Departamento de Sistemas Informáticos y Computación

UNIVERSIDAD
POLITECNICA
DE VALENCIA

Implementación paralela de métodos de Krylov con reinicio para problemas de valores propios y singulares

Tesis Doctoral
Requisito para la obtención del grado de doctor en informática por la Universidad Politécnica de Valencia

Valencia, 25 de Marzo de 2009

Autor: Andrés Enrique Tomás Domínguez
Directores: Dr. D. José E. Román Moltó
Dr. D. Vicente Hernández García

Resumen

Esta tesis aborda la paralelización de los métodos de Krylov con reinicio para problemas de valores propios y valores singulares (SVD). Estos métodos son de naturaleza iterativa y resultan adecuados para encontrar unos pocos valores propios o singulares de problemas dispersos. El procedimiento de ortogonalización suele ser la parte más costosa de este tipo de métodos, por lo que ha recibido especial atención en esta tesis, proponiendo y validando nuevos algoritmos para mejorar sus prestaciones paralelas.

La implementación se ha realizado en el marco de la librería SLEPc, que proporciona una interfaz orientada a objetos para la resolución iterativa de problemas de valores propios o singulares. SLEPc está basada en la librería PETSc, que dispone de implementaciones paralelas de métodos iterativos para la resolución de sistemas lineales, precondicionadores, matrices dispersas y vectores. Ambas librerías están optimizadas para su ejecución en máquinas paralelas de memoria distribuida y con problemas dispersos de gran dimensión.

Esta implementación incorpora los métodos para valores propios de Arnoldi con reinicio explícito, de Lanczos (incluyendo variantes semiortogonales) con reinicio explícito, y versiones de Krylov-Schur (equivalente al reinicio implícito) para problemas no Hermitianos y Hermitianos (Lanczos con reinicio grueso). Estos métodos comparten una interfaz común, permitiendo su comparación de forma sencilla, característica que no está disponible en otras implementaciones. Las mismas técnicas utilizadas para problemas de valores propios se han adaptado a los métodos de Golub-Kahan-Lanczos con reinicio explícito y grueso para problemas de valores singulares, de los que no existe ninguna otra implementación paralela con paso de mensajes.

Cada uno de los métodos se ha validado mediante una batería de pruebas con matrices procedentes de aplicaciones reales. Las prestaciones paralelas se han medido en máquinas tipo cluster, comprobando una buena escalabilidad incluso con un número muy grande de procesadores, y obteniendo unas prestaciones competitivas con respecto del estado del arte en este tipo de software.

Títol

Implementació paral·lela de mètodes de Krylov amb reinici per a problemes de valors propis i singulars

Resum

Aquesta tesi aborda la paral·lelització dels mètodes de Krylov amb reinici per a problemes de valors propis i valors singulars (SVD). Aquests mètodes són de naturalesa iterativa i resulten adequats per a trobar uns pocs valors propis o singulars de problemes dispersos. El procediment d'ortogonalització sol ser la part més costosa d'aquest tipus de mètodes, per la qual cosa ha rebut especial atenció en aquesta tesi, proposant i validant nous algorismes per a millorar les seues prestacions paral·leles.

La implementació s'ha realitzat en el marc de la llibreria SLEPc, que proporciona una interfície orientada a objectes per a la resolució iterativa de problemes de valors propis o singulars. SLEPc està basada en la llibreria PETSc, que disposa d'implementacions paral·leles de mètodes iteratius per a la resolució de sistemes lineals, precondicionadors, matrius disperses i vectors. Ambdues llibreries estan optimitzades per a la seua execució en màquines paral·leles de memòria distribuïda i amb problemes dispersos de gran dimensió.

Aquesta implementació incorpora els mètodes per a valors propis d'Arnoldi amb reinici explícit, de Lanczos (incloent variants semiortogonals) amb reinici explícit, i versions de Krylov-Schur (equivalent al reinici implícit) per a problemes no Hermitians i Hermitians (Lanczos amb reinici gruixut). Aquests mètodes comparteixen una interfície comuna, permetent la seua comparació de forma senzilla, característica que no està disponible en altres implementacions. Les mateixes tècniques utilitzades per a problemes de valors propis s'han adaptat als mètodes de Golub-Kahan-Lanczos amb reinici explícit i gruixut per a problemes de valors singulars, dels quals no existeix cap altra implementació paral·lela amb pas de missatges.

Cadascun dels mètodes s'ha validat mitjançant una bateria de proves amb matrius procedents d'aplicacions reals. Les prestacions paral·leles s'han mesurat en màquines tipus cluster, comprovant una bona escalabilitat fins i tot amb un nombre molt gran de processadors, i obtenint unes prestacions competitives respecte a l'estat de l'art en aquest tipus de programari.

Title

Parallel implementation of restarted Krylov methods for eigenvalue and singular value problems.

Summary

This thesis addresses the parallelization of restarted Krylov methods for eigenproblems and singular value (SVD) problems. These methods are iterative in nature and are appropriate for finding a few eigenvalues or singular values from sparse problems. In this kind of methods, the part with the higher cost is the ortogonalization procedure. Therefore this procedure has received special attention in this thesis. New algorithms have been proposed and validated in order to improve its parallel performance.

The implementation has been made in the framework provided by the SLEPc library. This library provides an object oriented interface for iterative eigensolvers or SVD solvers. SLEPc is based on PETSc, which provides parallel implementations of lineal solvers, preconditioners, sparse matrices and vectors. Both libraries are optimized for distributed memory parallel computers with very large sparse matrices.

This implementation includes the following eigenvalue methods: explicitly restarted Arnoldi, explicitly restarted Lanczos (with semiorthogonal variants) and Krylov-Schur (equivalent to implicit restart) for non-Hermitian and Hermitian (also known as thick restart Lanczos) matrices. These methods share a common interface, which enables an easy comparison between them, feature that is not available in any other implementation. The same techniques employed for eigenproblems have been adapted to the SVD Golub-Kahan-Lanczos methods with explicit and thick restart, which have no previous implementation with message-passing parallelism.

Each one of the methods has been validated through a battery of tests with matrices from a variety of applications. Parallel performance has been measured in cluster computers, showing good scalability even with a large number of processors, and obtaining competitive performance with current state of the art software.

Agradecimientos

En primer lugar, quisiera agradecer al Grupo de Grid y Computación de Altas Prestaciones (GRyCAP) su apoyo para la realización del programa de doctorado, y en particular a los directores de esta tesis, José Román y Vicente Hernández. Sin ellos este trabajo no hubiera sido posible.

También quiero agradecer a las instituciones que han proporcionado recursos para esta tesis, bien de forma económica como la Generalitat Valenciana, bien mediante el acceso a tiempo de cómputo. En concreto, el autor agradece los recursos y asistencia proporcionados por el Centro Nacional de Supercomputación (BSC) y por el National Energy Research Scientific Computing Center, financiado por la Office of Science of the U.S. Department of Energy con el contrato número DE-AC03-76SF00098.

Y por último, pero no por ello menos importante, quiero agradecer a Sonia su infinita compresión y paciencia durante el largo proceso de escritura de esta tesis. *And now for something completely different* ...

Índice

Índice de algoritmos

Introducción

El problema algebraico de valores propios, también conocidos como autovalores, surge de forma natural en muchos campos de la ciencia y la ingeniería al analizar el comportamiento dinámico de sistemas, entre otras muchas aplicaciones. La descomposición en valores singulares (SVD) es un problema relacionado con el de valores propios que permite obtener importante información acerca de una matriz. Históricamente, los métodos para resolver numéricamente estos problemas han sido demasiado costosos y solamente son viables gracias a los ordenadores actuales, aunque su implementación no es trivial.

Una parte de los problemas de valores propios y singulares tiene naturaleza dispersa, es decir, la mayoría de los elementos de la matriz son nulos, como por ejemplo ocurre con la discretización de ecuaciones en derivadas parciales. En estos problemas es mucho más eficiente utilizar algoritmos que no modifican los elementos nulos, lo que permite ahorrar memoria y abordar matrices de dimensiones mucho mayores. Por regla general en estos casos solamente se necesita calcular una pequeña parte de los valores propios o singulares que cumplen alguna condición conocida de antemano. Para este tipo de problemas los métodos de Krylov resultan muy interesantes, porque mantienen la estructura dispersa de la matriz original y permiten calcular de forma iterativa únicamente los valores requeridos. Esta convergencia se puede adaptar a las necesidades de un problema concreto mediante transformaciones espectrales o técnicas de reinicio.

Escenario

El usuario típico que necesita resolver un problema de valores propios o singulares es un experto en algún campo ajeno a la informática, pero con conocimientos básicos de programación. Este usuario desarrolla sus propios programas a medida para la resolución de un problema concreto. Gracias a la potencia actual de los ordenadores personales, estos desarrollos se suelen realizar en entornos dedicados que incorporan editor, intérprete, depurador y soporte para entrada/salida y visualización de resultados. El producto comercial MATLAB es el estándar de facto para estos entornos de desarrollo, aunque existen otros similares con algunas mejoras para aplicaciones específicas. Todos estos entornos tienen en común proporcionar un lenguaje de alto nivel, con potentes primitivas para el manejo de matrices y vectores, de forma que resulta bastante sencillo implementar los métodos numéricos necesarios para la resolución del problema. El principal inconveniente de este enfoque es que tiene unas prestaciones limitadas.

La limitación de prestaciones de los entornos tipo MATLAB viene dada tanto por el propio entorno interactivo como por la capacidad de cálculo y memoria de un único procesador. Para poder tratar problemas más grandes es necesario trabajar con lenguajes compilados y computadores paralelos. Tradicionalmente, el lenguaje más utilizado para computación numérica es el Fortran que proporciona soporte directo para matrices y aritmética compleja. Este lenguaje ha evolucionado mucho desde sus orígenes, aunque los compiladores actuales no implementan las últimas características añadidas al estándar como la programación orientada a objetos. Debido a la evolución histórica del Fortran, ciertas características avanzadas resultan más complicadas de utilizar que en otros lenguajes de propósito general, como, por ejemplo, la gestión de memoria dinámica y el soporte para estructuras de datos definidas por el usuario.

La necesidad de paralelizar un algoritmo surge cuando las prestaciones en un solo procesador no son satisfactorias, por lo que se suele disponer de una implementación secuencial de partida. Dado que no existe ningún método automático general para convertir eficientemente esta implementación en una paralela equivalente, es necesario codificar a mano la paralelización, llegando incluso a cambiar totalmente el algoritmo. Tradicionalmente las máquinas paralelas se han clasificado en dos tipos según si los procesadores comparten o no la memoria, lo que determina el paradigma de programación.

Las máquinas de memoria compartida se suelen programar mediante extensiones a lenguajes secuenciales como OpenMP. Este sistema permite partir de

un programa secuencial e ir añadiendo poco a poco primitivas paralelas para acelerar las partes más lentas. El inconveniente de estas máquinas es que tienen limitado el número máximo de procesadores, debido a la elevada complejidad y el cuello de botella que representa la memoria compartida. Por el contrario, las máquinas de memoria distribuida no tienen esta limitación y resultan más económicas, gracias a que aprovechan componentes de ordenadores convencionales.

La programación de las máquinas de memoria distribuida se suele realizar mediante lenguajes secuenciales, con librerías como MPI que implementan el paso de mensajes entre procesadores. El coste temporal de estos mensajes es elevado, y por lo tanto la distribución de datos entre procesadores es crítica para reducir su número y tamaño. El principal inconveniente de estas máquinas es que obligan a diseñar el algoritmo desde el principio teniendo en cuenta el paralelismo, por lo que generalmente no es fácil partir de una versión secuencial. Una ventaja del modelo de paso de mensajes es que puede aplicarse también a máquinas de memoria compartida, aunque no es la solución óptima.

En principio, es el propio programador el responsable de implementar los métodos numéricos necesarios para la resolución de su problema. A medida que estos métodos resultan cada vez más complejos, es necesario recurrir a implementaciones disponibles en paquetes software. Estos paquetes están realizados por expertos en computación numérica y suelen tener la forma de librerías de funciones, tanto para entornos de alto nivel tipo MATLAB como para lenguajes más generales como Fortran y C.

El modelo de abstracción utilizado en una librería de funciones encapsula únicamente la implementación de los métodos numéricos, pero deja al descubierto las estructuras de datos utilizadas. Esto funciona bien con las típicas estructuras de datos simples, pero no es suficiente con otras estructuras más complejas, como las necesarias para matrices dispersas o para el reparto eficiente entre varios procesadores. La programación orientada a objeto permite proporcionar una interfaz sencilla, que oculta al usuario la complejidad de la programación paralela y a la vez proporciona primitivas basadas en matrices y vectores. Esta interfaz efectivamente reduce el salto semántico desde la notación matemática o desde un lenguaje tipo MATLAB hasta una implementación paralela eficiente.

En la actualidad solamente están disponibles dos librerías paralelas que implementan métodos para problemas dispersos de valores propios no Hermitianos. La más antigua, ARPACK, implementa el método de Arnoldi con reinicio implícito y es muy eficiente. Pero su interfaz es de bajo nivel y no resulta sencilla

de utilizar, obligando al usuario a implementar en paralelo el producto matriz vector. Además, recientemente se ha propuesto el método de Krylov-Schur como alternativa más estable y fácil de implementar. Por otro lado, Anasazi es una moderna librería de clases C++ que dispone de este método, pero su implementación no resulta tan eficiente como ARPACK, y aunque dispone de una interfaz orientada a objetos, resulta complicada de utilizar por parte del usuario típico que no suele disponer de los conocimientos avanzados de programación necesarios. Para los problemas dispersos de valores singulares no existe ninguna implementación paralela para máquinas de memoria distribuida.

Objetivos de la tesis

El objetivo general de esta tesis es la implementación paralela de métodos de Krylov con técnicas de reinicio para valores propios y singulares en la librería SLEPc. Antes de los trabajos que condujeron a esta tesis, esta librería proporcionaba una interfaz orientado a objetos común a diferentes métodos básicos para problemas dispersos de valores propios. Los métodos más avanzados se podían acceder con esta misma interfaz pero estaban implementados en otras librerías.

La librería SLEPc está basada en PETSc, que proporciona implementaciones paralelas de métodos iterativos para la resolución de sistemas lineales, precondicionadores, matrices dispersas y vectores. Para garantizar la mejor portabilidad y compatibilidad posible, esta implementación se realiza en lenguaje C estándar utilizando la librería MPI para la comunicación entre procesadores. Los detalles de la paralelización de los algoritmos y la distribución de datos se ocultan al usuario mediante una interfaz orientado a objetos. Esta interfaz está diseñada para proporcionar primitivas de alto nivel y para ser utilizada desde los lenguajes C y Fortran. De esta forma se permite al usuario típico, con conocimientos básicos de programación, acceder a las características avanzadas de la librería.

En concreto, los objetivos específicos de esta tesis son:

- Implementación paralela con paso de mensajes de los métodos de Arnoldi y Lanczos.
- Estudio de la eficiencia de los procedimientos de ortogonalización aplicables a los métodos de Krylov.
- Comparación de las distintas estrategias de reortogonalización para el método de Lanczos.

- Soporte para problemas generalizados en el método de Lanczos mediante transformaciones espectrales.
- Aplicación del método de Lanczos para el cálculo de valores singulares.
- Utilización de técnicas de reinicio para mejorar o modificar la convergencia de los métodos.
- Validación numérica de los métodos implementados mediante una batería de pruebas.
- Medida experimental de la calidad de la implementación paralela comparándola con el estado actual del software para cálculo de valores propios.

Estructura de la memoria

Esta tesis está estructurada para que cada capítulo se base en el material presentado en los anteriores. Primero, el capítulo 1 introduce de forma breve los métodos para valores propios y singulares, y revisa el software disponible con especial atención a los problemas dispersos y a las implementaciones paralelas. El capítulo 2 presenta la librería SLEPc, su modelo de paralelismo y diseño de la interfaz de usuario. El capítulo 3 estudia los métodos de Krylov enfocados a su implementación en paralelo. El capítulo 4 detalla esta implementación realizada en SLEPc junto con medidas de prestaciones. Por último, el capítulo 5 presenta las conclusiones de esta tesis.

Notación y definiciones

La notación matemática utilizada en esta tesis es la típica utilizada en la literatura. Las letras latinas mayúsculas y minúsculas se utilizan para matrices y vectores respectivamente, mientras que las letras minúsculas griegas denotan escalares. La notación a_j se refiere a la columna j de una matriz A y la matriz A_n es la formada por n vectores columna $a_1, a_2, \ldots, a_n$. La letra I se reserva para la matriz identidad y e_j para sus columnas. La submatriz $A_{r:s,p:q}$ está formada por los elementos comprendidos entre las filas r y s y las columnas p y q, mientras que $A_{p:q}$ está formada por las columnas p y q completas. Una matriz diagonal de tamaño $n \times n$ se denota con $\mathrm{diag}(\alpha_1, \alpha_2, \ldots, \alpha_n)$ donde $\alpha_1, \alpha_2, \ldots, \alpha_n$ son los elementos ordenados de la diagonal principal.

La notación A^{-1} se utiliza para la inversa de A, A^T para la transpuesta y A^* para la transpuesta conjugada. Una matriz es simétrica si $A = A^T$ y Hermitiana si $A = A^*$. Una matriz es unitaria si $A^* = A^{-1}$, aunque el caso particular de las matrices reales $A^T = A^{-1}$ se suele denominar ortogonal.

El producto interior definido por la matriz B se denota $x^* By$, y en el caso del producto interior euclídeo (o producto escalar) se simplifica a $x^* y$. La norma euclídea para vectores se define como $\|x\|_2 = \sqrt{x^* x}$. La norma de Frobenius para matrices se define como

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}\,.$$

La notación $\langle v_1, v_2, \ldots, v_n \rangle$ hace referencia al subespacio generado por estos vectores. Un conjunto de vectores $v_1, v_2, \ldots, v_n$ son ortogonales si $v_i^* v_j = 0$ para todo $i \neq j$, si además todos los vectores tiene norma unidad se dice que son ortonormales.

Capítulo 1

Problemas de valores propios y singulares

La primera parte de este capítulo repasa de forma breve las características de los problemas de valores propios y singulares, así como los métodos más significativos para su resolución. La descripción de los métodos de Krylov se deja para el capítulo 3. En la segunda parte se revisa el estado actual del software que implementa estos métodos, con especial énfasis en los problemas dispersos. Para profundizar en los temas tratados existe una excelente bibliografía, desde el clásico [Wil65] pasando por obras más generales como [GVL96, Dem97] y material especializado como [Par80, Saa92, BDD+00, Ste01b]. Además, la descripción del software dispone de referencias específicas para cada implementación.

1.1. Problema de valores propios

El problema estándar de valores propios consiste en encontrar los valores $\lambda \in \mathbb{C}$ para los que el sistema lineal

$$Ax = \lambda x \tag{1.1}$$

tiene solución no trivial, siendo $A \in \mathbb{C}^{n\times n}$ y $x \in \mathbb{C}^n$. Esto ocurre solamente si

$$\det(A - \lambda I) = 0\,. \tag{1.2}$$

Expandiendo este determinante se obtiene un polinomio en λ de grado n, denominado polinomio característico, por lo que existen exactamente n valores propios. Sin embargo, este polinomio no se puede utilizar para resolver el problema puesto que el cálculo de sus raíces está muy mal condicionado.

Al vector x_i que es solución de $Ax_i = \lambda_i x_i$ se le denomina vector propio asociado al valor propio λ_i. El conjunto de todos los valores propios se denomina espectro de A y se denota por

$$\lambda(A) = \{\lambda_1, \lambda_2, \ldots, \lambda_n\} \ . \tag{1.3}$$

Los valores propios de una matriz Hermitiana son reales. Si además la matriz es real, entonces todos los vectores propios son reales. Los valores propios complejos de una matriz real siempre aparecen como pares complejos conjugados. Además sus vectores propios complejos solamente pueden aparecer asociados a un valor propio complejo y por lo tanto también son conjugados.

En algunas aplicaciones aparece el problema de valores propios generalizado

$$Ax = \lambda Bx \ , \tag{1.4}$$

donde $B \in \mathbb{C}^{n\times n}$ y el resto de elementos se definen como en el problema estándar (1.1). Si A y B son Hermitianas y además B es definida positiva,

$$x^* Bx > 0 \ \ \forall x \neq 0 \ , \tag{1.5}$$

el problema se denomina generalizado Hermitiano definido positivo. En esta tesis esta denominación se abreviará a generalizado Hermitiano cuando no se preste a confusión. Este caso resulta muy interesante porque se puede transformar el problema a uno estándar Hermitiano. Si B cumple la condición anterior pero con la desigualdad no estricta, se dice que es semidefinida positiva. En este caso el problema generalizado puede tratarse de la misma forma pero teniendo en cuenta el detalle de que los valores propios correspondientes a los valores nulos de B pueden ser infinitos.

1.1.1. Métodos basados en transformaciones

Dada una matriz X no singular, la transformación de semejanza construye otra matriz $B = XAX^{-1}$ con los mismos valores propios que A. Los vectores propios de B son de la forma Xx, donde x es un vector propio de A. Aplicando estas transformaciones de semejanza es posible reducir la matriz original a una forma canónica, donde es fácil el cálculo de los valores y vectores propios.

