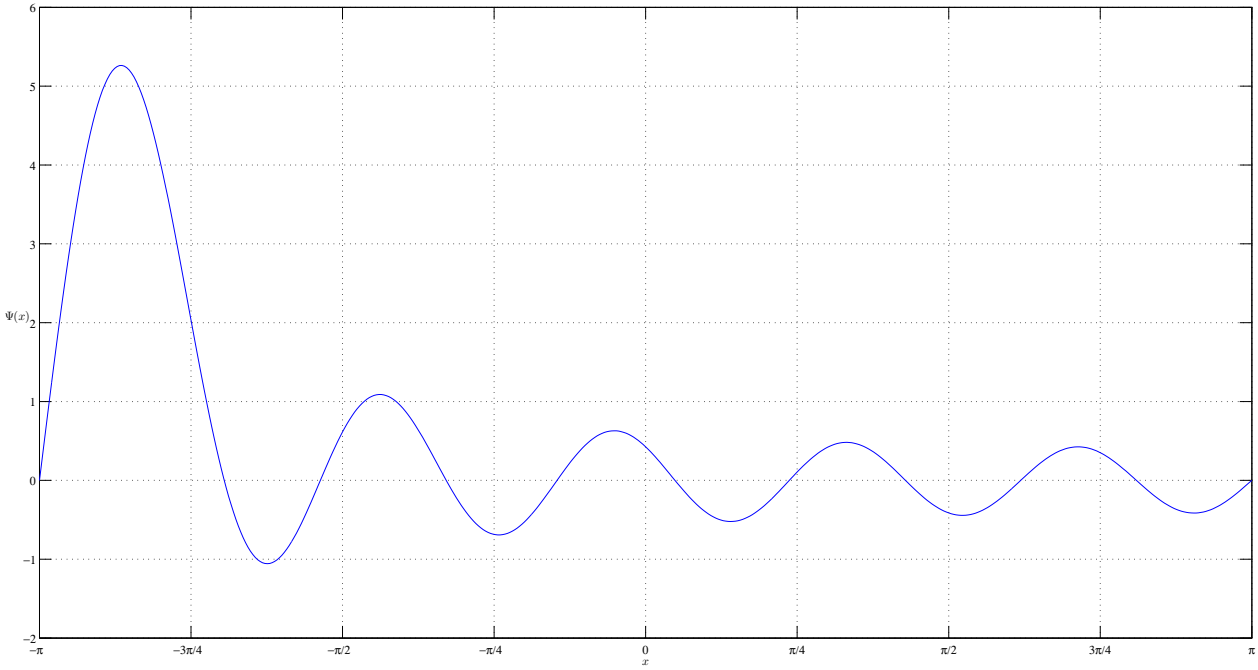

Figura 5.19: Suma Soluciones Dirac - Disparo $(y = 0, z = 1)$

A continuación, se presentan las diez primeras soluciones del método de disparo paralelo para el problema de Dirac con $y = 0, z = -1$.