Algoritmo 1.1 Iteración QR

para $j = 1, 2, \ldots$
Calcular la descomposición $QR = A$
$A = RQ$
fin

La forma más simple es una diagonal con los valores propios, pero no está garantizada su construcción en todos los casos. La forma de Jordan es una matriz bidiagonal superior con los valores propios en la diagonal y unos o ceros en la superdiagonal. Siempre es posible su construcción pero los algoritmos disponibles no son numéricamente estables. La forma de Schur es la más práctica de todas y se puede obtener en todos los casos de forma estable mediante transformaciones unitarias.

Se denomina descomposición de Schur a

$$A = UTU^* , \tag{1.6}$$

donde U es una matriz unitaria y T es una matriz triangular superior cuyos elementos diagonales son los valores propios de A. Las columnas de U se denominan vectores de Schur. Si s es un vector propio de T entonces Us es un vector propio de A. Cuando A es real, pero con valores propios complejos, es necesario almacenar el par conjugado en un bloque diagonal de dos elementos y entonces decimos que T tiene forma real de Schur. Esta forma es muy interesante para la implementación en un ordenador, porque permite trabajar con valores propios complejos utilizando únicamente aritmética real.

La iteración QR (algoritmo 1.1) obtiene la descomposición de Schur de forma iterativa. Este algoritmo tiene dos pasos, primero se obtiene una matriz Q unitaria y otra R triangular superior mediante la descomposición QR y después se sustituye la matriz A por el producto RQ. A medida que el algoritmo itera esta sucesión de matrices se acerca progresivamente a la forma de Schur. La convergencia del método original es bastante lenta y para mejorarla se aplican varias técnicas: reducción inicial a forma de Hessenberg, desplazamiento implícito, deflación y balanceado de la matriz.

Se denomina descomposición de Hessenberg a

$$A = UHU^* , \tag{1.7}$$

donde U es una matriz que acumula varias transformaciones unitarias y H tiene forma de Hessenberg superior, es decir, todos los elementos por debajo de la primera subdiagonal son nulos. Esta forma reducida se utiliza como primer paso para obtener la forma de Schur porque es fácil de calcular en un número finito de pasos con un algoritmo estable.

El equivalente a la descomposición de Hessenberg para matrices Hermitianas es la reducción a una tridiagonal simétrica. Para obtener sus valores y vectores propios se puede utilizar la iteración QR, pero existen otros algoritmos específicos para este caso:

Divide y vencerás es un algoritmo recursivo que divide la matriz tridiagonal en dos partes, resuelve el problema de cada parte, y une las dos soluciones [Cup80].

Bisección permite calcular los valores propios dentro de un intervalo, y luego utiliza la iteración inversa (explicada en la sección siguiente) para obtener los vectores propios. Puede ser mucho más rápido para intervalos pequeños, o para todo el espectro, si los valores propios están bien separados [Kah66].

Representaciones relativamente robustas (RRR) se basa en factorizaciones con L triangular y D diagonal del tipo $LDL^T = T - \sigma I$ de la tridiagonal T desplazada con σ cercano a grupos de valores propios [Dhi98]. Es el que menos operaciones realiza de todos, salvo en algunos casos particulares [DMPV08].

Para resolver el problema generalizado (ecuación 1.4) el método QZ funciona de la misma forma que la iteración QR, transformando la matriz A a forma de Hessenberg y B a triangular superior. Para el problema generalizado Hermitiano no se suelen utilizar métodos específicos, sino que se utiliza la descomposición de Cholesky $B = LL^*$, con L triangular superior, para transformar el problema en $L^{-1}AL^{-T}$ estándar Hermitiano y aplicar los métodos descritos anteriormente.

1.1.2. Método de la potencia

Los métodos anteriores no son los más adecuados para problemas dispersos, donde la mayoría de los elementos de la matriz son nulos, puesto que las transformaciones de semejanza no mantienen la estructura de la matriz original. Si solamente se desean unos pocos valores propios son más interesantes los

Algoritmo 1.2 Método de la potencia

Dado un vector inicial y
repetir
$v = y/\|y\|_2$
$y = Av$
$\theta = v^* y$
hasta $\|y - \theta v\|_2 \leq \text{tol}$

Algoritmo 1.3 Método de la iteración inversa

Dado un vector inicial y
repetir
$v = y/\|y\|_2$
$y = (A - \sigma I)^{-1} v$
$\theta = v^* y$
hasta $\|y - \theta v\|_2 \leq \text{tol}$

métodos basados en el producto matriz vector. Este producto se puede implementar eficientemente almacenando únicamente los elementos no nulos, con el consiguiente ahorro de memoria y operaciones aritméticas.

El método de la potencia (algoritmo 1.2) obtiene el valor propio dominante (de mayor módulo) θ y su vector propio asociado v. Este método está basado en el cociente de Rayleigh que permite obtener el valor propio λ a partir de su vector propio asociado x

$$\lambda = \frac{x^* A x}{x^* x} \,. \tag{1.8}$$

La convergencia de este método puede ser muy lenta y depende de la separación entre los dos valores propios de mayor módulo.

El método de la iteración inversa (algoritmo 1.3) se construye sustituyendo en el método de la potencia la matriz A por $(A-\sigma I)^{-1}$. En la práctica no se calcula explícitamente esta matriz sino que se resuelve un sistema lineal distinto en cada iteración tanto con métodos directos como iterativos. El valor propio se obtiene con $\lambda = \sigma + 1/\theta$ y converge rápidamente hacia el valor propio más cercano al

desplazamiento σ, lo que permite calcular cualquier parte del espectro. Además, la velocidad de convergencia depende de las separaciones entre σ y los dos valores propios más cercanos a σ. Este método es particularmente efectivo si se dispone de una buena aproximación al valor propio y se desea obtener el vector propio.

Se puede acelerar la convergencia de la iteración inversa usando la aproximación al valor propio como desplazamiento en la iteración siguiente. Este método se conoce como iteración del cociente de Rayleigh (RQI) y converge muy rápidamente. Sin embargo, no se utiliza en la práctica porque el coste de factorizar o calcular el precondicionador para una $A - \sigma I$ distinta en cada iteración es demasiado elevado. Además, conforme el algoritmo converge a un valor propio el condicionamiento del sistema lineal empeora en cada iteración, haciendo imposible su resolución con un método iterativo y necesitando consideraciones especiales con un método directo [Ips97].

El concepto de la iteración inversa puede extenderse a otros métodos, lo que se conoce como transformación espectral. Estas transformaciones consisten en sustituir la matriz original por otra con los mismos vectores propios pero con los valores propios recolocados. Eligiendo apropiadamente esta recolocación se puede calcular rápidamente valores propios a los que el algoritmo original tardaría mucho en converger. Otra función de las transformaciones espectrales es convertir un problema generalizado en uno estándar.

Una de las transformaciones espectrales más utilizadas es la de desplazamiento e inversión que convierte el problema generalizado $Ax = \lambda Bx$ en otro

$$(A - \sigma B)^{-1} Bx = \theta x \,. \tag{1.9}$$

Los valores propios originales se obtienen como en la iteración inversa $\lambda = \sigma + 1/\theta$ con una rápida convergencia a los valores cercanos a σ. Otra transformación espectral con propiedades interesantes es la de Cayley [MSR94].

1.1.3. Iteración del subespacio

El método de la iteración del subespacio (algoritmo 1.4) es una extensión con más vectores del método de la potencia. En este algoritmo los valores propios de la matriz H convergen a los valores propios dominantes de la matriz A. La descomposición QR es un paso análogo a la normalización en el método de la potencia y garantiza la independencia lineal entre los vectores V. Existen varias técnicas para reducir el coste de este algoritmo. En primer lugar, se pueden acumular varios productos por A antes de hacer la descomposición QR. Segundo,

Algoritmo 1.4 Método de la iteración del subespacio

Dada una matriz inicial Y
repetir
  Calcular la descomposición $VR = Y$
  $Y = AV$
  $H = V^*Y$
hasta $\|Y - VH\|_2 \leq \text{tol}$

Algoritmo 1.5 Procedimiento de Rayleigh-Ritz

1. Construir una base ortonormal $\{v_1, v_2, \ldots, v_m\}$ del subespacio $\mathscr{K}$ y sea V la matriz con columnas $v_1, v_2, \ldots, v_m$
2. Calcular $B_m = V^*AV$
3. Calcular los valores propios de B_m
4. Seleccionar los $k < m$ valores buscados $\tilde{\lambda}_1, \tilde{\lambda}_2, \ldots, \tilde{\lambda}_k$
5. Calcular los vectores propios y_i de B_m asociados a cada $\tilde{\lambda}_i$ y los vectores propios aproximados $\tilde{x}_i = Vy_i$ de A

no es necesario hacer el producto por A con los valores y vectores convergidos en iteraciones anteriores. Y por último se puede utilizar el proceso de Rayleigh-Ritz para obtener mejores aproximaciones a los valores propios.

El procedimiento de Rayleigh-Ritz (algoritmo 1.5) aproxima los valores propios mediante una proyección sobre un subespacio $\mathscr{K}$ de tamaño m más pequeño que la dimensión n del problema original. Las aproximaciones a cada par propio $\tilde{\lambda}$ y $\tilde{x}$ cumplen la condición de Galerkin:

$$A\tilde{x} - \tilde{\lambda}\tilde{x} \perp \mathscr{K} \ , \tag{1.10}$$

o de forma equivalente,

$$(A\tilde{x} - \tilde{\lambda}\tilde{x})^T v = 0, \ \forall v \in \mathscr{K} \ . \tag{1.11}$$

Tomando V como la matriz con columnas $\{v_1, v_2, \ldots, v_m\}$ y

$$\tilde{x} = Vy \ , \tag{1.12}$$

la ecuación 1.11 se convierte en

$$(AVy - \tilde{\lambda} Vy)^T v = 0, \ \forall v \in \mathscr{K} \ . \tag{1.13}$$

Por lo tanto, y y $\tilde{\lambda}$ deben cumplir

$$B_m y = \tilde{\lambda} y \tag{1.14}$$

con

$$B_m = V^T AV \ . \tag{1.15}$$

Los valores y vectores obtenidos mediante este procedimiento se conocen como valores y vectores de Ritz. Un aspecto interesante es que se pueden sustituir los vectores propios del paso 5 por vectores de Schur. Los vectores de Schur se pueden obtener de forma robusta y, en general, son menos sensibles a los errores de redondeo que los vectores propios.

1.1.4. Métodos de proyección

Los métodos de proyección se basan en construir un subespacio y aplicar el procedimiento de Rayleigh-Ritz. La construcción de la base del subespacio se basa en productos por la matriz del problema y en alguna forma de garantizar la ortogonalidad de los vectores generados. El cálculo de los valores propios de la matriz proyectada se realiza con transformaciones de semejanza. Por lo tanto, estos métodos pueden verse como una combinación de los métodos para matrices densas y dispersas.

Los métodos de Krylov utilizan el subespacio de Krylov generado por una matriz A y un vector inicial v

$$\mathscr{K}_m(A, v) \equiv \left\langle v, Av, A^2 v, \ldots, A^{m-1} v \right\rangle \tag{1.16}$$

donde m es la dimensión máxima del subespacio. El método de Arnoldi [Arn51] construye el subespacio de Krylov vector a vector, utilizando el proceso de Gram-Schmidt para garantizar la ortogonalidad contra todos los vectores de la base calculados en iteraciones anteriores. La forma del subespacio permite aplicar el procedimiento de Rayleigh-Ritz aprovechando los cálculos realizados durante la ortogonalización. Además,se obtiene una estimación del residuo sin necesidad de calcularlo explícitamente. El método de Lanczos [Lan50] puede verse como una simplificación del método de Arnoldi para matrices Hermitianas, donde el proceso de Gram-Schmidt solamente se realiza contra los dos últimos vectores

calculados. Sin embargo, cuando se trabaja en coma flotante este proceso no es suficiente para garantizar la ortogonalidad entre todos los vectores de la base. Esta pérdida de ortogonalidad hace que el método de Lanczos calcule valores propios repetidos o incluso espúreos. Para evitar este problema existen dos alternativas: realizar el proceso de Gram-Schmidt contra más vectores (técnicas conocidas como reortogonalización [Sim84a]) o realizar algún postproceso para eliminar los valores propios no válidos.

Un inconveniente de los métodos de Krylov es que su coste espacial y temporal suele crecer rápidamente con cada iteración del algoritmo. Para evitar este crecimiento del coste se utilizan técnicas de reinicio, que consisten en parar el método al llegar a una base del subespacio de Krylov de tamaño razonable y construir una nueva base aprovechando los vectores de la base anterior. Estas técnicas permiten modificar la convergencia de los métodos de Krylov, permitiendo calcular valores propios interiores. El reinicio explícito es la técnica más sencilla, utilizando como vector inicial de la nueva base una combinación lineal de los vectores de Schur obtenidos a partir de la base anterior. El reinicio implícito [Sor92] es más complicado, permitiendo reutilizar varios vectores de Schur como punto de partida de la nueva base, lo que mejora mucho la convergencia del método. Recientemente se ha propuesto el método de Krylov-Schur [Ste01a] como una alternativa más sencilla de calcular que el reinicio implícito. La versión de Krylov-Schur para matrices Hermitianas se conoce como método de Lanczos con reinicio grueso (*thick restart*) [WS00]. Estos métodos y técnicas se discuten en detalle en el capítulo 3 de esta tesis.

De la misma forma que la iteración del subespacio puede verse como una extensión del método de la potencia, pueden plantearse métodos de Krylov a bloques, es decir, que trabajen con varios vectores simultáneamente. Estos métodos presentan ventajas a la hora de trabajar con valores propios múltiples y se pueden implementar con operaciones de tipo matriz por matriz que aprovechan mejor la jerarquía de memoria. Sin embargo, su implementación resulta bastante compleja y la velocidad de convergencia no suele mejorar con respecto del método con un solo vector.

El método de Davidson [Dav75] trabaja con un subespacio que no es de Krylov, añadiendo vectores de forma que corrijan las aproximaciones al vector propio obtenidas con el subespacio anterior. El vector añadido en cada iteración se calcula como $(D_A - \theta I)^{-1} r$ siendo D_A la diagonal de A y $r = (A - \theta I)z$ el residuo del valor propio buscado θ y su vector propio asociado z. Además, este vector debe ser ortogonal con respecto al subespacio generado por los vectores de iteraciones anteriores. El método de Jacobi-Davidson [SV96] utiliza una

aproximación a $(A-\theta I)^{-1}r$ para el nuevo vector, añadiendo la restricción de ortogonalidad con respecto de z. La ventaja de estos métodos es que para calcular una buena actualización solamente necesitan calcular una aproximación de $(A-\sigma I)^{-1}$ mientras que las transformaciones espectrales necesitan una solución exacta. Para reducir el coste de los métodos de Davidson se pueden aplicar técnicas de reinicio como las utilizadas para métodos de Krylov y también se pueden plantear variantes a bloques.

1.2. Problema de valores singulares

La descomposición de valores singulares, más conocida por las siglas SVD (*singular value decomposition*), de una matriz rectangular $A \in \mathbb{C}^{m\times n}$ se puede escribir como

$$A = U\Sigma V^*, \tag{1.17}$$

donde $U = [u_1,\ldots,u_m]$ es una matriz unitaria $m\times m$ ($U^*U = I$), $V = [v_1,\ldots,v_n]$ es una matriz unitaria $n\times n$ ($V^*V = I$), y Σ es una matriz diagonal $m\times n$ con sus elementos reales no negativos $\Sigma = \operatorname{diag}\left(\sigma_1,\ldots,\sigma_{\min\{m,n\}}\right)$. Estos elementos se denominan valores singulares y se ordenan de forma que $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min\{m,n\}} \geq 0$. El número r de valores singulares mayores que cero es el rango de A. Si A es real, U y V son reales y ortogonales. A los vectores u_i y v_i asociados al valor singular σ_i se les denomina vectores singulares por la izquierda y derecha respectivamente. En esta tesis se asumirá sin pérdida de generalidad que $m \geq n$.

Una aplicación muy importante de la SVD es obtener una aproximación a la matriz $A = U\Sigma V^*$ tomando los primeros t valores y vectores singulares

$$A_t = U_t \Sigma_t V_t^*, \tag{1.18}$$

done $U_t = [u_1,\ldots,u_t]$, $V_t = [v_1,\ldots,v_t]$, y $\Sigma_t = \operatorname{diag}(\sigma_1,\ldots,\sigma_t)$. Esta descomposición se denomina SVD parcial de rango t y minimiza la norma $\|A-A_t\|_F$.

De forma similar a los métodos para valores propios, los métodos para SVD primero reducen la matriz a una bidiagonal mediante transformaciones unitarias. Después se utilizan métodos específicos para calcular la SVD de esta bidiagonal: iteración QR [DK90], DQDS [PM99] o divide y vencerás [GE95]. Estos métodos basados en transformaciones para la SVD no aprovechan el carácter disperso de la matriz. Para calcular descomposiciones parciales en este caso resultan más interesantes los métodos basados en proyecciones.

1.2.1. Relación con el problema de valores propios simétrico

La SVD de A está íntimamente relacionada con los problemas de valores propios de tres matrices Hermitianas:

- Los valores propios de A^*A (producto cruzado) son los cuadrados de los valores singulares de A y sus vectores propios son los vectores singulares por la derecha de A. Los vectores singulares por la izquierda pueden obtenerse mediante $u_i = Av_i/\sigma_i$ para $i = 1,\ldots,r$. Los vectores desde u_{r+1} hasta u_m se eligen de forma arbitrara manteniendo la ortogonalidad con el resto de vectores singulares por la izquierda.

- Los valores propios de AA^* son los cuadrados de los valores singulares de A, más exactamente $m-n$ ceros, y sus vectores propios son los vectores singulares por la izquierda de A. Los vectores singulares por la derecha pueden obtenerse mediante $v_i = A^*u_i/\sigma_i$ para $i = 1,\ldots,r$. Los vectores desde v_{r+1} hasta v_n se eligen de forma arbitrara manteniendo la ortogonalidad con el resto de vectores singulares por la derecha.

- Los valores propios de la matriz cíclica

$$H(A) = \begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix} \tag{1.19}$$

son $\pm\sigma_i$ con el vector propio asociado $\frac{1}{\sqrt{2}}\begin{bmatrix} \pm u_i \\ v_i \end{bmatrix}$ para $i = 1,2,\ldots,n$ y $m-n$ valores propios nulos.

A partir de estas matrices se puede calcular la SVD de una matriz dispersa con cualquier método de los vistos anteriormente para problemas dispersos de valores propios. Estas matrices no deben construirse explícitamente ya que resulta más eficiente calcular los productos matriz vector en función de productos por A y A^*.

1.3. Software disponible

En esta sección se revisa el estado actual del software que implementa métodos para problemas de valores propios y singulares con especial énfasis en los problemas dispersos. Esta revisión solamente tendrá en cuenta el software con

código fuente disponible de forma gratuita. La exclusión del software comercial no supone ninguna pérdida de validez, dado que no suele disponer de características avanzadas o incluso está basado directamente en otro software gratuito. Un ejemplo del primer tipo es la librería comercial HSL que solamente dispone de una implementación paralela del método de Lanczos básico. Ejemplos más significativos son los productos comerciales MATLAB y la librería NAG que utilizan la implementación gratuita de ARPACK para resolver el problema disperso de valores propios.

Prácticamente todo el software de este tipo tiene la forma de librería de funciones (o clases) para el lenguaje en el que está implementado. Este lenguaje condiciona fuertemente la forma de especificar el producto matriz por vector, que es el aspecto más importante de la interfaz de usuario. Además, utilizar una librería desde otro lenguaje distinto al de su implementación suele ser muy complicado, por lo que en algunos casos se proporcionan implementaciones alternativas en MATLAB y en C o Fortran. Las librerías implementadas en Fortran son una excepción ya que pueden ser utilizadas desde C/C++ con unas reglas sencillas de paso de parámetros.

MATLAB proporciona la mayor flexibilidad pudiendo trabajar con sus matrices propias y con el producto implementado en una función definida por el usuario. Las librerías realizadas con C++ tienen casi la misma flexibilidad puesto que incluyen clases con el producto matriz dispersa por vector y también pueden utilizar funciones definidas por el usuario mediante polimorfismo. Por el contrario, las implementaciones con C y Fortran solamente pueden utilizar funciones definidas por el usuario, lo que obliga a implementar el producto matriz por vector en todos los casos. Los punteros a funciones del C permiten mayor flexibilidad que Fortran, donde la función matriz por vector debe tener un nombre fijo de antemano. Para evitar este inconveniente algunas librerías utilizan el denominado *reverse communication interface* [DEK95]. Con este sistema el programa consiste en un bucle que primero llama a la librería y después realiza una operación determinada según el valor de retorno. Esta operación suele ser el producto matriz por vector que define el problema aunque también existen otras posibilidades. Este bucle se repite hasta un valor especial de retorno que indica la convergencia del método. El principal inconveniente de este sistema es la gran complejidad del código, sobre todo en la implementación del método numérico.

La resolución de problemas densos no se contempla en esta revisión dado que está completamente cubierta por la librería LAPACK [ABB+92]. Esta librería está implementada con FORTRAN 77 y su código fuente está disponible de

forma gratuita, lo que la ha convertido en el estándar de facto para aplicaciones de álgebra lineal con matrices densas. LAPACK implementa el método QR con desplazamientos múltiples para el problema de valores propios estándar y el método QZ con desplazamiento doble para el problema generalizado. Para el problema Hermitiano estándar dispone además de otros métodos: divide y vencerás, representaciones relativamente robustas y bisección. El problema Hermitiano generalizado está resuelto mediante un método estándar construyendo una factorización de Cholesky de la matriz B. Para realizar la SVD se dispone de la iteración QR y divide y vencerás, y para calcular solamente los valores singulares se utiliza el método DQDS.

La librería LAPACK original es secuencial, sin embargo algunos fabricantes han implementado versiones paralelas para máquinas de memoria compartida concretas, como por ejemplo, MKL para procesadores Intel y ESSL para máquinas IBM. Para máquinas de memoria distribuida, la librería ScaLAPACK [BCC+97] proporciona versiones paralelas de los algoritmos fundamentales como la iteración QR.

Esta revisión no realiza ninguna comparativa a fondo de los diferentes métodos implementados, para ello el lector puede consultar [LS96a], [BP02], o [AHLT05]. Más adelante, en la sección 4.4.2 se comparan las prestaciones del método de Krylov-Schur implementado en SLEPc con el Arnoldi con reinicio implícito de ARPACK y el Krylov-Schur a bloques de Anazasi. Esta sección está basada en el informe [HRTV07j] publicado como parte de la documentación de SLEPc y que es actualizado periódicamente.

Las tablas 1.1 y 1.2 detallan las librerías disponibles para problemas dispersos de valores propios así como la dirección donde puede obtenerse su código fuente. Estas tablas están ordenadas por el año en el que fue actualizado por última vez el código fuente, de esta forma se puede obtener una idea de la antigüedad del software. La tabla 1.1 muestra las librerías secuenciales mientras que la tabla 1.2 contiene las implementaciones paralelas utilizando el estándar MPI para máquinas de memoria distribuida. La tabla 1.3 proporciona información similar acerca de las librerías disponibles para problemas dispersos de valores singulares.

A continuación se realiza una descripción breve de las características de cada librería agrupadas según el tipo de problema al que están orientadas. Si una librería soporta distintos tipos de problemas se ha clasificado de acuerdo al más general. Las implementaciones en MATLAB suelen trabajar con todos los tipos de aritmética soportadas por el entorno. En los otros lenguajes se suele trabajar

Nombre	**Año**	**Lenguaje**	**Dirección**
Underwood	1975	F77	http://www.netlib.org/go/underwood.f
LOPSI	1981	F77	http://www.netlib.org/toms/570
LASO	1983	F77	http://www.netlib.org/laso
NAPACK	1987	F77	http://www.netlib.org/napack
LANZ	1991	F77	http://www.netlib.org/lanz
LANCZOS	1992	F77	http://www.netlib.org/lanczos/
DVDSON	1995	F77	http://www.cpc.cs.qub.ac.uk/summaries/ACPZ_v1_0.html
QMRPACK	1996	F77	http://www.netlib.org/linalg/qmr/
ARNCHEB	1997	F77	http://www.cerfacs.fr/algor/Softs/ARNCHEB/arncheb.html
SRRIT	1997	F77	http://www.netlib.org/toms/776
ARPACK++	1998	C++	http://www.ime.unicamp.br/~chico/arpack++/
JDQR/JDQZ	1998	F77/Matlab	http://www.math.uu.nl/people/sleijpen/index.html
NA18	1999	F77	http://www.netlib.org/numeralgo/na18
INSYLAN	2000	Matlab	http://www.cs.ucdavis.edu/~bai/ETsoftware/
JDCG	2000	Matlab	http://mntek3.ulb.ac.be/pub/docs/jdcg/
SPAM	2001	F90	http://www-unix.mcs.anl.gov/scidac/beskinetics/spam.htm
EIGIFP	2004	Matlab	http://www.ms.uky.edu/~qye/software.html
IRBLEIGS	2004	Matlab	http://www.math.uri.edu/~jbaglama/
IETL	2006	C++	http://www.comp-phys.org/software/ietl
JADAMILU	2006	F77	http://homepages.ulb.ac.be/~jadamilu/
PySPARSE	2007	Python	http://pysparse.sourceforge.net/

Tabla 1.1 Librerías secuenciales para problemas dispersos de valores propios.

Nombre	**Año**	**Lenguaje**	**Dirección**
ARPACK	1995	F77	http://www.caam.rice.edu/software/ARPACK/
BLZPACK	2000	F77	http://crd.lbl.gov/~osni
PDACG	2000	F77	http://www.dmsa.unipd.it/~sartoret/Pdacg/pdacg.htm
TRLAN	2002	F90	http://crd.lbl.gov/~kewu/trlan.html
BLOPEX	2004	C/Matlab	http://www-math.cudenver.edu/~aknyazev/software/BLOPEX
PRIMME	2006	C	http://www.cs.wm.edu/~andreas/software
ANASAZI	2007	C++	http://trilinos.sandia.gov/packages/anasazi/

Tabla 1.2 Librerías paralelas para problemas dispersos de valores propios.

Nombre	**Año**	**Lenguaje**	**Dirección**
SVDPACK	1992	F77/C	http://www.netlib.org/svdpack
PROPACK	2005	F77/Matlab	http://soi.stanford.edu/~rmunk/PROPACK

Tabla 1.3 Librerías para problemas dispersos de valores singulares.

solamente con aritmética real de doble precisión, en caso contrario se incluyen los detalles en la descripción de la librería.

1.3.1. Problema simétrico

Underwood proporciona una rutina de Lanczos a bloques con reortogonalización completa [GU77].

LASO implementa un método de Lanczos a bloques con reortogonalización selectiva [PS79].

LANCZOS emplea un método de Lanczos con la estrategia de postproceso propuesta en [CW85] para evitar la reortogonalización.

DVDSON [SF94] utiliza el método de Davidson a bloques con algunas extensiones como reortogonalización.

NA18 [SS99] implementa una versión del método de Davidson con deflación y bloques de tamaño variable para calcular tanto los valores propios a la izquierda del espectro como los de la derecha.

SPAM extiende el método de Davidson con la técnica de aproximación descrita en [SWTM01]. Dispone de versiones para aritmética de precisión doble y simple.

TRLAN es un método de Lanczos con reinicio grueso [WS00] paralelizado con MPI. Está implementado con Fortran 90 pero puede utilizarse desde FORTRAN 77 y C.

1.3.2. Problema generalizado Hermitiano

Estas librerías pueden utilizarse también para el problema simétrico estándar.

LANZ [JP93] es una implementación de Lanczos con reortogonalización parcial. Incorpora rutinas para transformación espectral y permite restringir la convergencia a un intervalo determinado del espectro. También permite calcular la inercia del problema, que es el número de valores propios positivos, negativos o cero. Soporta paralelismo en máquinas de memoria compartida.

BLZPACK es una versión de Lanczos paralelizada con MPI [Mar95]. Implementa un método a bloques que combina reortogonalización parcial y selectiva. También permite restringir la convergencia a un intervalo determinado del espectro. Soporta aritmética de precisión doble y simple. Emplea el *reverse communication interface* para especificar el producto matriz vector.

INSYLAN es un prototipo en MATLAB que implementa el método de Lanczos simétrico indefinido para problemas generalizados donde la matriz B no es simétrica definida positiva [BEK00].

PDACG es una implementación paralela con MPI del método del gradiente conjugado acelerado con deflación (DACG) [GSF92]. Este método minimiza el cociente de Rayleigh usando gradiente conjugado sobre subespacios de tamaño decreciente. A diferencia de otras librerías en C y Fortran, incluye una implementación paralela del producto matriz vector.

BLOPEX proporciona el método del gradiente conjugado precondicionado localmente óptimo a bloques (LOBPCG) [Kny01] para el cálculo de los valores propios interiores. Este método es una variante del gradiente conjugado precondicionado que se basa en la optimización local de una recurrencia de tres vectores: el actual vector propio aproximado, el residuo precondicionado y un cómputo implícito de la diferencia entre el vector actual y el de la iteración anterior.

Este paquete tiene varias versiones: una en MATLAB (que soporta números complejos), una secuencial implementada en C y otra paralela en C con MPI que necesita ser compilada junto a la librería HYPRE. Esta librería proporciona precondicionadores con métodos paralelos multimalla para problemas tanto estructurados como no estructurados [FY08].

EIGFP es una implementación en MATLAB del método de Krylov precondicionado sin inversa [GY03].

PRIMME [Sta07, SM07] proporciona un método genérico basado en Davidson y Jacobi-Davidson para matrices reales simétricas y complejas Hermitianas. Mediante parámetros el método se puede particularizar incluyendo GD+1, JDQMR, y LOBPCG. La implementación está realizada en C con MPI y dispone de una interfaz para FORTRAN 77.

JADAMILU [BN07] es una implementación en FORTRAN 77 del método de Jacobi-Davidson que incorpora un precondicionador de factorización incompleta (ILU).

PySPARSE es un paquete Python que proporciona una interfaz a la implementación en C del método de Jacobi-Davidson [Geu02]. Esta implementación trabaja a bloques y está optimizada para problemas simétricos.

1.3.3. Problema no Hermitiano

Estas librerías están orientadas al problema no Hermitiano, aunque algunas también incluyen métodos específicos para problemas Hermitianos. Estos algoritmos suelen ser variantes del mismo método aprovechando las distintas propiedades de ambos problemas.

LOPSI [SJ81] implementa la iteración del subespacio trabajando con vectores propios directamente por lo que es poco robusta.

NAPACK proporciona una versión del método de la potencia que puede trabajar con valores propios complejos conjugados. También incluye una versión básica de Lanczos sin reortogonalización ni postproceso.

QMRPACK [FN96] implementa el método de Lanczos por dos lados con *look-ahead*. Este método plantea una doble recurrencia para calcular los vectores propios por la derecha $Ax = \lambda x$ y por la izquierda $y^*A = \lambda y^T$. Utiliza el *reverse communication interface* para implementar el producto matriz vector y matriz transpuesta vector. Soporta aritmética real y compleja tanto en precisión doble como simple.

ARNCHEB [Bra93] implementa el método de Arnoldi con reinicio explícito, combinado con aceleración mediante polinomios de Chebyshev. Estos polinomios se utilizan para construir el vector de reinicio filtrando los vectores correspondientes a los valores propios no deseados.

SRRIT [BS97] es una versión avanzada de la iteración del subespacio basada en la descomposición de Schur. Utiliza la proyección de Rayleigh-Ritz y bloqueo de vectores convergidos. Dispone de versiones para aritmética de precisión doble y simple.

ARPACK [LSY98, MS96] proporciona el método de Arnoldi con reinicio implícito para aritmética real y compleja. Puede trabajar en ambos casos con precisión simple y doble. Para el caso Hermitiano se utiliza Lanczos con reortogonalización completa y reinicio implícito. ARPACK es uno de los paquetes más populares debido a su eficiencia y robustez. El producto matriz

vector se especifica mediante el *reverse communication interface*. La versión paralelizada con MPI se conoce también por el nombre de PARPACK mientras que ARPACK++ es una interfaz C++ a la versión secuencial.

JDQR es una implementación en MATLAB del método de Jacobi-Davidson descrito en [FSdV99]. La versión para problemas generalizados está implementada en otro paquete denominado JDQZ que también está disponible en FORTRAN 77 con aritmética compleja. JDCG [Not02] es una versión de JDQR para problemas simétricos utilizando el método del gradiente conjugado para resolver el sistema lineal interno.

IRBLEIGS [BCR03a] es una implementación en MATLAB de un método de Lanczos a bloques con reinicio implícito para problemas generalizados simétricos definidos positivos. También dispone de un método de Krylov-Schur a bloques para problemas no Hermitianos que mantiene la ortogonalidad con un procedimiento basado en reflexiones de Householder [Bag08].

ANASAZI [BHLT07] implementa el método de Krylov-Schur a bloques que es un algoritmo equivalente a Arnoldi con reinicio implícito. Además, dispone de los métodos de Davidson a bloques y LOBPCG para problemas simétricos. ANASAZI soporta paralelismo mediante MPI y forma parte de TRILINOS [HBH+05], una librería de clases que es independiente de la aritmética subyacente mediante el uso de las características avanzadas de C++.

IETL implementa los métodos de la potencia, iteración inversa y cociente de Rayleigh para aritmética real y compleja. También incluye una versión de Lanczos sin reortogonalización utilizando el postproceso propuesto en [CW85].

1.3.4. Problema de valores singulares

Las librerías descritas anteriormente pueden utilizarse para calcular la SVD mediante el producto cruzado o la matriz cíclica. De hecho, en algunas librerías se incluye alguna interfaz para ello o ejemplos de como hacer la implementación. En esta sección se describen las librerías específicas (tabla 1.3) para el cálculo de la SVD.

SVDPACK [Ber92] proporciona cuatro métodos distintos que calculan los valores propios del producto cruzado o de la matriz cíclica:

- Lanczos con reortogonalización selectiva.
- Lanczos híbrido a bloques con reortogonalización completa.
- Iteración del subespacio (procedimiento Ritzit de Rutishauser).
- Minimización de la traza.

Además de la implementación original en FORTRAN 77 existe otra en C y una interfaz C++.

PROPACK [Lar98] está basado en el método de Lanczos con reortogonalización parcial y trabaja sin construir el problema de valores propios equivalente. La versión en FORTRAN 77 incluye versiones sin reinicio y con reinicio implícito, soporta matrices reales y complejas tanto en doble como simple precisión y esta paralelizada para un modelo de memoria compartida con OpenMP. La implementación en MATLAB no dispone de reinicio implícito.

IRBLEIGS que ha sido descrita en la sección anterior, incluye una rutina MATLAB para calcular los valores propios de la matriz cíclica con el método de Lanczos con reinicio implícito. También se incluyen versiones específicas para la SVD del método de Lanczos con reinicio grueso y valores de Ritz armónicos [BR05, BR06].

Capítulo 2

La librería SLEPc

La librería SLEPc (Scalable Library for Eigenvalue Problem Computations) proporciona métodos iterativos para resolver problemas de valores propios y singulares. La implementación de estos métodos está optimizada para máquinas paralelas y problemas dispersos de gran dimensión. Estos problemas se pueden definir mediante matrices con aritmética real o compleja, e incluso con un producto matriz-vector definido por el usuario, lo que permite una gran flexibilidad. Además, la librería soporta problemas de valores propios generalizados mediante transformaciones espectrales.

SLEPc dispone de implementaciones propias de los métodos de la potencia, iteración inversa, iteración del cociente de Rayleigh (RQI), iteración del subespacio con proyección y, como consecuencia de los trabajos descritos en esta tesis, de los métodos con reinicio de Arnoldi, Lanczos, Krylov-Schur y Golub-Kahan-Lanczos.

SLEPc está siendo desarrollada y mantenida por el GRyCAP (Grupo de Grid y Computación de Altas Prestaciones) de la Universidad Politécnica de Valencia [HRV03, HRV05, HRTV07g] siguiendo estos criterios:

Estabilidad y robustez: De todas las posibles implementaciones de los algoritmos se escoge la más estable numéricamente. También los algoritmos incorporan detectores de condiciones de fallo, que abortan la ejecución y avisan al usuario de las causas del problema.

Eficiencia: Se intenta que las implementaciones paralelas de los algoritmos sean lo más eficientes posible. Además, se realizan modificaciones a los

algoritmos para reducir el coste de las comunicaciones entre procesadores. Si estas modificaciones comprometen la estabilidad numérica de los métodos, no están activas a no ser que sean solicitadas explícitamente por el usuario.

Abstracción: Los detalles de la implementación están ocultos al usuario mediante una interfaz común a todos los métodos. De esta forma se permite cambiar fácilmente de método numérico sin necesidad de reescribir el código de la aplicación. Además, esta interfaz común permite acceder a métodos numéricos implementados en otras librerías numéricas paralelas como ARPACK, BLZPACK, TRLAN, BLOPEX o PRIMME.

Documentación: Se ha preparado un completo manual de usuario [HRTV07h] que cubre el uso básico del software. El código fuente está ampliamente documentado permitiendo ofrecer un completo manual on-line de referencia. Este manual se genera de forma automática reduciendo los errores y el tiempo de preparación. Además, se incluyen ejemplos prácticos para que los usuarios los puedan tomar como punto de partida para sus aplicaciones. Para profundizar en los detalles de los métodos numéricos se han publicado unos informes técnicos que cubren los aspectos más avanzados [HRTV07d, HRTV07f, HRTV07i, HRTV07a, HRTV07c, HRTV07b, HRTV07e].

Facilidad de uso: La interfaz de usuario de los métodos será sencilla, permitiendo una utilización efectiva de la librería sin necesidad de conocer toda su funcionalidad. Además la interfaz debe permitir a los usuarios avanzados tener control sobre los métodos numéricos empleados.

Analizando a fondo los criterios de robustez, abstracción y documentación, estos pueden considerarse como una especialización de la facilidad de uso. Si una librería es robusta, el usuario no necesita preocuparse de que al cambiar el problema por otro similar deje de funcionar correctamente. Esto es particularmente importante con los valores propios, donde el tamaño del problema influye mucho en sus propiedades. En este contexto la convergencia de un algoritmo con una matriz pequeña de prueba no garantiza que el mismo algoritmo funcione correctamente con una matriz más grande. Por otro lado, una buena abstracción permite al usuario concentrarse en su problema sin necesidad de conocer los entresijos de la implementación. Y por último, una buena documentación es imprescindible para que el usuario sea capaz de aprovechar las características del software.

2.1. La librería PETSc

La librería SLEPc está construida a partir de las primitivas proporcionadas por PETSc (Portable, Extensible Toolkit for Scientific Computation) [BBG+07, BBE+07, BGMS97]. Esta librería proporciona los mecanismos para implementar de forma paralela aplicaciones modeladas mediante ecuaciones en derivadas parciales. A diferencia de otros paquetes similares más tradicionales, PETSc utiliza un diseño orientado a objetos.

El paradigma de programación orientado a objetos está basado en objetos formados por atributos (datos) y métodos (código). Los objetos se definen mediante clases que especifican la estructura de los atributos y la implementación de los métodos. Comparando con un paradigma de programación clásico, las clases son el equivalente a un tipo de datos definido por el usuario y los métodos son el equivalente a los procedimientos o funciones. Cada uno de los objetos creados en tiempo de ejecución se denominan instancias, ocupando una posición única en la memoria de forma equivalente a las variables de otros paradigmas de programación. Una instancia puede ser a su vez un atributo de otros objetos, lo que permite construir nuevos objetos por agregación de otros. Los atributos de una instancia solamente pueden ser accedidos en los métodos de esa misma instancia, esta restricción se conoce como encapsulamiento. En esta tesis se utiliza la palabra objeto para referirse tanto a la clase que lo define como a una instancia cualquiera cuando en el contexto no es necesario hacer esta distinción.

La pieza fundamental del paradigma de programación orientado a objetos es la herencia, mecanismo por el cual se puede definir una clase (subclase) a partir de otra clase (superclase) añadiendo o modificando atributos y/o métodos, lo que permite obtener nuevos objetos por especialización de otros. La herencia permite acceder a las instancias mediante una referencia a su superclase, obteniendo un tipo de polimorfismo si algún método ha sido redefinido. De esta forma la superclase define una interfaz común que puede ser implementada de forma diferente en las subclases. Una clase abstracta es aquella que proporciona solamente la declaración de los métodos dejando la implementación a las subclases.

PETSc utiliza clases para modelar tanto estructuras de datos como algoritmos numéricos. Esta encapsulación permite ocultar en gran medida los detalles de implementación, incluidas las distribuciones de datos y de cálculo entre los procesadores. Además la utilización de objetos proporciona una gran flexibilidad

permitiendo aprovechar las implementaciones realizadas con otras librerías numéricas similares de forma casi transparente.

PETSc proporciona muchos de los mecanismos necesarios para aplicaciones paralelas de álgebra lineal, como construcción de vectores y matrices que permiten solapar comunicaciones y cálculo. Además, incluye soporte para vectores distribuidos para métodos en diferencias finitas con mallas estructuradas. PETSc dispone de un buen número de implementaciones paralelas de precondicionadores y métodos de Krylov para sistemas lineales. También incluye implementaciones paralelas de métodos de Newton para sistemas no lineales y de métodos para ecuaciones diferenciales ordinarias.

Un aspecto importante de PETSc es que se puede controlar mediante parámetros de ejecución la selección de estructuras de datos y algoritmos e incluso establecer muchos de los atributos durante la creación de las instancias. Gracias a esto, el usuario dispone de una gran flexibilidad para probar distintos métodos numéricos sin necesidad de modificar su código.