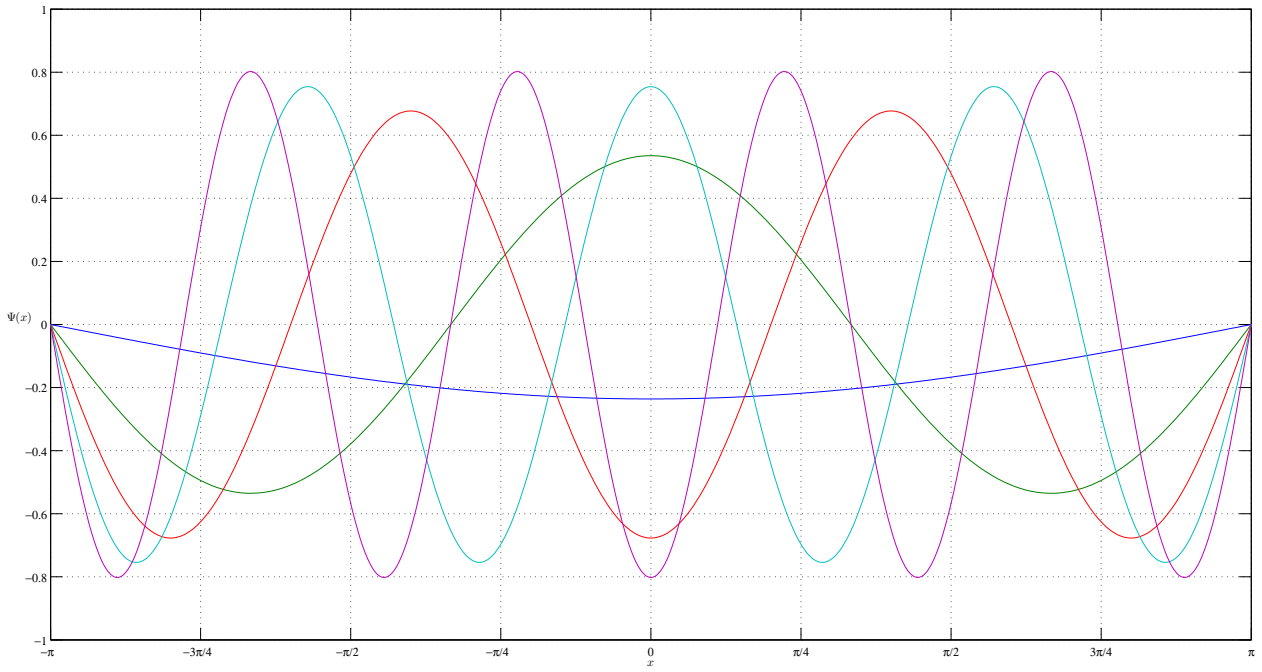

Figura 5.20: Soluciones pares Dirac - Disparo $(y = 0, z = -1)$

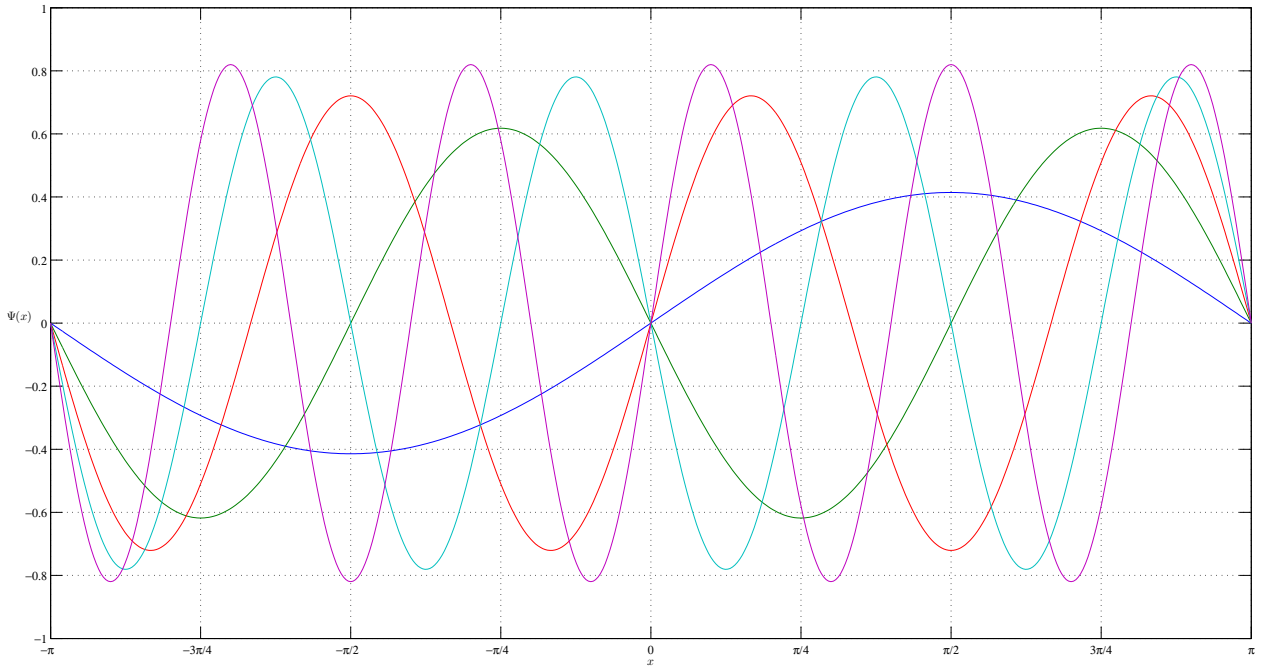