2.1.1. Diseño orientado a objetos

PETSc tiene un diseño orientado a objetos que modela en forma de clases tanto los vectores y matrices como los diferentes algoritmos numéricos. La interfaz a las matrices es independiente de la estructura de datos utilizada, disponiendo de distintas alternativas para el almacenamiento. Los algoritmos que resuelven el mismo tipo de problema derivan de una clase abstracta que proporciona una interfaz común. Mediante esta interfaz es posible cambiar de forma transparente de método numérico simplemente cambiando la clase de la instancia actual. El ciclo de vida típico de estos objetos es:

1. Creación del objeto
2. Establecimiento de parámetros y selección del método iterativo
3. Preparación de la ejecución
4. Ejecución de la parte iterativa
5. Recuperación de la solución
6. Destrucción y liberación de memoria

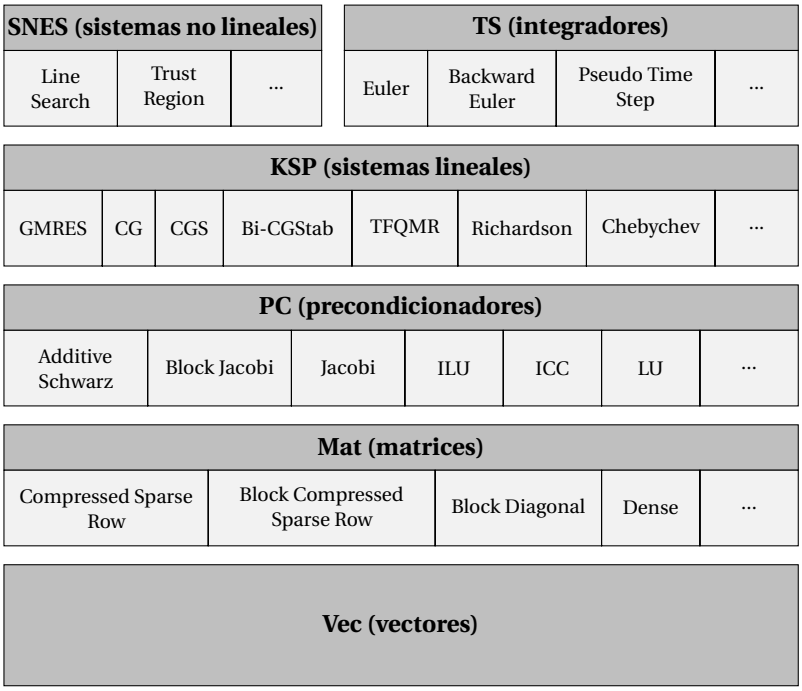

Figura 2.1 Clases de la librería PETSc.

Esta misma filosofía de diseño se aplica a las estructuras de datos como matrices y vectores, resultando muy interesante para ocultar las distintas posibilidades de implementación de las matrices dispersas.

La programación con PETSc se realiza principalmente mediante las clases de interfaz mostradas en la figura 2.1. Cada uno de los bloques se corresponde con clases abstractas que encapsulan tanto estructuras de datos (matrices y vectores) como métodos de resolución (sistemas de ecuaciones lineales y no lineales e integradores). En la parte inferior de los bloques se muestran algunas de las subclases encargadas de la implementación específica de una estructura de datos o de un método de resolución. La librería está organizada jerárquicamente de forma que cada clase depende para su implementación únicamente de la interfaz abstracta de las clases en los niveles inferiores del diagrama. Por ejemplo, los sistemas lineales utilizan precondicionadores, matrices y vectores, y a su vez

los precondicionadores están implementados mediante matrices y vectores. De esta manera resulta posible utilizar prácticamente cualquier combinación de algoritmos y estructuras de datos para resolver un problema determinado. Por ejemplo, todos los métodos para sistemas lineales pueden utilizar cualquiera de los precondicionadores, incluyendo algunos definidos en librerías externas. Este esquema permite también incluir métodos directos tratándolos como un caso especial en el que el precondicionador es exactamente una factorización completa de la matriz.

PETSc está implementado con el lenguaje C a pesar de que éste no dispone de soporte para objetos. Esto se debe a que cuando empezó su desarrollo otros lenguajes más adecuados como el C++ no estaban completamente estandarizados o no resultaban eficientes para la computación de altas prestaciones. Las principales ventajas de utilizar C son su gran eficiencia y portabilidad. Otra importante ventaja es que permite de forma fácil el uso de la librería desde otros lenguajes de programación como Fortran, C++ y Python. El único inconveniente es la mayor complejidad del código, puesto que las características orientadas a objeto deben ser implementadas por la propia librería en tiempo de ejecución. Gracias a la eficiencia de C, este soporte a objetos tiene un coste similar o incluso menor que en otros lenguajes especializados.

La implementación del polimorfismo se realiza mediante punteros a funciones almacenados en una estructura de C junto a los atributos de la instancia. Estos punteros son inicializados en el momento de establecer la clase del objeto. Para invocar a los métodos de las clases se utilizan *funciones envoltorio*. Por convención, el nombre de estas funciones empieza por la clase del objeto invocado y su primer parámetro es la propia instancia del objeto. Estas funciones únicamente llaman al método correspondiente mediante su puntero ocultando al usuario los detalles de esta implementación. Si el método invocado no está definido en la clase se genera una excepción.

La herencia se permite solamente entre dos niveles de clases, para lo que se utiliza un puntero en la superclase a una estructura donde se almacenan los atributos añadidos por la subclase. Los métodos añadidos por la subclase utilizan el mismo sistema de funciones envoltorio con la salvedad de que si el método no esta definido en la clase no se ejecuta ningún código ni se produce ninguna excepción. Este comportamiento está pensado para facilitar el cambio de clase en tiempo de ejecución.

De la misma forma que con la herencia, el tratamiento de excepciones está gestionado por la propia librería PETSc. Por convención, todas las funciones de PETSc devuelven un entero distinto de cero para indicar condiciones de

excepción. El tipo de la excepción ocurrida se indica mediante unos códigos de error predefinidos. Para comprobar de forma fácil el valor de retorno de las funciones se proporciona al usuario una macro que, por defecto, aborta el programa y muestra en pantalla la traza de ejecución en caso de error. Este comportamiento puede modificarse mediante funciones de gestión definidas por el usuario.

PETSc utiliza por defecto aritmética real de doble precisión en todos sus cálculos. Pero se puede recompilar para otros tipos de datos simplemente cambiando la definición de los tipos básicos mediante macros. De esta forma PETSc soporta aritmética compleja, para lo que requiere un compilador de C++ o de C99. El principal inconveniente de este enfoque es que los nombres de la funciones no cambian para reflejar el tipo de datos con el que trabajan, lo que impide el uso simultáneo de versiones compiladas para distintas aritméticas.

PETSc proporciona también herramientas para estudiar las prestaciones de una aplicación midiendo el tiempo de ejecución y el número de operaciones en coma flotante realizadas. Esta información se agrupa por secciones del programa definidas por el usuario y por primitivas de PETSc.

2.1.2. Modelo de paralelismo

En esta sección se presenta el modelo de paralelismo utilizado en PETSc y SLEPc y su relación con el estado actual de la computación paralela. Para un análisis más profundo de este tema el lector puede consultar obras generales como [WA98, GGKK03] o más específicas como [GLS99, CDK+01, AR06]. Las máquinas paralelas más potentes en la actualidad son prácticamente todas de tipo cluster con un gran número de nodos interconectados mediante una red de altas prestaciones. Estos nodos son ordenadores completos construidos a partir de los mismos componentes utilizados para equipos convencionales de gama alta, de esta forma se aprovecha al máximo la economía de escala. Como cada uno de los nodos dispone de su propia memoria principal, el cluster es considerado una máquina paralela de memoria distribuida.

Desde hace bastante tiempo, los microprocesadores de gama alta incorporan facilidades hardware para compartir la memoria entre un número pequeño de procesadores independientes. Recientemente, este número se ha incrementado gracias a la inclusión de varios núcleos de proceso en un único circuito integrado. De esta forma, cada uno de los nodos de un cluster se convierte en una pequeña máquina paralela de memoria compartida. La tabla 2.1 muestra como ejemplo las 10 máquinas más potentes según la lista TOP500 [DMS97] de junio del 2008.

Pos.	Arquitectura	Frecuencia	Red	Procs.	Nodos	P/N
1	Opteron	1,8 Ghz	Infiniband	6.120	3.060	2
2	PowerPC	700 Mhz	BlueGene/L	212.992	106.496	2
3	PowerPC	850 Mhz	BlueGene/P	163.840	40.960	4
4	Opteron	2,0 Ghz	Infiniband	62.976	3.936	16
5	Opteron	2,1 Ghz	Cray XT4	31.328	7.832	4
6	PowerPC	850 Mhz	BlueGene/P	65.536	16.384	4
7	Xeon	3,0 Ghz	Infiniband	14.336	3.584	4
8	Xeon	3,0 Ghz	Infiniband	14.352	1.794	8
9	PowerPC	850 Mhz	BlueGene/P	40.960	10.240	4
10	Xeon	3,0 Ghz	BlueGene/P	10.240	2.560	4

Tabla 2.1 Las 10 máquinas paralelas más potentes según la lista TOP500 de junio del 2008.

En esta tabla se observa que el número de nodos es del orden de decenas o centenas de miles mientras que el número de procesadores dentro de un nodo es de 2 a 16. A pesar de contar con un número más pequeño de procesadores, la máquina en el primer puesto consigue superar a las demás gracias a que dispone de 4 procesadores auxiliares tipo Cell por cada nodo.

Las máquinas paralelas de memoria compartida y distribuida se programan tradicionalmente de forma diferente para aprovechar mejor sus características. Aunque existen herramientas bastante estandarizadas para cada paradigma concreto, no existe un consenso en como programar las máquinas actuales que combinan ambos modelos de memoria. Como el paso de mensajes es más general y funciona también en las máquinas de memoria compartida, la alternativa más sencilla es utilizar este modelo entre todos los procesadores, compartan o no memoria. El inconveniente de esta alternativa es que puede no aprovechar toda la potencia de la máquina.

PETSc utiliza el estándar MPI [MPI94] para la comunicación entre procesadores. Este estándar permite desarrollar programas independientes de la arquitectura hardware paralela subyacente. Se define como una librería de funciones que encapsulan las primitivas de paso de mensajes entre procesadores, de esta forma puede utilizarse desde distintos lenguajes de programación secuenciales, aprovechando los conocimientos y desarrollos previos. En el modelo de ejecución de PETSc cada procesador arranca una copia idéntica del programa en un espacio de direcciones privado (SPMD, *simple program multiple data*), lo

que se ajusta a un modelo hardware de memoria distribuida. También puede utilizarse en máquinas de memoria compartida, aunque con un paralelismo de grano más grueso que otros modelos específicos para estas arquitecturas. MPI está implementado en prácticamente todas las máquinas paralelas y suele tener excelentes prestaciones porque ha sido optimizado para el hardware de interconexión por el propio fabricante. También existen implementaciones del estándar de dominio público que funcionan sobre hardware de interconexión de propósito general como MPICH [GLDS96, GL96] y OpenMPI [GFB+04].

El coste de la comunicación en el paradigma de paso de mensajes resulta complicado de estimar con precisión debido a la cantidad de elementos que participan, desde el medio físico de la red hasta la calidad de la implementación de la librería MPI. En particular, las características de la red pueden variar con la carga del sistema dependiendo de la topología y del encaminamiento utilizado. Sin embargo, existe un modelo muy sencillo, aunque inexacto, que permite predecir de forma general las prestaciones de las aplicaciones paralelas con paso de mensajes.

Este modelo simplificado de paso de mensajes considera que las comunicaciones se realizan mediante *tramas*. Para enviar cada una de estas tramas es necesario esperar un tiempo de acceso al medio. Este tiempo se considera muy superior al necesario para transmitir una pequeña cantidad de datos a la velocidad proporcionada por el ancho de banda de la red. También se considera que ambos tiempos son muy superiores al necesario para realizar operaciones aritméticas con una cantidad de datos similar. Este modelo se cumple para casi todas las clases de redes de interconexión utilizadas en máquinas tipo cluster, desde redes de área local de propósito general hasta redes de propósito específico y altas prestaciones. Aunque estas redes específicas tienen un tiempo de acceso al medio muy rápido, generalmente las tramas contienen una cabecera/pie de tamaño no despreciable con información necesaria para el funcionamiento de la red, como encaminamiento, corrección de errores, etc. También suele existir un tamaño mínimo de trama, por lo que para enviar una cantidad de datos pequeña es necesario añadir datos de relleno, aumentando el sobrecoste de la transmisión. El tamaño máximo de trama se considera lo bastante grande como para permitir la transmisión de un número razonable de datos. Este modelo también es válido para el paso de mensajes en una máquina de memoria compartida, donde el tiempo necesario para la sincronización y copia de datos entre procesos es mucho mayor que el tiempo necesario para realizar operaciones aritméticas con una cantidad de datos similares.

La principal consecuencia de este modelo es que para transmitir igual cantidad de datos es mucho más eficiente utilizar pocos mensajes de tamaño grande que muchos mensajes pequeños. También puede ser más eficiente un algoritmo que realice muchos cálculos y pocas comunicaciones que otro algoritmo equivalente que haga menos cálculos a costa de más comunicaciones, aunque existen otros factores que influyen en la eficiencia como las sincronizaciones o el solapamiento entre comunicaciones y cálculo.

2.1.3. Distribución de datos

En PETSc se intenta que la paralelización sea prácticamente transparente para el usuario. Para ello la gestión de la distribución de datos se realiza de forma automática. Esto no impide que el usuario pueda modificar esta distribución si lo considera necesario. La distribución de datos por defecto en PETSc para las matrices es por bloques de filas donde a cada procesador se le asigna un conjunto de filas contiguas. Los elementos de los vectores se encuentran repartidos en bloques contiguos entre los procesadores. Con esta distribución, la paralelización de las primitivas típicas que aparecen en los algoritmos para matrices dispersas queda de la forma siguiente:

Producto matriz por vector: PETSc proporciona la función `MatMult`, cuyas prestaciones dependen del patrón de dispersión de la matriz. Si se elige un reparto adecuado de los elementos entre los procesadores solamente son necesarias comunicaciones punto a punto entre procesadores vecinos.

Producto escalar y norma de vectores: Las funciones `VecDot` y `VecNorm` calculan el producto escalar y la norma con una multireducción. Esta primitiva de MPI es una comunicación global donde todos los procesadores envían sus datos parciales y todos reciben el resultado global de la operación. La implementación de esta primitiva implica múltiples mensajes y está optimizada para cada arquitectura de red específica. También se dispone de la función `VecMDot` que permite realizar el producto con varios vectores con una única comunicación.

Suma, escalado y copia de vectores: Estas operaciones (`VecAXPY`, `VecScale`, `VecSet`, `VecCopy`) pueden realizarse totalmente en paralelo sin comunicaciones gracias a cómo se realiza la distribución de los datos en PETSc. También se dispone de la función `VecMAXPY` que es una versión optimizada de `VecAXPY` para la suma escalada de varios vectores.

Todas estas primitivas tienen en común realizar un número de operaciones aritméticas del orden de la cantidad de datos utilizados [GVL96, sec. 1.1.15]. Por ejemplo, el cálculo de la norma de un vector supone aproximadamente dos operaciones aritméticas por cada elemento. Como la memoria es mucho más lenta que la unidad aritmética, las prestaciones están limitadas por el ancho de banda disponible entre el procesador y la memoria. Además, en estos casos la cantidad de datos manejada por el algoritmo suele demasiado grande como para poder aprovechar la jerarquía de memorias. Esto no es importante en una máquina de memoria distribuida donde cada procesador dispone de un bus de memoria independiente, pero resulta crítico en una máquina de memoria compartida. Por lo tanto, en las máquinas tipo cluster actuales, utilizar más de un proceso MPI dentro de un nodo puede resultar contraproducente para este tipo de primitivas.

2.2. Interfaz de usuario de SLEPc

SLEPc añade al esquema de PETSc dos clases de interfaz nuevas, EPS y ST, para resolver el problema de valores propios. La clase EPS modela los métodos iterativos y la clase ST modela las transformaciones espectrales adecuadas para estos métodos. En la figura 2.2 se pueden observar las distintas clases disponibles en la interfaz de programación. SLEPc continua la filosofía de PETSc proporcionando interfaces para otras librerías con métodos iterativos de resolución del problema de valores propios.

La ejecución de un método iterativo en SLEPc está esquematizada en la figura 2.3, donde el acceso a la matriz del problema se realiza únicamente mediante un producto matriz vector. Si es necesario, también se utiliza un producto interior para mantener la simetría en los problemas generalizados tal y como se detalla en la sección 3.2.2. Estas dos operaciones están proporcionadas por la transformación espectral. De esta forma se simplifica la implementación y a la vez se obtiene la posibilidad de utilizar todas las combinaciones posibles de métodos y transformaciones.

La transformación espectral está implementada con primitivas de PETSc para matrices dispersas. En lugar de construir la inversa de las matrices del problema se utilizan los métodos de resolución para sistemas lineales de PETSc, con lo que el producto matriz vector se convierte en la resolución de un sistema lineal con un método iterativo. Para obtener la máxima precisión en el problema de valores propios se deben utilizar métodos lineales directos, que son un caso

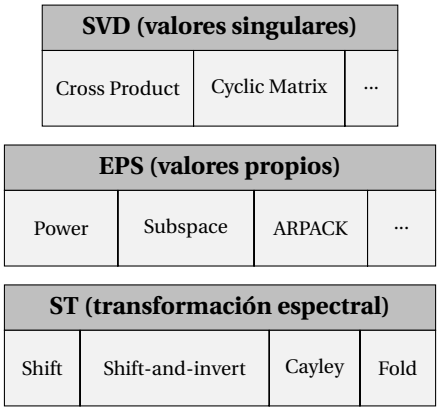

Figura 2.2 Clases de la librería SLEPc.

especial del modelo de ejecución de PETSc, colocando una factorización exacta en lugar del precondicionador.

Para realizar la descomposición en valores singulares se ha añadido a SLEPc la clase de interfaz SVD. Esta clase proporciona una interfaz para los métodos iterativos similar al de la clase EPS. Las clases derivadas de SVD proporcionan métodos basados en calcular los valores propios del producto cruzado o la matriz cíclica y métodos específicos para los valores singulares.

2.2.1. EPS

La clase de interfaz EPS modela un método genérico iterativo para resolver el problema de valores propios. Todos los objetos derivados de la clase EPS acceden a las matrices del problema mediante un producto matriz-vector definido por un objeto de la clase ST. Este objeto también se encarga de definir el producto interior para los problemas generalizados.

El ciclo de vida típico de un objeto EPS es similar a los objetos de PETSc para sistemas lineales:

1. Creación del objeto mediante la función `EPSCreate`

2. Selección del método iterativo y establecimiento de parámetros con varias funciones:

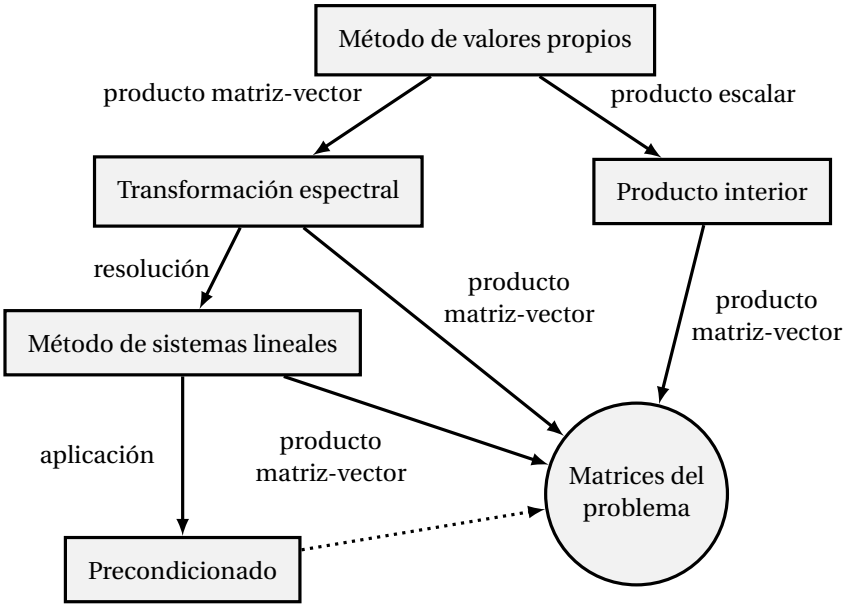

Figura 2.3 Esquema de ejecución de un método iterativo dentro de la clase EPS. Las líneas continuas indican el acceso a una matriz mediante operaciones con vectores, mientras que la línea discontinua indica el acceso a los elementos de la matriz.

- `EPSSetType` selecciona el método iterativo.
- `EPSSetOperators` especifica las matrices del problema.
- `EPSSetProblemType` define el tipo de problema que puede ser:
  - Estándar Hermitiano (`EPS_HEP`).
  - Estándar no Hermitiano (`EPS_NHEP`).
  - Generalizado Hermitiano con B semidefinida positiva (`EPS_GHEP`).
  - Generalizado no Hermitiano (`EPS_GNHEP`).
  - Generalizado no Hermitiano con B Hermitiana semidefinida positiva (`EPS_PGNHEP`).
- `EPSSetWhichEigenpairs` selecciona la parte del espectro donde están los valores propios deseados.
- `EPSSetInitialVector` establece el vector inicial.
- `EPSSetDimensions` especifica los parámetros `nev` (número de valores propios buscados) y `ncv` (tamaño del subespacio).
- `EPSSetTolerances` define la tolerancia para el criterio de parada y número máximo de iteraciones.

Todos los atributos que no implican matrices o vectores pueden especificarse en tiempo de ejecución mediante la función `EPSSetFromOptions` que procesa los parámetros especificados en la línea de ejecución del programa. La tabla 2.2 muestra la correspondencia entre estos parámetros y las funciones para establecer los atributos de la clase EPS.

3. Preparación de la ejecución con la función `EPSSetUp` que comprueba la consistencia de los parámetros. También se realizan todos los cálculos invariantes durante el método iterativo como la construcción del precondicionador o factorización de la matriz del problema.

4. Ejecución de la parte iterativa en la función `EPSSolve`.

5. Recuperación de la solución con las funciones `EPSGetConverged` y `EPSGetEigenpair`. Esta última función calcula los vectores propios a partir de los vectores de Schur solamente si el usuario los solicita.

6. Destrucción y liberación de memoria mediante `EPSDestroy`.

Función	Parámetro de ejecución
EPSSetType	-eps_type
EPSSetProblemType	-eps_hermitian -eps_gen_hermitian -eps_non_hermitian -eps_gen_non_hermitian -eps_pos_gen_non_hermitian
EPSSetWhichEigenpairs	-eps_largest_magnitude -eps_smallest_magnitude -eps_largest_real -eps_smallest_real -eps_largest_imaginary -eps_smallest_imaginary
EPSSetDimensions	-eps_nev -eps_ncv
EPSSetTolerances	-eps_tol -eps_max_it

Tabla 2.2 Equivalencia entre funciones y parámetros de ejecución para establecer atributos en la clase EPS.

Para comprobar la validez del resultado se dispone de las funciones EPSComputeResidualNorm y EPSComputeRelativeError que calculan respectivamente la norma del residuo y el error relativo de un valor propio convergido λ y su vector propio asociado x. La norma del residuo se calcula como $\|Ax - \lambda x\|_2$ y el error relativo como

$$\frac{\|Ax - \lambda Bx\|_2}{\|\lambda x\|_2}.$$

2.2.2. ST

La clase de interfaz ST modela una transformación espectral, existiendo implementaciones del desplazamiento, desplazamiento e inversión, plegado y Cayley. El usuario puede combinar libremente cualquier transformación espectral con cualquier método implementados en las clases derivadas de EPS. Estas transformaciones permiten además convertir el problema de valores propios

Transformación espectral	**Problema estándar**	**Problema generalizado**
Desplazamiento	$A+\sigma I$	$(A+\sigma I)B^{-1}$
Desplazamiento e inversión	$(A-\sigma I)^{-1}$	$(A-\sigma B)^{-1}B$
Cayley	$(A-\sigma I)^{-1}(A+\tau I)$	$(A-\sigma B)^{-1}(A+\tau B)$
Plegado	$(A-\sigma I)^2$	$(B^{-1}A+\sigma I)^2$

Tabla 2.3 Operadores utilizados en las transformaciones espectrales.

Función	**Parámetro de ejecución**
`STSetType`	`-st_type`
`STSetShift`	`-st_shift`
`STSetMatMode`	`-st_matmode`
`STSetMatStructure`	`-st_matstructure`

Tabla 2.4 Equivalencia entre funciones y parámetros de ejecución para establecer atributos en la clase ST.

generalizado en uno estándar. En la tabla 2.3 se muestra un resumen de estas transformaciones en los dos tipos de problemas.

Por defecto, los operadores de las transformaciones espectrales no se construyen explícitamente. En lugar de calcular la inversa de matrices se resuelve un sistema de ecuaciones lineales cada vez que sea necesario con un objeto de la clase KSP. Con este esquema resulta interesante emplear un buen precondicionador, porque su elevado coste de construcción se amortiza enseguida al resolver un sistema lineal con la misma matriz en cada iteración del método de valores propios. Sin embargo, para calcular los valores propios con una buena precisión suele ser necesario utilizar un método directo para matrices dispersas. Actualmente, PETSc no dispone de versiones paralelas de este tipo de métodos y hay que recurrir a librerías externas como MUMPS [ADLK01], SPOOLES [AG99], SuperLU [LD03] o DSCPACK [Rag02]. Estas librerías están totalmente integradas dentro de la clase PC y, gracias a la filosofía de PETSc, su utilización es tan sencilla como la de un precondicionador propio.

El usuario no crea una instancia de la clase ST, ésta se crea automáticamente dentro del objeto EPS y el usuario puede obtener un puntero a este objeto con la función `EPSGetST`. La interfaz de la clase ST es relativamente sencilla permitiendo especificar el tipo de transformación y el valor de σ. La equivalencia entre las funciones y los parámetros de ejecución se muestran en la tabla 2.4. La

funcionalidad más avanzada está en `STSetMatMode` dónde se controla cómo se construirá la matriz $A - \sigma B$ a partir de las matrices del problema. Con esta matriz se resolverá varias veces un sistema lineal, existiendo tres posibilidades:

- Realizar el producto $(A - \sigma B)x$ a partir de los productos Ax y Bx, sin calcular $A - \sigma B$ de forma explícita. Este método tiene el inconveniente de no poder utilizar ningún precondicionador avanzado ni métodos directos y por lo tanto la resolución del sistema lineal puede necesitar muchas iteraciones.
- Mediante una nueva matriz temporal que almacena el resultado $A - \sigma B$. De esta forma se puede utilizar cualquier tipo de precondicionador e incluso calcular la factorización completa de la matriz. Este procedimiento puede resultar muy costoso en términos de memoria.
- Sustituyendo la matriz A por $A - \sigma B$. Esto permite utilizar precondicionadores avanzados como en el caso anterior sin el consumo extra de memoria para la copia de la matriz. Después de la ejecución del método iterativo de valores propios el valor original de A se recupera con $(A - \sigma B) + \sigma B$. El inconveniente de este procedimiento son los posibles efectos laterales al trabajar con precisión finita donde el valor de A recuperado puede ser ligeramente diferente al original.

Las funciones `STGetKSP` y `KSPGetPC` permiten acceder al sistema lineal y al precondicionador utilizado para construir la transformación espectral. Estas dos funciones devuelven punteros a las instancias de las clases KSP y PC creadas dentro del objeto ST. Con estos punteros se pueden modificar sus atributos permitiendo seleccionar los métodos para sistemas lineales y precondicionadores más adecuados para un problema concreto. Una forma más sencilla de modificar estos atributos es mediante los parámetros de ejecución. Los parámetros de los KSP y PC creados dentro del ST empiezan por el prefijo `-st_` para distinguirlos fácilmente de otras instancias creadas por el usuario.

2.2.3. SVD

Para calcular la descomposición en valores singulares parcial se ha añadido a SLEPc la clase de interfaz SVD. Al igual que la clase EPS, esta nueva clase modela un método iterativo, por lo que sus interfaces de usuario son muy similares:

1. Creación del objeto mediante la función `SVDCreate`

2. Selección del método iterativo y establecimiento de parámetros con varias funciones:

   - `SVDSetType` selecciona el método iterativo.
   - `SVDSetOperator` especifica la matriz del problema.
   - `SVDSetWhichSingularTriplets` selecciona si se desean los valores singulares más grandes o más pequeños.
   - `SVDSetInitialVector` establece el vector inicial.
   - `SVDSetDimensions` selecciona el número de valores singulares deseados (`nsv`) y el tamaño del subespacio (`ncv`).
   - `SVDSetTolerances` define la tolerancia para el criterio de parada y número máximo de iteraciones.

   Todos los atributos que no implican matrices o vectores pueden especificarse en tiempo de ejecución mediante la función `SVDSetFromOptions` que procesa los parámetros especificados en la línea de ejecución del programa. La tabla 2.5 muestra la correspondencia entre estos parámetros y las funciones para establecer los atributos de la clase SVD.

3. Preparación de la ejecución con la función `SVDSetUp` que comprueba la consistencia de los parámetros.

4. Ejecución de la parte iterativa en la función `SVDSolve`.

5. Recuperación de la solución con las funciones `SVDGetConverged` y `SVDGetSingularTriplet`. Esta última función permite ahorrar cálculos en algunos casos si se solicitan solamente los vectores singulares por un lado.

6. Destrucción y liberación de memoria mediante `SVDDestroy`.

Para comprobar la validez del resultado se dispone de las funciones `SVDComputeResidualNorms` y `SVDComputeRelativeError` que calculan la norma de los residuos y el error relativo de un trío valor, vector por la izquierda y vector por la derecha singulares convergido. Si u y v son los vectores por la izquierda y la derecha asociados al valor singular σ, las normas de los residuos se calculan como

Función	Parámetro de ejecución
SVDSetType	-svd_type
SVDSetWhichSingularTriplets	-svd_largest -svd_smallest
SVDSetDimensions	-svd_nsv -svd_ncv
SVDSetTolerances	-svd_tol -svd_max_it
SVDSetTransposeMode	-svd_transpose_mode

Tabla 2.5 Equivalencia entre funciones y parámetros de ejecución para establecer atributos en la clase SVD.

$$\gamma_1 = \|Av - \sigma u\|_2$$
$$\gamma_2 = \|A^T u - \sigma v\|_2 .$$

Con estos residuos γ_1 y γ_2 se calcula el error relativo como

$$\delta = \frac{\sqrt{\gamma_1^2 + \gamma_2^2}}{\sigma} .$$

Además del producto matriz vector, los métodos iterativos para valores singulares utilizan también el producto con la matriz transpuesta. Aunque se dispone de esta primitiva para la matrices de PETSc, en general su implementación no resulta tan eficiente como el producto con la matriz sin transponer debido al formato de almacenamiento en memoria. Para evitar esta pérdida de prestaciones a costa de un mayor uso de memoria, la clase SVD construye una copia transpuesta de la matriz. Se asume que la matriz tiene más filas que columnas, en caso contrario se intercambian las referencias a la matriz original y a la copia transpuesta. Estos mecanismos resultan totalmente transparentes para el usuario, pero se puede evitar la creación de la matriz auxiliar con la función SVDSetTransposeMode o el parámetro de ejecución -svd_transpose_mode.

Dentro de la clase SVD existen dos clases derivadas SVDCROSS y SVDCYCLIC para tratar el problema de valores singulares como un problema de valores propios mediante el producto cruzado y la matriz cíclica, respectivamente. Estas matrices no se construyen de forma explícita, sino que se implementa su pro-

ducto matriz vector a partir de los productos matriz vector y matriz transpuesta vector de la matriz original.

Las clases SVDCROSS y SVDCYCLIC construyen internamente una instancia de la clase EPS para resolver el problema de valores propios asociado. Mediante las funciones `SVDCrossGetEPS` y `SVDCyclicGetEPS` se puede obtener el puntero al objeto EPS incluido en la instancia de SVD. Aunque se puede utilizar este puntero para modificar los atributos del objeto EPS, resulta más fácil utilizar los parámetros de ejecución que empiezan por el prefijo `-svd_` para distinguirlos de otras instancias de la clase EPS.

Capítulo 3

Métodos de Krylov

En este capítulo se revisan a fondo los métodos de Arnoldi y Lanczos para problemas de valores propios, y el método de Golub-Kahan-Lanczos para problemas de valores singulares. Estos métodos de Krylov consisten básicamente en un bucle con dos fases: expansión de la base (un producto matriz por vector) y añadir éste resultado a la base (ortogonalización). El producto matriz vector se encuentra implementado de forma eficiente en PETSc, mientras que la ortogonalización se analizará en profundidad en este capítulo, estudiando su impacto en el modelo de paralelismo utilizado en SLEPc. Las principales aportaciones de la tesis se realizan en este campo, proponiendo alternativas que reducen el número de comunicaciones necesarias con respecto de los algoritmos originales. Además se mostrarán las técnicas de reinicio necesarias para mejorar las propiedades de convergencia de los métodos de Krylov.

3.1. Método de Arnoldi

Los métodos de Krylov extraen aproximaciones a los vectores propios de una matriz cuadrada A de orden n a partir del denominado subespacio de Krylov, definido como

$$\mathcal{K}_m(A,v) \equiv \left\langle v, Av, A^2v, \ldots, A^{m-1}v \right\rangle , \tag{3.1}$$

donde v es un vector arbitrario y $m \leq n$ es la dimensión máxima del subespacio.

Si se realizan n pasos del método de Arnoldi (algoritmo 3.1) se obtiene un reducción ortogonal a forma de Hessenberg, $AV = VH$, donde H es una matriz

Algoritmo 3.1 Iteración de Arnoldi

Dado un vector v_1 de norma unidad
para $j = 1, 2, \ldots, m$
  $v_{j+1} = Av_j$
  para $i = 1, 2, \ldots, j$
    $h_{i,j} = v_i^* v_{j+1}$
    $v_{j+1} = v_{j+1} - h_{i,j} v_i$
  fin
  $h_{j+1,j} = \|v_{j+1}\|_2$
  si $h_{j+1,j} = 0$ **parar**
  $v_{j+1} = v_{j+1}/h_{j+1,j}$
fin

Hessenberg superior de orden n, con los elementos de la primera subdiagonal positivos, y V es una matriz unitaria cuyas columnas forman una base del subespacio de Krylov.

Las matrices V y H están determinadas de forma única (salvo cambios de signo) por la primera columna de V, un vector normalizado $v_1 = Ve_1$ denominado vector inicial [GVL96, teorema 7.4.2]. Para algunos posibles vectores iniciales, el método de Arnoldi se tiene que parar después de m pasos porque $\|v_{j+1}\|_2$ se hace cero tras la ortogonalización, debido a que v_{j+1} es una combinación lineal de los vectores anteriores. En ese caso el algoritmo produce una matriz V_m de dimensión $n \times m$, con columnas ortogonales, y una matriz H_m Hessenberg superior de orden m con sus elementos no nulos h_{ij} definidos por el algoritmo, que satisface

$$AV_m - V_m H_m = 0. \tag{3.2}$$

Sin embargo, $\|v_{j+1}\|_2$ nunca llega a ser exactamente cero en la implementación del algoritmo con coma flotante, y puede resultar problemático detectar esta condición.

En el caso general, donde $\|v_{j+1}\|_2$ no se anula después de m pasos, se cumple la siguiente relación

$$AV_m - V_m H_m = h_{m+1,m} v_{m+1} e_m^*, \tag{3.3}$$

donde al producto $h_{m+1,m} v_{m+1}$ se le suele llamar el *residuo* de la factorización de Arnoldi de orden m.

Se puede demostrar que el algoritmo 3.1 construye una base ortogonal del subespacio de Krylov $\mathcal{K}_m(A, v_1)$. Los vectores de la base son las columnas de V_m, que se denominan vectores de Arnoldi. Por construcción se tiene que $V_m^* v_{m+1} = 0$, y multiplicando por la izquierda la ecuación 3.3 por V_m^* se obtiene

$$V_m^* A V_m = H_m \,, \tag{3.4}$$

es decir, la matriz H_m representa la proyección ortogonal de A en el subespacio de Krylov, lo que nos permite calcular m aproximaciones de Rayleigh-Ritz a los valores propios de A,

$$H_m y_i = \lambda_i y_i \,, \tag{3.5}$$

donde los valores λ_i son aproximaciones a los valores propios de A y $x_i = V_m y_i$ a sus vectores propios asociados. Para determinar cuáles de estas m aproximaciones son suficientemente precisas se utiliza la norma del residuo, que se puede calcular con un coste pequeño aplicando la ecuación 3.3

$$\|Ax_i - \lambda_i x_i\|_2 = \|AV_m y_i - \lambda_i V_m y_i\|_2 = \|(AV_m - V_m H_m) y_i\|_2 = h_{m+1,m} |e_m^* y_i| \,. \tag{3.6}$$

3.1.1. Ortogonalización de Gram-Schmidt clásica con refinamiento iterativo

La formulación del método de Arnoldi, presentada en el algoritmo 3.1, utiliza el procedimiento de Gram-Schmidt para ortonormalizar los vectores a medida que se van generando. El procedimiento de Gram-Schmidt clásico calcula la proyección del vector sobre el complemento ortogonal del subespacio generado por los vectores anteriores, es decir, $(I - V_j V_j^*) v_{j+1}$. Dado que las columnas de V_j son ortogonales, se puede calcular esta misma proyección vector a vector como $(I - v_j v_j^*) \cdots (I - v_1 v_1^*) v_{j+1}$, alternativa conocida como procedimiento de Gram-Schmidt modificado. Aunque estos procedimientos son matemáticamente equivalentes, cuando se implementan en coma flotante el Gram-Schmidt modificado resulta mucho más estable que el clásico.

A pesar de utilizar Gram-Schmidt modificado, la implementación del algoritmo 3.1 puede fallar, debido a que el problema de ortogonalizar con respecto de un subespacio de Krylov está muy mal condicionado. Una posible solución es utilizar procedimientos basados en reflexiones de Householder o rotaciones de Givens que son más robustos [GVL96, cap. 5]. Pero estos procedimientos resultan muy complicados de implementar en paralelo de forma eficiente, y por lo tanto no se considerará su aplicación en esta tesis. Otra solución es realizar una segunda pasada del procedimiento de Gram-Schmidt al vector, técnica

conocida como reortogonalización [Rut67, Abd71]. En esta tesis se utilizará el término refinamiento en lugar de reortogonalización, para evitar confusiones con las variantes del método de Lanczos que se presentarán más adelante en este capítulo.

Para reducir el coste del refinamiento se propone en [DGKS76] un criterio muy sencillo de calcular para determinar si es necesaria la segunda ortogonalización. Este criterio consiste en comparar la norma del vector antes y después de la ortogonalización, y realizar el refinamiento si se ha reducido en un factor η. En [GL02] se demuestra que $\eta = 1/\sqrt{2}$ y un único paso de refinamiento es suficiente en casos no muy mal condicionados. La experiencia muestra que permitir un segundo paso de refinamiento mejora el comportamiento del método de Arnoldi, sin perjuicio de los casos más comunes. Si este criterio se sigue cumpliendo después del segundo refinamiento, se puede asumir que los vectores son linealmente dependientes. Esto resulta muy útil para detectar la parada del método de Arnoldi, ya que en coma flotante no es trivial comprobar si el vector se anula después de la ortogonalización.

El procedimiento modificado de Gram-Schmidt recorre y actualiza el vector a ortogonalizar varias veces, y como generalmente este vector no cabe entero en la memoria cache, no resulta eficiente debido a un mal aprovechamiento de la jerarquía de memoria del ordenador. Además, en la implementación en paralelo con SLEPc, para cada producto escalar es necesario una multireducción, lo que supone un elevado coste de comunicaciones. Por el contrario, el procedimiento clásico puede expresarse con operaciones matriz vector, permitiendo el uso de operaciones que aprovechan mucho mejor la jerarquía de memoria, y en paralelo los mensajes correspondientes a los productos escalares se pueden combinar en un único mensaje. Dado que cuando se utiliza refinamiento las implementaciones en coma flotante de ambos procedimientos obtienen el mismo resultado [Hof89], en el resto de esta tesis se trabajará con Gram-Schmidt clásico. Aplicando este procedimiento con refinamiento iterativo al algoritmo 3.1 se obtiene el algoritmo 3.2. En este algoritmo se pueden combinar las comunicaciones del cálculo de $\rho = \|v_{j+1}\|_2$ con las comunicaciones del producto $h_{1:j,j} = V_j^* v_{j+1}$, evitando parte del sobrecoste paralelo de la aplicación del criterio de refinamiento.

3.1.2. Estimación de la norma

En el método de Arnoldi es necesario calcular la norma del vector v_{j+1} después de ortogonalizarlo, lo que supone en la implementación con SLEPc una

Algoritmo 3.2 Iteración de Arnoldi con Gram-Schmidt clásico y refinamiento iterativo

Dado un vector v_1 de norma unidad
para $j = 1, 2, \ldots, m$
  $v_{j+1} = Av_j$
  $\rho = \|v_{j+1}\|_2$
  $h_{1:j,j} = V_j^* v_{j+1}$
  $v_{j+1} = v_{j+1} - V_j h_{1:j,j}$
  $h_{j+1,j} = \|v_{j+1}\|_2$
  $k = 1$
  mientras $h_{j+1,j} < \eta\rho \wedge k < 3$
    $k = k + 1$
    $\rho = h_{j+1,j}$
    $c = V_j^* v_{j+1}$
    $v_{j+1} = v_{j+1} - V_j c$
    $h_{1:j,j} = h_{1:j,j} + c$
    $h_{j+1,j} = \|v_{j+1}\|_2$
  fin
  si $h_{j+1,j} < \eta\rho$ **parar**
  $v_{j+1} = v_{j+1}/h_{j+1,j}$
fin

operación de multireducción en cada iteración. La técnica de estimación de la norma se basa en realizar éste cálculo de forma aproximada, sin necesidad de utilizar comunicaciones. Esta técnica se basa en la relación

$$v'_{j+1} = v_{j+1} - V_j h_{1:j,j} \,, \tag{3.7}$$

donde v_{j+1} y v'_{j+1} son el vector antes y después de la ortogonalización, y $V_j h_{1:j,j}$ es el vector proyección de v_{j+1} sobre el subespacio generado por $v_1, v_2, \ldots, v_j$.