Figura 5.21: Soluciones impares Dirac - Disparo $(y = 0, z = -1)$

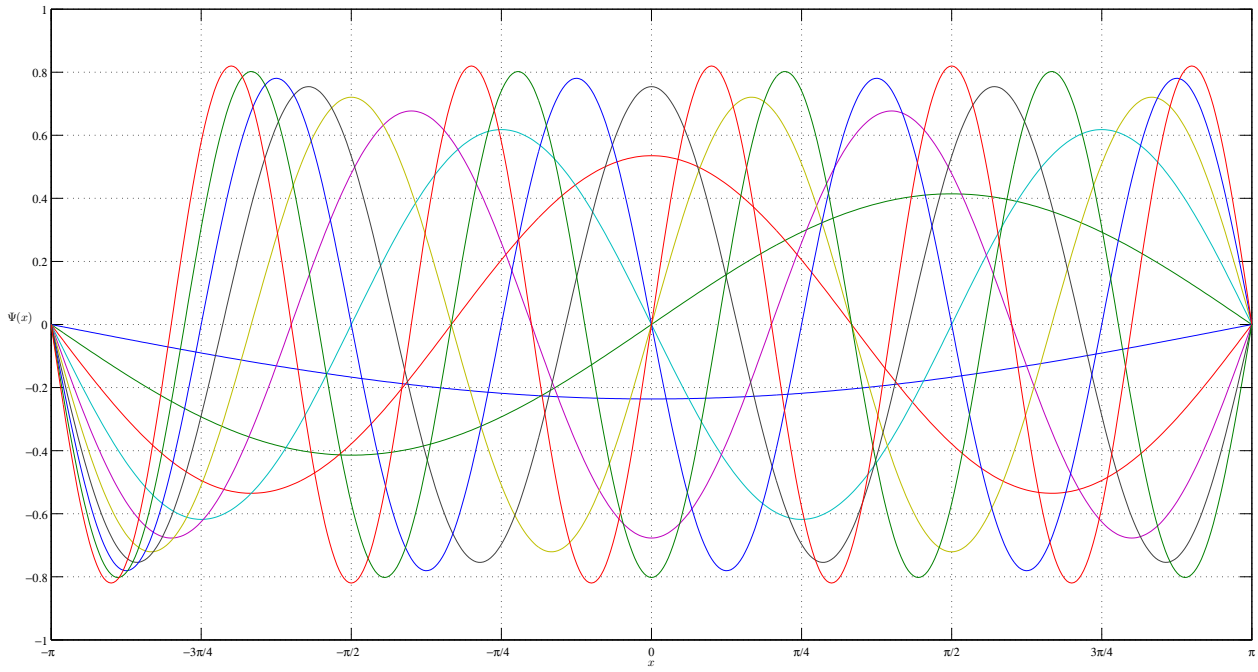

Figura 5.22: Soluciones Dirac - Disparo ($y = 0, z = -1$)