Usando aritmética exacta, v'_{j+1} es ortogonal a este subespacio y entonces es posible aplicar el teorema de Pitágoras al triángulo rectángulo formado por estos tres vectores,

$$\|v_{j+1}\|_2^2 = \|v'_{j+1}\|_2^2 + \|V_j h_{1:j,j}\|_2^2 \,. \tag{3.8}$$

Como las columnas de V_j son ortonormales, la norma que buscamos se puede calcular como

$$\|v'_{j+1}\|_2 = \sqrt{\|v_{j+1}\|_2^2 - \sum_{i=1}^{j} h_{i,j}^2} \,. \tag{3.9}$$

El algoritmo 3.3 muestra la aplicación de esta técnica al método de Arnoldi con el procedimiento Gram-Schmidt clásico y refinamiento iterativo. Para derivar la ecuación 3.9, se ha asumido que v'_{j+1} es ortogonal a $v_1, v_2, \ldots, v_j$ y que éstos son también ortogonales entre sí. Estas suposiciones no se cumplen necesariamente al trabajar con aritmética de precisión finita, y por tanto decimos que se calcula una estimación de la norma. Generalmente, esta estimación es bastante precisa porque los coeficientes $h_{1:j,j}$ son muy pequeños comparados con $\|v_{j+1}\|_2^2$. En casos excepcionales, como cuando v_{j+1} es prácticamente una combinación lineal de los vectores anteriores, estas cantidades son comparables y entonces lo más seguro es calcular la norma de forma explícita. Este detalle se omite en el algoritmo 3.3 para mejorar su legibilidad.

La implementación paralela de esta técnica, con primitivas de PETSc, combina los mensajes asociados al cálculo de la norma y los productos escalares en una única multireducción. Las dos operaciones combinadas están representadas dentro de un marco en el algoritmo 3.3. Dado que el resto de operaciones son totalmente paralelas, se utiliza el mínimo posible de comunicaciones, una por cada pasada de Gram-Schmidt clásico, aplicando refinamiento solamente si es necesario y calculando la norma del vector ortogonalizado. Además, la estimación de la norma necesita menos operaciones aritméticas que el cálculo explícito de la norma.

Esta técnica se ha utilizado por Frank y Vuik [FV99] para mejorar las prestaciones paralelas del método GCR [EES83] para sistemas lineales. Una estrategia

Algoritmo 3.3 Iteración de Arnoldi con Gram-Schmidt clásico, refinamiento iterativo y estimación de la norma

Dado un vector v_1 de norma unidad
para $j = 1, 2, \ldots, m$
  $v_{j+1} = Av_j$
  $\boxed{\begin{array}{l} \rho = \|v_{j+1}\|_2 \\ h_{1:j,j} = V_j^* v_{j+1} \end{array}}$
  $v_{j+1} = v_{j+1} - V_j h_{1:j,j}$
  $h_{j+1,j} = \sqrt{\rho^2 - \sum_{i=1}^{j} h_{i,j}^2}$
  $k = 1$
  mientras $h_{j+1,j} < \eta\rho \wedge k < 3$
    $k = k + 1$
    $\boxed{\begin{array}{l} \rho = \|v_{j+1}\|_2 \\ c = V_j^* v_{j+1} \end{array}}$
    $v_{j+1} = v_{j+1} - V_j c$
    $h_{1:j,j} = h_{1:j,j} + c$
    $h_{j+1,j} = \sqrt{\rho^2 - \sum_{i=1}^{j} c_i^2}$
  fin
  si $h_{j+1,j} < \eta\rho$ **parar**
  $v_{j+1} = v_{j+1}/h_{j+1,j}$
fin

similar se ha aplicado en el contexto del gradiente conjugado con precondicionado, para reducir el número de sincronizaciones por iteración [Saa89, Meu87].

3.1.3. Normalización retrasada

Esta técnica fue originalmente propuesta por Kim y Chronopoulos [KC92] en el contexto de un método de Arnoldi sin refinamiento. La idea básica es retrasar la normalización (incluyendo el cálculo de la norma) moviéndola desde el final de la iteración actual a la siguiente, justo después del producto matriz vector. Esto permite combinar la multireducción del cálculo de la norma con las comunicaciones de la ortogonalización. Como resultado se obtiene un método de Arnoldi modificado, donde la expansión de la base se realiza con un vector no normalizado. Esto no resulta problemático siempre que todos los cálculos se corrijan a posteriori con la norma. Esta técnica no parece compatible con el criterio de refinamiento iterativo presentado anteriormente.

3.1.4. Refinamiento retrasado

La idea básica del refinamiento retrasado es similar a la normalización retrasada y ha sido presentada en [HRT06b]. En este caso, lo que se retrasa hasta la siguiente iteración es el refinamiento de la ortogonalización. De esta forma se pueden combinar las comunicaciones de la primera ortogonalización del vector v_{j+1} con el refinamiento del vector v_j. Al igual que la anterior, esta técnica no parece compatible con el criterio de refinamiento iterativo, y se obtiene un método de Arnoldi modificado.

3.1.5. Refinamiento retrasado y normalización retrasada

Como el número de comunicaciones utilizando refinamiento iterativo no puede reducirse más, podría ocurrir que un algoritmo con un paso de refinamiento incondicional, implementado en una sola multireducción, resultase más rápido con un gran número de procesadores. Combinando las dos técnicas presentadas en los puntos anteriores, es posible construir un método de Arnoldi que realice la ortogonalización de Gram-Schmidt clásica refinada con solamente una operación de multireducción. Aunque la idea es relativamente sencilla, su implementación es bastante complicada, porque cada paso del algoritmo está trabajando simultáneamente con datos de tres iteraciones consecutivas del método de Arnoldi. Además, este método modificado tendrá características

numéricas diferentes del original porque se utilizan aproximaciones para hacer avanzar al algoritmo. Esta técnica ha sido presentada en [HRT07c].

El resultado de aplicar estas técnicas es el algoritmo 3.4, Arnoldi con Gram-Schmidt clásico, refinamiento retrasado y normalización retrasada. En este algoritmo, la expansión de la base se realiza tan pronto como se dispone de un vector en la dirección *correcta*, y el refinamiento y normalización se retrasan hasta las iteraciones posteriores. Debido a esto, el algoritmo tiene que hacer cálculos con vectores aproximados tal y como se explica a continuación.

En una iteración determinada j, el algoritmo trabaja con los siguientes vectores: $v_1, v_2, \ldots, v_{j-2}$, que son los vectores de Arnoldi definitivos, esto es, han sido refinados y normalizados; w_{j-1}, que solamente necesita ser normalizado para ser el $(j-1)$-ésimo vector de Arnoldi; w_j, que necesita refinamiento y normalización; u_j, que es una versión normalizada de w_j y por tanto una aproximación a v_j; y w_{j+1}, que es la nueva dirección en la base de Arnoldi y que está empezando todo el proceso.

Para retrasar el refinamiento, se utiliza la aproximación u_j en lugar de v_j para la expansión de la base y la segunda ortogonalización de w_{j-1}. Esta aproximación se calcula normalizando w_j justo después de la primera ortogonalización, por lo que es posible que no sea completamente ortogonal a las columnas de V_{j-1}. Como consecuencia de esto, el nivel de ortogonalidad entre los vectores de Arnoldi no es tan buena como en los algoritmos anteriores. Además, la normalización de u_j está también retrasada, y se utiliza en su lugar el vector w_j para calcular w_{j+1} y $h_{1:j,j}$, por lo que estos valores deben ser corregidos tan pronto como la norma de w_j esté disponible. Si $\rho = \|w_j\|_2$ y $u_j = \rho^{-1} w_j$, entonces la corrección de w_{j+1} es

$$w'_{j+1} = A u_j = A\rho^{-1} w_j = \rho^{-1} A w_j = \rho^{-1} w_{j+1} , \tag{3.10}$$

y la corrección de $h_{1:j,j}$ es

$$h'_{1:j,j} = \left[V_{j-2}, u_{j-1}, u_j \right]^* w'_{j+1} . \tag{3.11}$$

Expandiendo esta última ecuación se obtienen las correcciones siguientes,

$$\begin{aligned}
h'_{1:j-2,j} &= V_{j-2}^* w'_{j+1} = V_{j-2}^* \rho^{-1} w_{j+1} = \rho^{-1} V_{j-2}^* w_{j+1} = \rho^{-1} h_{1:j-2,j} , \\
h'_{j-1,j} &= u_{j-1}^* w'_{j+1} = u_{j-1}^* \rho^{-1} w_{j+1} = \rho^{-1} u_{j-1}^* w_{j+1} = \rho^{-1} h_{j-1,j} , \\
h'_{j,j} &= u_j^* w'_{j+1} = \left[\rho^{-1} w_j \right]^* \rho^{-1} w_{j+1} = \rho^{-2} w_j^* w_{j+1} = \rho^{-2} h_{j,j} .
\end{aligned}$$

Algoritmo 3.4 Iteración de Arnoldi con Gram-Schmidt clásico, refinamiento retrasado y normalización retrasada

Dado un vector v_1 de norma unidad
para $j = 1, 2, \ldots, m$
  $w_{j+1} = Aw_j$
  $\boxed{h_{1:j,j} = \left[V_{j-2}, u_{j-1}, w_j\right]^* w_{j+1}}$
  si $j > 1$
    $\boxed{\rho = \|w_j\|_2}$
    $u_j = \rho^{-1} w_j$
    $w_{j+1} = \rho^{-1} w_{j+1}$
    $h_{1:j-1,j} = \rho^{-1} h_{1:j-1,j}$
    $h_{j,j} = \rho^{-2} h_{j,j}$
  fin
  $w_{j+1} = w_{j+1} - \left[V_{j-2}, u_{j-1}, u_j\right] h_{1:j,j}$
  si $j > 1$
    $\boxed{c_{1:j-1,j-1} = \left[V_{j-2}, u_{j-1}\right]^* w_j}$
    $h_{1:j-1,j-1} = h_{1:j-1,j-1} + c_{1:j-1,j-1}$
    $w_j = w_j - \left[V_{j-2}, u_{j-1}\right] c_{1:j-1,j-1}$
  fin
  si $j > 2$
    $\boxed{h_{j-1,j-2} = \|w_{j-1}\|_2}$
    $v_{j-1} = w_{j-1} / h_{j-1,j-2}$
  fin
fin
$h_{m,m-1} = \|w_m\|_2$
$v_m = w_m / h_{m,m-1}$
$c_{1:m,m} = V_m^* w_{m+1}$
$w_{m+1} = w_{m+1} - V_m c_{1:m,m}$
$h_{1:m,m} = h_{1:m,m} + c_{1:m,m}$
$h_{m+1,m} = \|w_{m+1}\|_2$
$v_{m+1} = w_{m+1} / h_{m+1,m}$

En el algoritmo 3.4 se pueden implementar todas las operaciones representadas con un borde en una sola operación de multireducción. El resto de operaciones, a excepción del producto matriz vector, no necesitan comunicaciones de ningún tipo. Esto crea una *pipeline* donde varios productos escalares, correspondientes a diferentes vectores de Arnoldi, se calculan simultáneamente sin dependencias de datos entre ellos. De esta forma, se ortogonaliza por primera vez w_{j+1}, se ortogonaliza por segunda vez w_j y se normaliza w_{j-1} a la vez, mientras se usa la aproximación u_j en lugar de v_j hasta que se dispone del valor exacto. Las operaciones que aparecen al final del bucle terminan el proceso de los últimos vectores.

3.2. Método de Lanczos

El método de Lanczos simétrico puede considerarse como un caso particular del método de Arnoldi cuando la matriz es Hermitiana. En este caso la matriz H_m se convierte en una matriz tridiagonal real y simétrica,

$$T_m = \begin{bmatrix} \alpha_1 & \beta_1 & & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \beta_{m-2} & \alpha_{m-1} & \beta_{m-1} & \\ & & & \beta_{m-1} & \alpha_m & \end{bmatrix}. \tag{3.12}$$

A partir de un vector inicial v_1 y $\beta_0 = 0$ la recurrencia

$$\beta_j v_{j+1} = Av_j - \alpha_j v_j - \beta_{j-1} v_{j-1}, \tag{3.13}$$

con $\alpha_j = v_j^* A v_j$ y $\beta_j = v_{j+1}^* A v_j$ genera una matriz V_m cuyas columnas son los vectores de Lanczos $v_1, v_2, \ldots, v_m$ y que cumple

$$AV_m - V_m T_m = \beta_m v_{m+1} e_m^*, \tag{3.14}$$

expresión análoga a la ecuación 3.3 del método de Arnoldi. La relación entre los dos métodos se puede observar claramente en el algoritmo 3.5 donde los cálculos realizados con v_{j+1} pueden verse como una ortogonalización con respecto a v_{j-1} y v_j mediante Gram-Schmidt modificado. En aritmética exacta v_{j+1} es ortogonal a $v_1, v_2, \ldots, v_{j-2}$ por construcción, y por lo tanto no es necesario incluir estos vectores en el procedimiento de Gram-Schmidt, ahorrando muchas operaciones

Algoritmo 3.5 Iteración de Lanczos

Dado un vector v_1 de norma unidad
$\beta_0 = 0$
para $j = 1, 2, \ldots, m$
    $v_{j+1} = Av_j$
    $v_{j+1} = v_{j+1} - \beta_{j-1} v_{j-1}$
    $\alpha_j = v_j^* v_{j+1}$
    $v_{j+1} = v_{j+1} - \alpha_j v_j$
    $\beta_j = \|v_{j+1}\|_2$
    si $\beta_j = 0$ **parar**
    $v_{j+1} = v_{j+1}/\beta_j$
fin

con respecto del método de Arnoldi. El análisis numérico en [Pai72, Pai80] propone este algoritmo como el más estable de las diferentes alternativas para el cálculo de α_j y β_j.

3.2.1. Pérdida de ortogonalidad

El gran inconveniente del método de Lanczos es que su implementación en coma flotante pierde la ortogonalidad entre los vectores debido a errores de redondeo. Esto hace que el método calcule valores propios repetidos o espúreos. La solución más obvia es ortogonalizar v_{j+1} explícitamente con respecto a todos los vectores de Lanczos. Esta alternativa se denomina reortogonalización *completa*. En este caso el método realiza exactamente las mismas operaciones que Arnoldi, salvo por el hecho de que la matriz proyectada se puede considerar como tridiagonal simétrica en los cálculos.

Otra alternativa [CW85] consiste en ignorar totalmente la pérdida de ortogonalidad, y realizar un postproceso para eliminar los valores repetidos o espúreos. Los inconvenientes de esta variante son que no es capaz de establecer la multiplicidad de los valores propios, y que necesita muchas iteraciones para obtener un pequeño número de valores propios distintos. Esta alternativa se conoce como reortogonalización *local*, y si se utiliza Gram-Schmidt con refinamiento, como reortogonalización *local extendida*.

El análisis realizado por Paige [Pai72, Pai76, Pai80] muestra que la pérdida de ortogonalidad se produce cuando un valor de Ritz está cerca de converger, dando lugar a una familia de métodos que aprovechan este fenómeno para ortogonalizar explícitamente solamente cuando sea necesario. El análisis realizado en [Sim84a] muestra que el método de Lanczos funciona correctamente si $\|V^*V - I\|_2 \leq \sqrt{\epsilon}$, es decir, si la ortogonalidad entre los vectores de Lanczos es del orden de la raíz cuadrada de la precisión de la máquina. Se han propuesto varios métodos, llamados semiortogonales, que aprovechan estas características para ahorrar operaciones con respecto de la reortogonalización completa. La diferencia entre estos métodos está en el criterio para realizar la reortogonalización.

Selectiva: Se reortogonaliza con respecto de los vectores de Ritz cuya norma del residuo sea menor que $\sqrt{\epsilon}\|A\|_2$ [PS79].

Periódica: Se reortogonaliza con respecto de todos los vectores de Lanczos cuando se detecta que el nivel de ortogonalidad ha superado $\sqrt{\epsilon}$. Para estimar el nivel de ortogonalidad se utiliza una recurrencia sencilla de calcular [Grc81].

Parcial: Se reortogonaliza con respecto de un subconjunto de los vectores de Lanczos cuando se detecta la misma condición que en el caso anterior [Sim84b].

Un esquema general de estos métodos se muestra en el algoritmo 3.6, donde la diferencia entre las distintas variantes es la forma de seleccionar el conjunto $\mathscr{R}$ que contiene los vectores a utilizar en la reortogonalización. La ortogonalización puede realizarse aprovechando las técnicas desarrolladas en las sección 3.1 para el método de Arnoldi, en particular el procedimiento de Gram-Schmidt clásico con refinamiento y estimación de la norma. Además, como es problemático detectar que β_j se anula en coma flotante, esta técnica puede ser aprovechada para comprobar si v_{j+1} es una combinación lineal de los vectores anteriores y parar el método.

3.2.2. Mantenimiento de la simetría para problemas generalizados

Los problemas de valores propios generalizados $Ax = \lambda Bx$ con A Hermitiana y B Hermitiana definida positiva se pueden resolver con el método de Lanczos combinado con una transformación espectral. La única modificación necesaria

Algoritmo 3.6 Iteración de Lanczos semiortogonal

Dado un vector v_1 de norma unidad
$\beta_0 = 0$
para $j = 1, 2, \ldots, m$
    $v_{j+1} = Av_j$
    $v_{j+1} = v_{j+1} - \beta_{j-1} v_{j-1}$
    $\alpha_j = v_j^* v_{j+1}$
    $v_{j+1} = v_{j+1} - \alpha_j v_j$
    Determinar conjunto $\mathscr{R} \subseteq \{v_1, v_2, \ldots, v_{j-2}\}$
    si $\mathscr{R} \neq \emptyset$
        Ortogonalizar v_{j+1} con respecto de $\mathscr{R}$
    fin
    $\beta_j = \|v_{j+1}\|_2$
    si $\beta_j = 0$ **parar**
    $v_{j+1} = v_{j+1}/\beta_j$
fin

Algoritmo 3.7 Iteración de Lanczos con desplazamiento e inversión para problemas generalizados

Dado un vector v_1 de norma unidad
$\beta_0 = 0$
para $j = 1, 2, \ldots, m$
    $v_{j+1} = (A - \sigma B)^{-1} B v_j$
    $v_{j+1} = v_{j+1} - \beta_{j-1} v_{j-1}$
    $\alpha_j = v_j^* B v_{j+1}$
    $v_{j+1} = v_{j+1} - \alpha_j v_j$
    $\beta_j = \sqrt{v_{j+1}^* B v_{j+1}}$
    si $\beta_j = 0$ **parar**
    $v_{j+1} = v_{j+1}/\beta_j$
fin

al método original es cambiar el producto interior, de forma que el resultado de esta transformación sea un operador Hermitiano con respecto de dicho producto interior. Por ejemplo, el algoritmo 3.7 muestra el caso de la transformación de desplazamiento e inversión $(A-\sigma B)^{-1}Bx = \theta x$, donde el nuevo operador es simétrico con respecto del producto interior definido por la matriz B, y por lo tanto todos los productos interiores del método se realizan de la forma u^*Bv.

Si la matriz que define el producto interior es semidefinida positiva, esta técnica continúa siendo aplicable pero pueden aparecer componentes no deseadas en los vectores de Lanczos. Estas componentes corresponden a los vectores propios asociados a valores propios nulos de la matriz que define el producto interior, y que el procedimiento de ortogonalización no puede eliminar. Sin embargo, estas componentes no afectan al funcionamiento del método y solamente es necesario corregir este problema al final del proceso, técnica conocida como *purificación* [NOPEJ87, MS97]. La forma más sencilla de esta técnica es aplicar a los vectores convergidos el operador de la transformación espectral, de forma que anule las componentes no deseadas. Para un mejor funcionamiento del método es también recomendable aplicar esta corrección al vector inicial. Este método de Lanczos para problemas generalizados tiene la ventaja de evitar la convergencia a los posibles valores propios infinitos, lo que puede resultar problemático en una implementación en coma flotante.

3.3. Método de Golub-Kahan-Lanczos

La bidiagonalización de una matriz A fue propuesta en [GK65] como una forma de tridiagonalizar el producto A^*A, sin necesidad de construirlo explícitamente, para el cálculo de la descomposición de valores singulares

$$A = U\Sigma V^* \,. \tag{3.15}$$

La bidiagonalización consiste en construir una descomposición

$$A = PBQ^*, \tag{3.16}$$

donde P y Q son matrices unitarias y B es bidiagonal superior de orden $m \times n$. A partir de la descomposición de valores singulares de B

$$B = X\Sigma Y^*, \tag{3.17}$$

se puede obtener la descomposición de valores singulares de A como

$$A = PX\Sigma Y^*Q^*, \tag{3.18}$$

donde $U = PX$ y $V = QY$.

Prescindiendo de las filas de B que solamente contienen ceros en la ecuación 3.16 se obtiene

$$A = P_n B_n Q_n^*, \tag{3.19}$$

donde P_n es ahora una matriz $m \times n$ con columnas ortonormales, Q_n es una matriz unitaria de orden n y B_n es una matriz cuadrada con la siguiente estructura

$$B_n = P_n^* A Q_n = \begin{bmatrix} \alpha_1 & \beta_1 & & & & \\ & \alpha_2 & \beta_2 & & & \\ & & \alpha_3 & \beta_3 & & \\ & & & \ddots & \ddots & \\ & & & & \alpha_{n-1} & \beta_{n-1} \\ & & & & & \alpha_n \end{bmatrix}. \tag{3.20}$$

Los coeficientes de esta matriz son reales y se obtienen como $\alpha_j = p_j^* A q_j$ y $\beta_j = p_j^* A q_{j+1}$, donde p_j y q_j son las columnas de P_n y Q_n respectivamente. Es posible derivar dos fórmulas recurrentes para calcular estos coeficientes conjuntamente con los vectores p_j y q_j, dado que q_1 determina de forma única las columnas de P_n y Q_n (aparte de un cambio de signo y bajo la suposición de que A es de rango completo y B_n es irreducible).