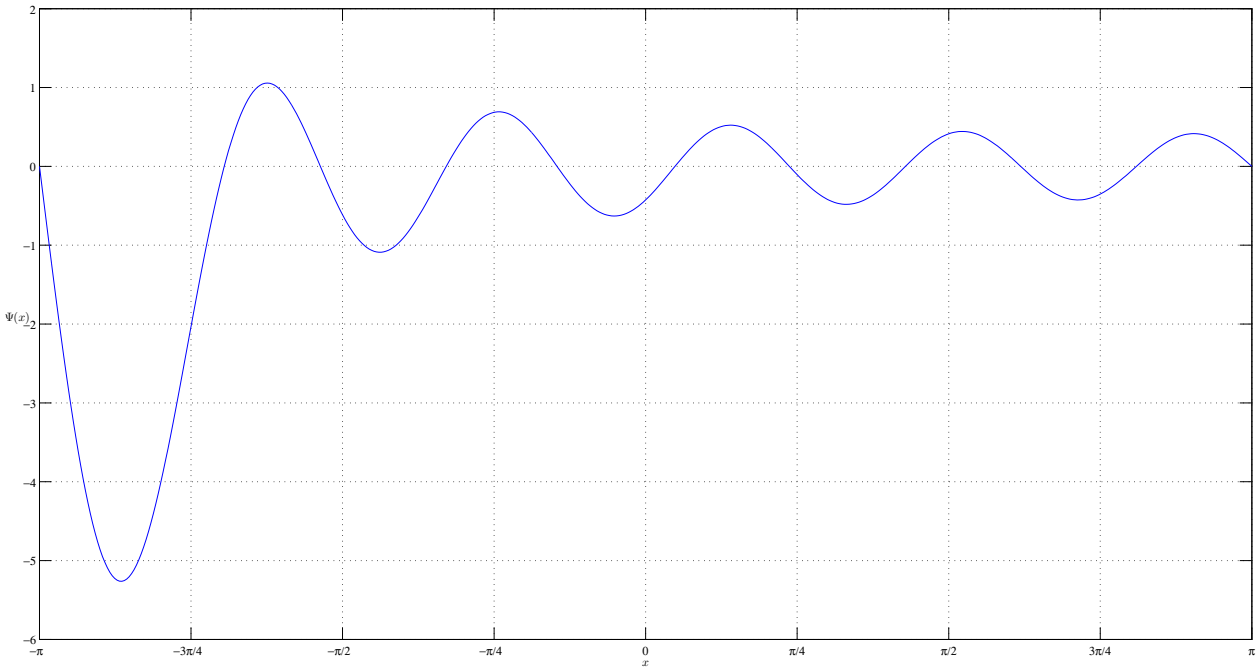

Figura 5.23: Suma Soluciones Dirac - Disparo ($y = 0, z = -1$)

La tabla 5.11 (en la pág. 72) muestra el valor de la energía que se obtuvo E_{disparo}, y el valor de la función Ψ en el ultimo punto Ψ_n, del método de disparo para el problema de Dirac. Otra vez, se respeta la propiedad de entrelazado, pero los valores de Ψ_n se van alejando de cero conforme se buscan valores propios con un índice mayor. Para obtener estos resultados se utilizaron los siguientes parámetros: $E_i = 0, E_f = 20, \Delta E = 0.0001$; $x_i = -\pi, x_f = \pi, \Delta x = 0.012566$; $\epsilon = 0.001$

i	Ψ_n	E_{disparo}
0	3.574388000E-05	1.078700000E+00
1	4.979779000E-05	1.414200000E+00
2	9.861971000E-05	1.802800000E+00
3	7.784952000E-05	2.236050000E+00
4	8.084182000E-05	2.692600000E+00
5	1.318840000E-04	3.162250000E+00
6	-2.442342000E-05	3.640050000E+00
7	-2.238969000E-04	4.123150000E+00
8	-1.160660000E-04	4.609750000E+00
9	-1.575235000E-04	5.099050000E+00
10	-1.109656000E-04	5.590150000E+00
11	7.530727000E-05	6.082750000E+00
12	-1.386229000E-04	6.576450000E+00
13	1.159656000E-04	7.071050000E+00
14	-1.533971000E-04	7.566350000E+00
15	7.970880000E-05	8.062250000E+00
16	1.110008000E-04	8.558650000E+00
17	-1.680925000E-05	9.055400000E+00
18	-1.270847000E-05	9.552500000E+00
19	-2.339856000E-05	1.004990000E+01
20	7.172086000E-05	1.054755000E+01
21	-3.563066000E-05	1.104540000E+01
22	7.289701000E-05	1.154345000E+01
23	-2.692072000E-05	1.204165000E+01
24	8.623182000E-06	1.254000000E+01
25	-1.129107000E-04	1.303850000E+01
26	1.288694000E-04	1.353710000E+01
27	-1.248394000E-04	1.403580000E+01
28	1.600655000E-04	1.453460000E+01
29	8.421215000E-06	1.503345000E+01
30	-4.231477000E-05	1.553240000E+01
31	-9.781499000E-05	1.603145000E+01
32	-1.457496000E-04	1.653050000E+01
33	-1.442867000E-04	1.702970000E+01
34	1.048962000E-04	1.752890000E+01
35	-5.451473000E-05	1.802815000E+01
36	1.273540000E-05	1.852745000E+01
37	3.242446000E-06	1.902680000E+01
38	2.165363000E-05	1.952620000E+01