Multiplicando por la izquierda la ecuación 3.20 por P_n, se obtiene la relación $AQ_n = P_n B_n$. Además, si se transponen los dos lados de la ecuación 3.20 y la multiplicamos por la izquierda por Q_n, se obtiene $A^* P_n = Q_n B_n^*$. Igualando las primeras k columnas de ambas relaciones resulta

$$AQ_k = P_k B_k, \tag{3.21}$$

$$A^* P_k = Q_k B_k^* + \beta_k q_{k+1} e_k^*, \tag{3.22}$$

donde B_k es la submatriz $k \times k$ superior izquierda de B_n. Se pueden obtener expresiones análogas en forma de vector igualando solamente la columna j

$$Aq_j = \beta_{j-1} p_{j-1} + \alpha_j p_j, \tag{3.23}$$

$$A^* p_j = \alpha_j q_j + \beta_j q_{j+1}. \tag{3.24}$$

Estas expresiones proporcionan las recurrencias

$$\alpha_j p_j = Aq_j - \beta_{j-1} p_{j-1},$$
$$\beta_j q_{j+1} = A^* p_j - \alpha_j q_j,$$

Algoritmo 3.8 Iteración de Golub-Kahan-Lanczos

Dado un vector q_1 de norma unidad
$\beta_0 = 0$
para $j = 1, 2, \ldots, m$
$p_j = Aq_j - \beta_{j-1} p_{j-1}$
$\alpha_j = \|p_j\|_2$
$p_j = p_j / \alpha_j$
$q_{j+1} = A^* p_j - \alpha_j q_j$
$\beta_j = \|q_{j+1}\|_2$
si $\beta_j = 0$ **parar**
$q_{j+1} = q_{j+1} / \beta_j$
fin

con $\alpha_j = \|Aq_j - \beta_{j-1} p_{j-1}\|_2$ y $\beta_j = \|A^* p_j - \alpha_j q_j\|_2$ dado que las columnas de P_n y Q_n están normalizadas. A partir de estas recurrencias se obtiene la iteración de Golub-Kahan-Lanczos (algoritmo 3.8).

Las ecuaciones 3.21 y 3.22 se pueden combinar si se multiplica por la izquierda la primera por A^*, resultando

$$A^* A Q_k = Q_k B_k^* B_k + \alpha_k \beta_k q_{k+1} e_k^*, \tag{3.25}$$

donde la matriz $B_k^* B_k$ es tridiagonal simétrica definida positiva. Comparando este último resultado con la ecuación 3.14, se puede concluir que el algoritmo 3.8 calcula la misma información que la tridiagonalización de Lanczos (algoritmo 3.5) aplicada a la matriz A^*A. En particular los vectores q_j forman una base ortonormal del subespacio de Krylov

$$\mathcal{K}_k(A^*A, q_1) \equiv \left\langle q_1, A^*Aq_1, \ldots, (A^*A)^{k-1} q_1 \right\rangle. \tag{3.26}$$

Otra forma de combinar las ecuaciones 3.21 y 3.22 es multiplicar la segunda por la izquierda por A, resultando

$$AA^* P_k = P_k B_k B_k^* + \beta_k A q_{k+1} e_k^*. \tag{3.27}$$

A primera vista no parece una descomposición de Lanczos porque el vector Aq_{k+1} no es ortogonal a P_k. Sin embargo, utilizando la ecuación 3.23 se obtiene

$$\begin{aligned} AA^*P_k &= P_k B_k B_k^* + \beta_k^2 p_k e_k^* + \beta_k \alpha_{k+1} p_{k+1} e_k^* \\ &= P_k (B_k B_k^* + \beta_k^2 e_k e_k^*) + \beta_k \alpha_{k+1} p_{k+1} e_k^*, \end{aligned}$$

donde la matriz $B_k B_k^* + \beta_k^2 e_k e_k^*$ es también tridiagonal simétrica definida positiva. Por lo tanto se puede llegar a la misma conclusión para la matriz AA^* y los vectores p_j forman una base del subespacio de Krylov $\mathscr{K}_k(AA^*, p_1)$.

Hay otra forma alternativa de deducir el algoritmo 3.8, que afianza la relación entre esta bidiagonalización y la tridiagonalización de Lanczos. La idea es aplicar el algoritmo 3.5 a la matriz cíclica $H(A)$ definida en la sección 1.2.1 con el vector inicial

$$z_1 = \begin{bmatrix} 0 \\ q_1 \end{bmatrix}. \tag{3.28}$$

Se puede demostrar que los vectores generados son

$$z_{2j-1} = \begin{bmatrix} 0 \\ q_j \end{bmatrix} \quad \text{y} \quad z_{2j} = \begin{bmatrix} p_j \\ 0 \end{bmatrix}, \tag{3.29}$$

y que la matriz proyectada después de $2k$ iteraciones es

$$T_{2k} = \begin{bmatrix} 0 & \alpha_1 & & & & & \\ \alpha_1 & 0 & \beta_1 & & & & \\ & \beta_1 & 0 & \alpha_2 & & & \\ & & \alpha_2 & 0 & \beta_2 & & \\ & & & \beta_2 & 0 & \ddots & \\ & & & & \ddots & \ddots & \alpha_k \\ & & & & & \alpha_k & 0 \end{bmatrix}, \tag{3.30}$$

por lo que dos iteraciones de este procedimiento calculan la misma información que un paso del algoritmo 3.8. Es sencillo demostrar que hay una permutación que transforma T_{2k} a la matriz $H(B_k)$. Un aspecto importante del algoritmo 3.8 es que utiliza la mitad de espacio de almacenamiento que el algoritmo 3.5 aplicado a $H(A)$, para obtener una descomposición equivalente.

Debido a estas equivalencias, todas las propiedades de convergencia del algoritmo 3.5 se aplican a la bidiagonalización con el algoritmo 3.8. En particular,

se pueden calcular de forma fácil las estimaciones de error para las aproximaciones de Ritz. Después de k pasos de la bidiagonalización de Lanczos, los valores singulares de B_k son los valores de Ritz $\tilde{\sigma}_i$ (valores singulares aproximados de A), y los vectores de Ritz son

$$\tilde{u}_i = P_k x_i, \qquad \tilde{v}_i = Q_k y_i, \tag{3.31}$$

donde x_i e y_i son los vectores singulares por la izquierda y derecha de B_k. Con estas definiciones y las ecuaciones 3.21–3.22 es fácil demostrar que

$$A\tilde{v}_i = \tilde{\sigma}_i \tilde{u}_i, \qquad A^* \tilde{u}_i = \tilde{\sigma}_i \tilde{v}_i + \beta_k q_{k+1} e_k^* x_i. \tag{3.32}$$

Si se define la norma del residuo asociada a un valor singular aproximado $\tilde{\sigma}_i$ y sus vectores asociados $\tilde{u}_i$ y $\tilde{v}_i$ como

$$\|r_i\|_2 = \left(\|A\tilde{v}_i - \tilde{\sigma}_i \tilde{u}_i\|_2^2 + \|A^* \tilde{u}_i - \tilde{\sigma}_i \tilde{v}_i\|_2^2 \right)^{\frac{1}{2}}, \tag{3.33}$$

entonces se puede calcular fácilmente como

$$\|r_i\|_2 = \beta_k |e_k^* x_i|. \tag{3.34}$$

La equivalencia entre los dos algoritmos también incluye los problemas de pérdida de ortogonalidad del método de Lanczos. La solución más sencilla es aplicar la reortogonalización completa (algoritmo 3.9), donde la ortogonalización puede realizarse aprovechando las técnicas desarrolladas en las secciones 3.1.1 y 3.1.2 para el método de Arnoldi.

Para evitar el elevado coste de la reortogonalización completa, una posibilidad es ignorar la pérdida de ortogonalidad y realizar un postproceso para eliminar los valores singulares repetidos y/o espúreos [CWL83]. Otra posibilidad es adaptar las estrategias de reortogonalización descritas en la sección 3.2.1. Por ejemplo, se puede aplicar una reortogonalización parcial tal y como se describe en [Lar98].

3.3.1. Ortogonalización por un lado

Para mitigar el coste de la reortogonalización se propone en [SZ00] mantener la ortogonalidad solamente en uno de los conjuntos de vectores P o Q, ya que la ortogonalidad del otro no influye en la convergencia hacia los valores singulares dominantes. Por ejemplo, si se decide mantener la ortogonalidad con los

Algoritmo 3.9 Iteración de Golub-Kahan-Lanczos con reortogonalización completa

Dado un vector q_1 de norma unidad
para $j = 1, 2, \ldots, m$
  $p_j = Aq_j$
  Ortogonalizar p_j con respecto de P_{j-1}
  $\alpha_j = \|p_j\|_2$
  $p_j = p_j/\alpha_j$
  $q_{j+1} = A^* p_j$
  Ortogonalizar q_{j+1} con respecto de Q_j
  $\beta_j = \|q_{j+1}\|_2$
  si $\beta_j = 0$ **parar**
  $q_{j+1} = q_{j+1}/\beta_j$
fin

Algoritmo 3.10 Iteración de Golub-Kahan-Lanczos con reortogonalización completa por un lado

Dado un vector q_1 de norma unidad
$\beta_0 = 0$
para $j = 1, 2, \ldots, m$
  $p_j = Aq_j - \beta_{j-1} p_{j-1}$
  $\alpha_j = \|p_j\|_2$
  $p_j = p_j/\alpha_j$
  $q_{j+1} = A^* p_j$
  Ortogonalizar q_{j+1} con respecto de Q_j
  $\beta_j = \|q_{j+1}\|_2$
  si $\beta_j = 0$ **parar**
  $q_{j+1} = q_{j+1}/\beta_j$
fin

Algoritmo 3.11 Iteración de Golub-Kahan-Lanczos con reortogonalización completa por un lado y retraso de la normalización

Dado un vector q_1 de norma unidad
$\beta_0 = 0$
para $j = 1, 2, \ldots, m$
  $p_j = Aq_j - \beta_{j-1} p_{j-1}$
  $q_{j+1} = A^* p_j$
  $\boxed{\begin{array}{l} \alpha_j = \|p_j\|_2 \\ \rho = \|q_{j+1}\|_2 \\ c = Q_j^* q_{j+1} \end{array}}$
  $p_j = p_j / \alpha_j$
  $q_{j+1} = q_{j+1} / \alpha_j$
  $\rho = \rho / \alpha_j$
  $c = c / \alpha_j$
  $q_{j+1} = q_{j+1} - Q_j c$
  $\beta_j = \sqrt{\rho^2 - \sum_{i=1}^{j} c_i^2}$
  $k = 1$
  mientras $\beta_j < \eta \rho \wedge k < 3$
    $k = k + 1$
    $\boxed{\begin{array}{l} \rho = \|q_{j+1}\|_2 \\ c = Q_j^* q_{j+1} \end{array}}$
    $q_{j+1} = q_{j+1} - Q_j c$
    $\beta_j = \sqrt{\rho^2 - \sum_{i=1}^{j} c_i^2}$
  fin
  si $\beta_j < \eta \rho$ **parar**
  $q_{j+1} = q_{j+1} / \beta_j$
fin

vectores Q se obtiene el algoritmo 3.10, donde se reduce a la mitad el coste de la reortogonalización completa.

A este algoritmo se le puede aplicar una técnica parecida al retraso de la normalización, presentado en la sección 3.1.3, obteniendo el algoritmo 3.11. Al retrasar la normalización de p_j es necesario corregir con el valor α_j al vector q_{j+1} y los resultados ρ y c obtenidos a partir de éste. En este algoritmo se utiliza también la ortogonalización clásica de Gram-Schmidt con refinamiento iterativo (sección 3.1.1) y estimación de la norma (sección 3.1.2), de forma que se consigue implementar el método con una única multireducción para las operaciones dentro de cada marco.

3.4. Técnicas de reinicio

Un inconveniente de los métodos de Arnoldi y Lanczos con reortogonalización es que su coste espacial crece con el número de iteraciones, puesto que es necesario mantener en memoria un vector de Arnoldi/Lanczos por cada iteración. El coste temporal resulta todavía más importante dado que crece de forma cuadrática con el número de vectores calculados. Para reducir estos costes se pueden aplicar técnicas de reinicio, en donde se garantiza no usar más de un número limitado de vectores.

Las técnicas de reinicio son también interesantes para modificar el comportamiento de los métodos de Krylov, que de formal natural convergen hacia los valores propios de los extremos del espectro.

3.4.1. Reinicio explícito

El reinicio explícito es la técnica más simple, consistiendo en parar el proceso tras m iteraciones y volver a empezar con un vector de inicio calculado en base a los resultados actuales. Una forma sencilla de obtener el nuevo vector de inicio, es utilizar el vector de Ritz asociado al valor de Ritz más cercano a los valores propios buscados. Otra posibilidad es utilizar una combinación lineal de los vectores de Ritz. También se puede aplicar un polinomio en A al vector inicial, para filtrar las componentes en las direcciones no deseadas, donde dicho polinomio se construye a partir de los valores de Ritz disponibles [Saa84].

Para que un método con reinicio sea capaz de encontrar distintos valores propios, es necesario hacer deflación con los vectores convergidos en etapas anteriores. Esto se realiza con una técnica denominada *bloqueo*, en la cual los

Algoritmo 3.12 Método de Arnoldi con reinicio explícito

Dado un vector v_1 de norma unidad
$k = 0$
repetir
  para $j = k+1, k+2, \ldots, m$
    $u = Av_j$
    Ortogonalizar u con respecto de V_j obteniendo $h_{1:j,j}$
    $h_{j+1,j} = \|u\|_2$
    $v_{j+1} = u/h_{j+1,j}$
  fin
  Calcular la forma real de Schur $H_m = U_m S_m U_m^*$
  Ordenar S_m y U_m colocando los valores deseados al principio de S_m
  Calcular la estimación del residuo, $\tau_i = h_{m+1,m}|e_m^* y_i|$, donde $H_m y_i = \theta_i y_i$
  Bloquear vectores convergidos ajustando k
  $H_m = S_m$
  $V_m = V_m U_m$
hasta $\langle convergencia \rangle$

vectores asociados a valores propios convergidos no se modifican en sucesivas iteraciones. Suponiendo que, tras m iteraciones de Arnoldi, los primeros k valores y vectores propios han convergido a la precisión requerida, entonces escribimos V_m como

$$V_m = \left[\begin{array}{c|c} V_{1:k}^{(l)} & V_{k+1:m}^{(a)} \end{array} \right], \tag{3.35}$$

donde el superíndice (l) denota los vectores bloqueados y el superíndice (a) indica los vectores activos. En las siguientes iteraciones de Arnoldi, solamente se calcularán $m-k$ nuevos vectores que reemplazarán a los activos. Para realizar la deflación, estos nuevos vectores se ortogonalizan con los k vectores bloqueados.

El algoritmo 3.12 muestra el esquema del reinicio explícito para el método de Arnoldi. En cada iteración del bucle externo la matriz H_m tiene las k primeras columnas en forma triangular superior (excepto para los valores propios complejos

conjugados) y el resto en forma de Hessenberg superior,

$$H_m = \left[\begin{array}{ccccc|ccccc} \times & \times & \times & \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times & \times & \times & \times \\ & & & \times & \times & \times & \times & \times & \times \\ \hline & & & & \times & \times & \times & \times & \times \\ & & & & \times & \times & \times & \times & \times \\ & & & & & \times & \times & \times & \times \\ & & & & & & \times & \times & \times \\ & & & & & & & \times & \times \end{array}\right]. \tag{3.36}$$

La submatriz $H_{1:k,1:k}$ y los vectores $V_{1:k}$ forman una descomposición parcial de Schur de A,

$$AV_{1:k} = V_{1:k}H_{1:k,1:k}, \tag{3.37}$$

donde los elementos diagonales $h_{1,1}, h_{2,2}, \ldots, h_{k,k}$ son los valores propios convergidos y $v_1, v_2, \ldots, v_k$ sus vectores de Schur asociados. En este algoritmo se utiliza la forma real de Schur, que permite almacenar los valores propios complejos conjugados en un bloque diagonal 2×2 utilizando únicamente aritmética real.

Los valores propios calculados en cada reinicio se ordenan a partir de la posición $k+1$ mediante rotaciones [BD93]. Esta ordenación debe colocar los valores propios deseados al principio de la descomposición; por ejemplo, para encontrar los valores propios dominantes hay que ordenar de mayor a menor valor absoluto.

Antes de reiniciar el proceso se calculan los nuevos vectores de Schur aproximados, y el vector de reinicio es el correspondiente al primer valor propio que no ha convergido según la ordenación elegida. Si se han obtenido los valores y vectores propios con la tolerancia deseada el algoritmo finaliza, condición que no se detalla en el algoritmo por claridad. Al final de todo este proceso los vectores propios se han de obtener a partir de los vectores de Schur convergidos.

La aplicación del reinicio explícito es mucho más sencilla en el método de Lanczos simétrico (algoritmo 3.13), puesto que la parte bloqueada de H_m es una matriz diagonal cuyos elementos diagonales son iguales a los valores propios. Además, los vectores de Schur son directamente los vectores propios.

La técnica de bloqueo permite superar la limitación del método de Lanczos sin reinicio, que no permite calcular valores propios múltiples. Al bloquear los vectores convergidos, la deflación garantiza que los valores propios repetidos,

Algoritmo 3.13 Método de Lanczos con reinicio explícito

Dado un vector v_1 de norma unidad
$k = 0$
repetir
  para $j = k+1, k+2, \ldots, m$
    $u = Av_j$
    Ortogonalizar u con respecto de $\{v_1, \ldots, v_k, v_{j-1}, v_j\}$ obteniendo α_j
    Determinar conjunto $\mathscr{R} \subseteq \{v_{k+1}, v_{k+2}, \ldots, v_{j-2}\}$
    si $\mathscr{R} \neq \emptyset$
      Ortogonalizar u con respecto de $\mathscr{R}$
    fin
    $\beta_j = \|u\|_2$
    $v_{j+1} = u/\beta_j$
  fin
  Calcular los valores y vectores propios de T_m, $T_m y_i = \theta_i y_i$
  Ordenar los valores deseados al principio de la descomposición
    $T_m = Y_m \Theta_m Y_m^*$
  Calcular la estimación del residuo, $\tau_i = \beta_m |e_m^* y_i|$
  Bloquear valores y vectores propios convergidos ajustando k
  $V_m = V_m Y_m$
hasta $\langle convergencia \rangle$

que aparezcan en reinicios sucesivos, corresponden a vectores propios diferentes, y por tanto, es un valor propio múltiple.

La principal dificultad del algoritmo 3.13 es la reortogonalización necesaria para mantener la ortogonalidad entre los vectores de Lanczos. La diferencia entre los distintos esquemas de reortogonalización reside en determinar el conjunto $\mathscr{R}$ de vectores. Con $\mathscr{R} = \{v_{k+1}, v_{k+2}, \ldots, v_{j-2}\}$ en todas las iteraciones se obtiene la reortogonalización completa. Para la reortogonalización selectiva, periódica y parcial se calcula $\mathscr{R}$ con las alternativas descritas en la sección 3.2.1.

La reortogonalización local se obtiene de forma trivial con $\mathscr{R} = \emptyset$. Como m suele ser bastante pequeño la heurística descrita en [CW85] no funciona correctamente, y en este caso no queda más remedio que calcular explícitamente la norma del residuo para detectar los valores espúreos y descartar directamente los valores repetidos. Aunque esta técnica puede parecer ineficiente, la deflación mejora mucho las prestaciones de esta variante, reduciendo el número de espúreos con respecto de un método de Lanczos sin reinicio. En la práctica, calcular la norma del residuo resulta menos costoso que mantener la ortogonalidad entre los vectores de Lanczos.

La aplicación del reinicio explícito al método de Golub-Kahan-Lanczos es muy similar a la vista anteriormente para el método de Lanczos. Las diferencias más importantes son que se trabaja con dos conjuntos de vectores P y Q, y que al final del proceso se calcula la SVD de la matriz proyectada. El algoritmo 3.14 muestra este método con reortogonalización completa por los dos lados. Obviamente, el reinicio explícito también se puede aplicar al método con reortogonalización por un solo lado, aunque por brevedad ese algoritmo no se detalla en esta sección.