Tabla 5.11: Solución Dirac ($y = 0, z = 1$) - Disparo paralelo

5.6. Discusión

A lo largo de las pruebas realizadas durante esta tesis observamos que la implementación paralela híbrida obtuvo una mejora en tiempo respecto de MAGMA, sobre todo en la version paralela. Esta mejora en el tiempo de ejecución no solo significa que pueden tratarse problemas con mayor tamaño en un menor tiempo, si no que también, en el caso de la implementación híbrida, es posible pasar la barrera del tamaño de la matriz, impuesta por la cantidad de memoria de la GPU, utilizando varios nodos con múltiples tarjetas y paso de mensajes. Aunque se presentan resultados hasta un tamaño de 18000 nuestra implementación puede potencialmente resolver una matriz de 46000. En cuanto al método de disparo, también se obtuvo una mejora significativa en el tiempo de la version paralela respecto a la secuencial. Sin embargo, no utiliza múltiples nodos, porque para resolver los casos de estudio de esta tesis no se requiere memoria extra, esto no descarta la posibilidad de que existan problemas que si la necesiten.

Una vez obtenido una mejora en los tiempos de ejecución, la siguiente pregunta natural es ¿qué tan buenos son los resultados obtenidos con estos tiempos? Se debe destacar que en el método de diagonalización nuestros resultados presentan una gran calidad en los eigenpares y en la ortogonalidad de los vectores propios. Siendo competitivos con los resultados de las bibliotecas actuales como MAGMA y MKL. Pero es notorio que, en el método de disparo, se debe realizar una investigación mas extensa para obtener mejores resultados, ya que el error se incrementa muy rápidamente conforme aumenta el índice de los valores propios.

También se debe mencionar que los resultados obtenidos muestran que se puede aplicar con éxito la diagonalización heterogénea a casos específicos del problema regular de valores propios de Sturm-Liouville, ya que, se compararon los resultados que obtuvimos con los de la solución analítica, cuando ésta se conocía, obteniendo errores muy pequeños.

Capítulo 6

Conclusiones y trabajo a futuro

En este capítulo, en la sección 6.1 se presentan las principales conclusiones y contribuciones de esta tesis, y en la sección 6.2 (en la pág 76) se discuten algunos aspectos que nos gustaría abordar en una investigación futura.

6.1. Conclusiones

En este trabajo se implementó el algoritmo para diagonalizar matrices simétricas tridiagonales conocido como algoritmo divide y vencerás, de forma paralela híbrida en múltiples nodos, utilizando tanto GPUs como CPUs. Esta implementación resuelve el problema de la restricción del tamaño de la matriz que se puede utilizar con respecto a la cantidad de memoria que posee una GPU. Como mencionamos anteriormente, esta cantidad es muy pequeña en comparación con la cantidad de memoria a la que puede acceder un CPU.

Utilizar el algoritmo de Cuppen en múltiples nodos redujo el tiempo de cómputo de la diagonalización de matrices simétricas tridiagonales con respecto tanto a la implementación secuencial que presentamos en este trabajo, como a la implementación secuencial y paralela de bibliotecas actuales.

Las pruebas realizadas mostraron que las soluciones obtenidas por la implementación paralela heterogénea tienen un alto grado de exactitud con respecto a la ortogonalidad de vectores propios y la calidad de eigenpares.

Para conseguir una buena implementación es importante considerar donde es más apropiado ejecutar cada fase del algoritmo, en el CPU, la GPU, o ambos, teniendo en cuenta el tamaño del problema y la granularidad de la tarea a realizarse. Otro factor relevante es, mantener al mínimo la comunicación entre los nodos y realizar los cálculos sobre estructuras de datos que ocupen la menor cantidad de memoria posible.

Utilizar la implementación paralela heterogénea para resolver problemas regulares de valores propios de Sturm-Liouville es factible, esto fue verificado con un caso de estudio en el que se conoce la solución analítica.

Crear versiones paralelas de las rutinas que calculan el error en las soluciones obtenidas

representa un gran beneficio ya que se puede avanzar en realizar nuevas versiones al detectar problemas de forma más rápida. Las funciones de error que utilizamos en este trabajo son altamente paralelas y nos permitieron calcular los errores hasta diez veces más rápido ahorrándonos días de trabajo.

La version paralela del método de disparo mejoro el tiempo con respecto a su version secuencial y fue aplicada tanto a un problema del que se conocía su solución analítica como a uno del que no. Sin embargo, la calidad de las soluciones obtenidas disminuye conforme se buscan valores propios con un mayor índice.

Las principales contribuciones de esta tesis son:

- Implementación paralela heterogénea (GPUs + CPUs) escalable del algoritmo divide y vencerás para diagonalizar matrices tridiagonales simétricas. La primera de la que tenemos conocimiento.
- Mayor escalabilidad en la implementación del algoritmo divide y vencerás y aumento en la independencia de la cantidad de memoria de la GPU.
- Mejora del tiempo de cómputo de la diagonalización de matrices tridiagonales simétricas.
- Muestra del beneficio de utilizar múltiples nodos tanto con GPUs como con CPUs en el problema de valores propios de matrices tridiagonales simétricas.
- Implementación paralela del método de disparo para el problema del pozo infinito de Schrödinger y Dirac, la cual puede extenderse con pocos cambios a otros casos.
- Mejora del tiempo de cómputo del método de disparo.
- Aplicación exitosa de la implementación paralela heterogénea del algoritmo divide y vencerás a un problema regular de valores propios de Sturm-Liouville. Para el problema del pozo infinito de Schrödinger obtuvimos soluciones muy cercanas a las analíticas.

Se sabe que los solucionadores de valores propios de matrices tridiagonales simétricas son bloques constructores para resolver problemas más generales, tales como, valores propios generales. El enfoque divide y vencerás, además, puede ser aplicado a solucionadores de valores singulares. La importancia de este trabajo recae en la gran cantidad de problemas de valores propios que surgen en aplicaciones científicas y de ingeniería. Por lo tanto, nuestro trabajo potencialmente puede llevar a nuevos descubrimientos científicos.

6.2. Trabajo futuro

Ciertamente, queda mucho trabajo por hacer. Quizá la extension más importante sería tratar matrices densas simétricas. Para lograr esto, primeramente, se debe implementar un algoritmo de tridiagonalización, por ejemplo, el método de Householder, lo ideal sería implementar un version que utilice múltiples nodos. Aunque durante esta tesis se desarrollo el marco principal de trabajo para la diagonaliación de matrices tridiagonales simétricas, se

debe extender la implementación de forma que utilice un número arbitrario de nodos reduciendo al máximo la comunicación entre ellos. Posiblemente, una forma de lograrlo es usar una estructura de árbol binario. Otro punto de interés es utilizar otros métodos para resolver la ecuación secular, con el fin de obtener mejores soluciones y ver que tanto afectan el tiempo total de cómputo. En cuanto al método de disparo, evidentemente, se debe realizar una investigación más extensa para obtener mejores resultados, ya que el error se incrementa muy rápidamente conforme aumenta el índice de los valores propios.

También, como trabajo a futuro se pueden realizar pruebas adicionales con la implementación actual, por ejemplo: resolver otros problemas de Sturm-Liouville que se presentan en el mundo real, hacer comparaciones con otras bibliotecas actuales como CULA, y medir el consumo energético de cada implementación.

En general, se pueden realizar mejoras a las implementaciones desarrolladas en este trabajo utilizando características de CUDA como streams, memoria compartida y memoria unificada. Además, sería particularmente benéfico utilizar el paralelismo dinámico de las nuevas versiones de CUDA en el método de disparo. Las tecnologías que más nos permitirían reducir los tiempos de cómputo son aquellas que transfieren datos directamente de una GPU a otra, como son, MVAPICH2-GDR 2.0 y OpenMPI 1.8.3, que utilizan GPUDirect.

Bibliografía

[1] Werner O. Amrein, Andreas M. Hinz, and David P. Pearson. *Sturm-Liouville theory: past and present*. Springer, Germany, 1st edition, 2005.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999.

[3] Innovative Computing Laboratory (ICL) Team. Matrix Algebra on GPU and Multicore Architectures (MAGMA) library, version 1.5.0. Web site: `http://icl.cs.utk.edu/magma/` [Last accessed: 31 November 2014].

[4] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2nd edition, 1992.

[5] Christof Vömel, Stanimire Tomov, and Jack Dongarra. Divide and conquer on hybrid GPU-accelerated multicore systems. *SIAM Journal on Scientific Computing*, 34(2):70–82, April 2012.

[6] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1st edition, 1997.

[7] Ren-Cang Li. Solving secular equations stably and efficiently. Technical Report UCB/CSD-94-851, EECS Department, University of California, Berkeley, Berkeley, CA, USA, Dec 1994.

[8] Ming Gu and Stanley C. Eisenstat. A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem. *SIAM Journal on Matrix Analysis and Applications*, 15(4):1266–1276, October 1994.

[9] J.J.M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik*, 36(2):177–195, March 1980.

[10] Azzam Haidar, Mark Gates, Stan Tomov, and Jack Dongarra. Toward a scalable multi-gpu eigensolver via compute-intensive kernels and efficient communication. In Allen D Malony, Mario Nemirovsky, and Sam Midkiff, editors, *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 223–232, Eugene, Oregon, USA, 10–12 June 2013. ACM.

[11] Boris Moiseevich Levitan and Išchan S Sargsjan. *Sturm—Liouville and Dirac Operators*. Springer, 1991.

[12] Ravi P. Agarwal and Donal O'Regan. *Ordinary and partial differential equations: with special functions, Fourier series, and boundary value problems*. Springer, USA, 1st edition, 2009.

[13] Mohammed Al-Gwaiz. *Sturm-Liouville theory and its applications*. Springer, 2007.

[14] Erwin Schrödinger. Quantisierung als eigenwertproblem. *Annalen der physik*, 385(13):437–490, 1926.

[15] Amilcar Meneses Viveros. *Simulación de paquetes de onda y teoría de Weyl*. PhD thesis, Computer Science Department, CINVESTAV-IPN, 2009.

[16] Paul AM Dirac. The quantum theory of the electron. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, pages 610–624, 1928.

[17] B. Kolman, D.R. Hill, and V.H.I. Mercado. *Introductory Linear Algebra*. Pearson Education, 2008.

[18] J.H. Wilkinson. *The Algebraic Eigenvalue Problem*. Monographs on numerical analysis. Clarendon Press, 1988.

[19] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU Press, 4th edition, 2012.

[20] Susan Morgan. Making sense of parallel programming terms. In Solaris Studio documentation. Web site: `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/parallel-terminology-362819.html` [Last accessed: 31 November 2014].

[21] Blaise Barney. Introduction to Parallel Computing (Tutorial). Web site: `https://computing.llnl.gov/tutorials/parallel_comp` [Last accessed: 31 November 2014].

[22] AMD. What is Heterogeneous Computing? Web site: `http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-computing/` [Last accessed: 31 November 2014].

[23] Nvidia. What is CUDA? Web site: `http://www.nvidia.com/object/cuda_home_new.html` [Last accessed: 31 November 2014].

[24] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2013.

[25] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[26] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[27] Nicholas Wilt. *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education, 2013.

[28] OpenMP Architecture Review Board. What is OpenMP? Web site: `http://openmp.org/openmp-faq.html` [Last accessed: 31 November 2014].

[29] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008.

[30] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0. Web site: `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf` [Last accessed: 31 November 2014].

[31] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. 1999.

[32] Peter Arbenz. Chapter 4 The Cuppen's divide and conquer algorithm. In Lecture notes on solving large scale eigenvalue problems. Web site: `http://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter4.pdf` [Last accessed: 31 November 2014].

[33] StataCorp. Permutation. In Stata 13 Base Reference Manual. Web site: `http://www.stata.com/manuals13/m-1permutation.pdf` [Last accessed: 31 November 2014].