3.4.2. Reinicio implícito

La principal dificultad del reinicio explícito es la elección de los parámetros para construir el nuevo vector inicial. El reinicio implícito es una alternativa en la cual este vector no se construye de forma explícita. En su lugar, contrae la descomposición de Arnoldi de tamaño m

$$AV_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^*, \tag{3.38}$$

a una nueva de tamaño $p < m$ donde se conservan los vectores de Ritz asociados a los valores propios más cercanos al espectro buscado. Esto se consigue aplicando $m - p$ pasos de la iteración QR (algoritmo 1.1) en su variante de desplazamientos implícitos [GVL96, cap. 7]. La primera fase de este proceso obtiene

Algoritmo 3.14 Método de Golub-Kahan-Lanczos con reinicio explícito

Dado un vector q_1 de norma unidad
$k = 0$
repetir
  para $j = k+1, k+2, \ldots, m$
    $p_j = Aq_j$
    Ortogonalizar p_j con respecto de P_{j-1}
    $\alpha_j = \|p_j\|_2$
    $p_j = p_j/\alpha_j$
    $q_{j+1} = A^* p_j$
    Ortogonalizar q_{j+1} con respecto de Q_j
    $\beta_j = \|q_{j+1}\|_2$
    $q_{j+1} = q_{j+1}/\beta_j$
  fin
  Calcular los valores y vectores singulares de $B_m = X_m \Sigma_m Y_m^*$
  Calcular la estimación del residuo, $\tau_i = \beta_m |e_m^* x_i|$
  Bloquear valores y vectores singulares convergidos ajustando k
  $Q_m = Q_m Y_m$
  $P_m = P_m X_m$
hasta $\langle convergencia \rangle$

$$A\tilde{V}_m = \tilde{V}_m \tilde{H}_m + h_{m+1,m} v_{m+1} e_m^* Q \,, \tag{3.39}$$

donde $\tilde{V}_m = V_m Q$, $\tilde{H}_m = Q^* H_m Q$ y $Q = Q_1 Q_2 \cdots Q_{m-p}$. Cada Q_j es la matriz ortogonal generada por algoritmo QR con el desplazamiento μ_j tal que

$$QR = (H_m - \mu_1 I)(H_m - \mu_2 I) \cdots (H_m - \mu_{m-p} I) \,. \tag{3.40}$$

Debido a la estructura Hessenberg de las matrices Q_j, resulta que los primeros $p-1$ elementos del vector $e_m^* Q$ son nulos. Por lo tanto, las primeras p columnas en la ecuación 3.39 forman una descomposición de Arnoldi. Igualando las primeras p columnas en ambos lados de esta ecuación se obtiene

$$A\tilde{V}_p = \tilde{V}_p \tilde{H}_p + f_p e_p^* \,, \tag{3.41}$$

con f_p calculado como una combinación lineal de $\tilde{v}_{p+1}$ y v_{m+1}. Esta nueva descomposición se expande con una variación de la iteración de Arnoldi (algoritmo 3.1) hasta un tamaño m, repitiendo el proceso de contracción y expansión hasta obtener los valores propios buscados con la precisión requerida.

El caso más usual es que los desplazamientos μ_j sean los $m-p$ valores de Ritz más alejados de los valores propios buscados. Este proceso puede interpretarse como una aplicación de un polinomio en A de grado $m-p$ al vector inicial [Sor92]. También existen otras estrategias para la selección de los desplazamientos como puntos de Leja [BCR98], valores de Ritz armónicos y vectores de Ritz refinados [Jia02].

El reinicio implícito se puede aplicar al método de Lanczos, pero en este caso es recomendable utilizar reortogonalización completa [CRS94, BCR03a, BCR03b]. Este coste extra de mantener la ortogonalidad queda ampliamente compensado por la mejora obtenida en la velocidad de convergencia. Esta técnica también se puede aplicar al método de Golub-Kahan-Lanczos para el problema de valores singulares [Lar01].

3.4.3. Método de Krylov-Schur

La técnica de reinicio implícito, presentada en la sección anterior, resulta muy complicada de implementar de forma numéricamente estable [LS96b]. En particular, esta técnica necesita una deflación explícita para purgar los vectores de Ritz no deseados que la iteración QR no puede eliminar por problemas de estabilidad [PL93]. El método de Krylov-Schur [Ste01a] no presenta estos problemas puesto que no utiliza la iteración QR para contraer la descomposición de

Arnoldi. Este método proporciona la misma aceleración que el reinicio implícito con los valores de Ritz como desplazamientos, pero de una forma más sencilla y estable.

El método de Krylov-Schur se define generalizando la descomposición de Arnoldi de orden m,

$$AV_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^* , \tag{3.42}$$

calculada por el algoritmo 3.1, a la denominada descomposición de Krylov de orden m,

$$AV_m = V_m B_m + v_{m+1} b_{m+1}^* , \tag{3.43}$$

en donde la matriz B_m no está restringida a la forma de Hessenberg superior y b_{m+1} es un vector arbitrario.

Asumiendo que todos los vectores v_i son ortonormales entre sí, al multiplicar por la izquierda la ecuación 3.43 por V_m^* se obtiene que B_m es el cociente de Rayleigh $V_m^* A V_m$, por lo que el procedimiento de Rayleigh-Ritz es todavía válido.

La última ecuación es equivalente a

$$AV_m = \begin{bmatrix} V_m & v_{m+1} \end{bmatrix} \begin{bmatrix} B_m \\ b_{m+1}^* \end{bmatrix} . \tag{3.44}$$

Un caso especial es la descomposición de Krylov-Schur, en la que la matriz B_m está en la forma real de Schur, es decir, casi-triangular mostrando los valores propios en la diagonal o en bloques diagonales de tamaño 2×2. La figura 3.1 muestra una representación gráfica de estas factorizaciones. Por simplicidad la figura no muestra bloques 2×2.

La descomposición de Arnoldi es un caso particular de la descomposición de Krylov. Se puede demostrar [Ste01a] que para cualquier descomposición de Krylov existe una descomposición de Arnoldi equivalente, es decir, que tiene las mismas aproximaciones de Ritz. También es posible transformar una descomposición de Krylov en otra de Krylov-Schur utilizando solamente transformaciones de semejanza ortogonales.

La idea básica del método de Krylov-Schur es expandir y contraer, de forma iterativa, una descomposición de Krylov de forma parecida a la técnica de reinicio implícito. La expansión se realiza en ambos métodos con la iteración de Arnoldi (algoritmo 3.1) y la contracción se realiza con transformaciones de semejanza ortogonales.

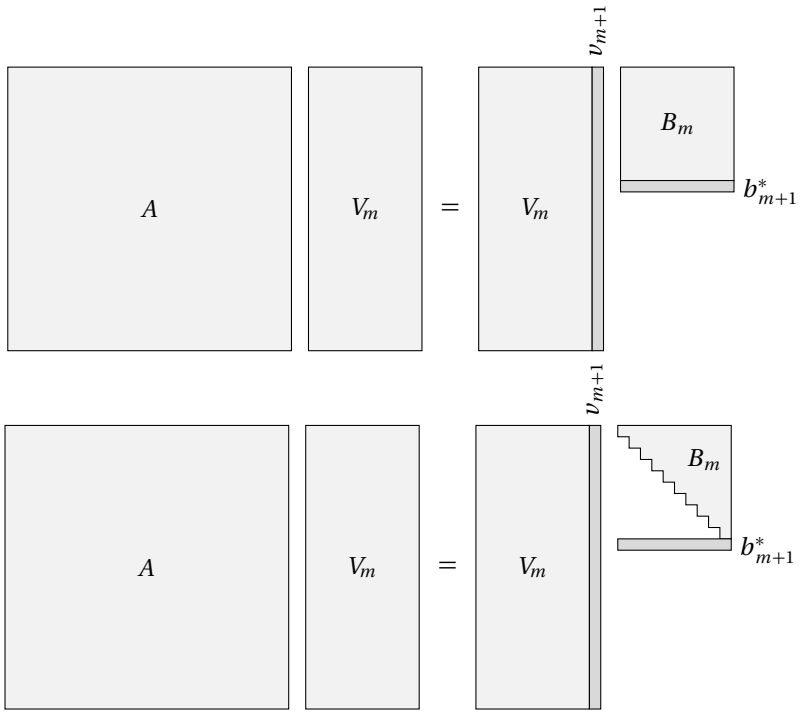

Figura 3.1 Esquema de una descomposición general de Krylov (arriba) y de una descomposición de Krylov-Schur (abajo).

Algoritmo 3.15 Método de Krylov-Schur

Entrada: Matriz A, vector inicial v_1, y número de pasos m
Salida: $k \leq p$ pares de Ritz

1. Construir una descomposición de Krylov de orden m
2. Aplicar transformaciones ortogonales para conseguir una descomposición de Krylov-Schur
3. Reordenar los bloques diagonales de la descomposición de Krylov-Schur
4. Truncar la descomposición de Krylov-Schur a otra de orden p
5. Extender a una descomposición de Krylov de orden m
6. Si no se alcanza la convergencia, volver al paso 2

Una versión esquemática del método de Krylov-Schur se muestra en el algoritmo 3.15. El paso 1 se realiza con el algoritmo 3.1 original. En el paso 2 es necesario aplicar el algoritmo QR para calcular la matriz ortogonal Q_1 tal que $S_m = Q_1^* B_m Q_1$ tenga forma real de Schur, y entonces

$$AV_m Q_1 = \left[\begin{array}{cc} V_m Q_1 & v_{m+1} \end{array} \right] \left[\begin{array}{c} S_m \\ b_{m+1}^* Q_1 \end{array} \right]. \tag{3.45}$$

En este punto, los valores de Ritz están disponibles en los bloques diagonales de S_m. Estos valores se dividen en dos subconjuntos: Ω_w que contienen $p < m$ valores de Ritz *buscados* y Ω_u que contienen $m - p$ valores de Ritz *no deseados*. El objetivo del paso 3 es mover los valores de Ritz buscados al principio de la matriz S_m mediante un transformación ortogonal Q_2, resultando en la descomposición de Krylov-Schur ordenada

$$A\tilde{V}_m = \left[\begin{array}{cc} \tilde{V}_m & v_{m+1} \end{array} \right] \left[\begin{array}{cc} S_w & \star \\ 0 & S_u \\ \hline b_w^* & \star \end{array} \right], \tag{3.46}$$

donde $\tilde{V}_m = V_m Q_1 Q_2$, $\lambda(S_w) = \Omega_w$, $\lambda(S_u) = \Omega_u$, y b_w^* es el vector con las primeras p componentes de $b_{m+1}^* Q_1 Q_2$. Gracias al bloque nulo de la ecuación anterior, el truncamiento del paso 4 del algoritmo se realiza simplemente escribiendo

$$A\tilde{V}_p = \left[\begin{array}{cc} \tilde{V}_p & \tilde{v}_{p+1} \end{array} \right] \left[\begin{array}{c} S_w \\ b_w^* \end{array} \right], \tag{3.47}$$

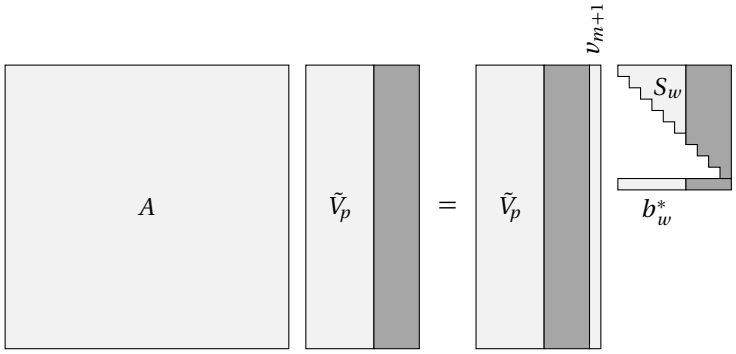

Figura 3.2 Esquema del truncamiento de la descomposición reordenada de Krylov-Schur.

Figura 3.3 Estructura de la matriz B_m después del paso 5 (antes de la reducción a forma de Krylov-Schur), para el caso no simétrico (izquierda) y el simétrico (derecha).

donde $\tilde{V}_p$ son las primeras p columnas de $\tilde{V}_m$, pero $\tilde{v}_{p+1} = v_{m+1}$. Este paso está ilustrado en la figura 3.2, donde las partes que se van a eliminar de la factorización están sombreadas.

Por último, en el paso 5 se realiza una variación de la iteración de Arnoldi (algoritmo 3.1) que empieza por el vector v_{p+2} pero se incluyen los vectores v_1 a v_{p+1} en la ortogonalización. En la izquierda de la figura 3.3 se muestra la estructura de la nueva matriz B_m, sombreando las columnas añadidas en este proceso de expansión.

En el caso que la matriz del problema sea simétrica, se tiene que B_m es también simétrica tal y como muestra la figura 3.3. Por lo tanto, no es necesario forzar la ortogonalidad con respecto de todos los vectores v_j, y se puede utilizar

una variación de la iteración de Lanczos (algoritmo 3.5) para realizar las fases de expansión. Además se pueden utilizar simples permutaciones para ordenar los valores de Ritz en lugar de transformaciones ortogonales. Este método se propuso en [WS00] con el nombre Lanczos con reinicio grueso (*thick restart Lanczos*).

El reinicio grueso también puede adaptarse a la bidiagonalización para el problema de la SVD tal y como se propone en [BR05]. La idea principal es reducir la bidiagonalización de Lanczos completa de k pasos, (3.21)–(3.22), a una nueva

$$A\tilde{Q}_{\ell+1} = \tilde{P}_{\ell+1}\tilde{B}_{\ell+1}, \tag{3.48}$$

$$A^*\tilde{P}_{\ell+1} = \tilde{Q}_{\ell+1}\tilde{B}^*_{\ell+1} + \tilde{\beta}_{\ell+1}\tilde{q}_{\ell+2}e^*_{k+1}, \tag{3.49}$$

donde el valor de $\ell < k$ puede ser, por ejemplo, el número de valores singulares buscados. El punto clave es construir la descomposición de (3.48)– (3.49) de tal forma que mantiene la información espectral relevante contenida en la descomposición completa anterior. Esto se consigue directamente colocando las primeras ℓ columnas de $\tilde{Q}_{\ell+1}$ para que sean las aproximaciones buscadas a los vectores singulares por la derecha, y de forma similar para que las columnas $\tilde{P}_{\ell+1}$ sean las correspondientes aproximaciones a los vectores singulares por la izquierda. Esta descomposición se obtiene de forma sencilla, definiendo $\tilde{Q}_{\ell+1}$ como

$$\tilde{Q}_{\ell+1} = \left[\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_\ell, q_{k+1}\right], \tag{3.50}$$

es decir, los vectores de Ritz $\tilde{v}_i = Q_k y_i$ junto con el último vector generado por el algoritmo 3.8. Las columnas de esta matriz son ortogonales debido a que $Q_k^* q_{k+1} = 0$ por construcción. De forma similar, se define $\tilde{P}_{\ell+1}$ como

$$\tilde{P}_{\ell+1} = \left[\tilde{u}_1, \tilde{u}_2, \ldots, \tilde{u}_\ell, \tilde{p}_{\ell+1}\right], \tag{3.51}$$

con $\tilde{u}_i = P_k x_i$ y $\tilde{p}_{\ell+1}$ un vector de norma unidad calculado como $\tilde{p}_{\ell+1} = f/\|f\|_2$, donde f es el vector resultado de ortogonalizar Aq_{k+1} con respecto de los primeros ℓ vectores de Ritz por la izquierda, $\tilde{u}_i$,

$$f = Aq_{k+1} - \sum_{i=1}^{\ell} \tilde{\rho}_i \tilde{u}_i. \tag{3.52}$$

Los coeficientes de la ortogonalización se pueden calcular fácilmente como $\tilde{\rho}_i = \beta_k e_k^* x_i$ (estos valores son similares a las estimaciones del residuo de la

Algoritmo 3.16 Bidiagonalización de Lanczos con reinicio grueso

Entrada: Matriz A, vector inicial q_1, y número de pasos k
Salida: $\ell \leq k$ valores y vectores singulares de Ritz

1. Construir una bidiagonalización de Lanczos de orden k
2. Calcular aproximaciones de Ritz para los valores y vectores singulares
3. Truncar la bidiagonalización de Lanczos a otra de orden ℓ
4. Extender la bidiagonalización de Lanczos a otra de orden k
5. Si no se alcanza la convergencia, volver al paso 2

ecuación 3.34, pero aquí el signo es relevante). La nueva matriz proyectada es

$$\tilde{B}_{\ell+1} = \begin{bmatrix} \tilde{\sigma}_1 & & & & \tilde{\rho}_1 \\ & \tilde{\sigma}_2 & & & \tilde{\rho}_2 \\ & & \ddots & & \vdots \\ & & & \tilde{\sigma}_\ell & \tilde{\rho}_\ell \\ & & & & \tilde{\alpha}_{\ell+1} \end{bmatrix}, \tag{3.53}$$

donde $\tilde{\alpha}_{\ell+1} = \|f\|_2$ se calcula de forma que la ecuación 3.48 se mantiene. Para completar la forma de bidiagonalización de Lanczos, solamente queda definir $\tilde{\beta}_{\ell+1}$ y $\tilde{q}_{\ell+2}$ en la ecuación 3.49, que resultan ser $\tilde{\beta}_{\ell+1} = \|g\|_2$ y $\tilde{q}_{\ell+2} = g/\|g\|_2$, donde $g = A^* \tilde{p}_{\ell+1} - \tilde{\alpha}_{\ell+1} q_{k+1}$.

Está demostrado en [BR05] que la bidiagonalización de Lanczos se mantiene si el algoritmo 3.8 se aplica para $j = \ell+2, \ldots, k$ empezando por los valores de $\tilde{\beta}_{\ell+1}$ y $\tilde{q}_{\ell+2}$ indicados anteriormente. Por lo tanto, se obtiene una nueva descomposición de tamaño completo pero la matriz proyectada ya no es bidiagonal,

$$\tilde{B}_k = \begin{bmatrix} \tilde{\sigma}_1 & & & & \tilde{\rho}_1 & & & \\ & \tilde{\sigma}_2 & & & \tilde{\rho}_2 & & & \\ & & \ddots & & \vdots & & & \\ & & & \tilde{\sigma}_\ell & \tilde{\rho}_\ell & & & \\ & & & & \tilde{\alpha}_{\ell+1} & \beta_{\ell+1} & & \\ & & & & & \ddots & \ddots & \\ & & & & & & \alpha_{k-1} & \beta_{k-1} \\ & & & & & & & \alpha_k \end{bmatrix}, \tag{3.54}$$

donde los valores sin tilde se calculan con el algoritmo 3.8. Si se realiza este proceso de forma iterativa se obtiene el algoritmo 3.16, donde el paso 4 se puede realizar con una variación del algoritmo 3.11. A partir del nuevo vector inicial, este algoritmo calcula el vector inicial por la izquierda correspondiente, $p_{\ell+1}$, y continua de la manera habitual.

Los métodos con reinicio grueso presentan los mismos problemas de ortogonalidad descritos en la sección 3.2.1 para el método de Lanczos, pero no se han desarrollado todavía alternativas robustas a la reortogonalización completa. Además, la experiencia parece sugerir que para mantener las propiedades de convergencia del método, es necesario mantener la ortogonalidad con los vectores bloqueados, haciendo poco interesantes los métodos semiortogonales o con heurísticas de postproceso.

Capítulo 4

Implementación en la librería SLEPc

Este capítulo describe la implementación en la librería SLEPc de los métodos de Krylov descritos en el capítulo anterior. Esta implementación ha dado lugar a varias clases independientes, para organizar el código en función del problema a resolver y el tipo de reinicio utilizado. Los métodos de Arnoldi y Lanczos con reinicio explícito están en clases separadas, puesto que existen bastantes diferencias entre las implementaciones del problema Hermitiano y no Hermitiano. Por el contrario, el método de Krylov-Schur está implementado en una única clase para los dos tipos de problemas. El método de Golub-Kahan-Lanczos está dividido en dos clases, una con reinicio explícito y otra con reinicio grueso.

Las clases descritas en este capítulo son formalmente independientes, pero en la práctica reutilizan gran parte del código entre ellas. En un sistema orientado a objetos, más estricto que PETSc, sería necesario utilizar una jerarquía de clases u otras técnicas avanzadas para poder implementar la misma reutilización de código. En particular, los procedimientos de ortogonalización son comunes a todos los métodos implementados. Otros ejemplos concretos, como la ordenación de la forma de Schur y construcción de vectores propios a partir de vectores de Schur, se detallan en la descripción de cada una de las clases.

La paralelización de los métodos se realiza mediante primitivas de PETSc, utilizadas para la matriz del problema y los vectores de la base del subespacio. La matriz proyectada está replicada en todos los procesadores, dado que suele ser demasiado pequeña para ser paralelizada de forma eficiente. Además,

de esta forma se pueden aprovechar las funciones de LAPACK, para hacer las transformaciones necesarias y calcular los valores y vectores de Ritz.

Los métodos implementados se han validado con una batería de pruebas, intentando reproducir el uso práctico de la librería SLEPc. Esta batería de pruebas consiste en calcular los valores propios o singulares más grandes de las matrices en las colecciones Harwell-Boeing [DGL89] y NEP [BDDD97]. Estas matrices se suelen utilizar en la literatura para validar los métodos presentados porque proceden de aplicaciones reales, suelen presentar dificultades particulares a la hora de calcular sus valores propios y están disponibles públicamente.

Para medir las prestaciones paralelas se han utilizado tres máquinas diferentes. La primera es Odin, un cluster de 55 biprocesadores Xeon a 2,8 GHz interconectados con una red SCI [IEE93] con topología de toro 2D, perteneciente al GRyCAP [1]. Para estudiar la escalabilidad con más procesadores se ha utilizado la máquina MareNostrum del Centro Nacional de Supercomputación, un cluster IBM con 2560 nodos unidos por una red Myrinet con dos procesadores de doble núcleo PowerPC 970MP a 2,3 GHz por nodo. Solamente se ha utilizado un procesador por nodo en estas dos máquinas, debido a que las prestaciones empeoran cuando se lanza más de un proceso por nodo. Este efecto se debe a la compartición del ancho de banda de memoria entre los procesadores del mismo nodo, tal y como se ha detallado en la sección 2.1.3. También se han realizado algunas pruebas en Seaborg, perteneciente al National Energy Research Scientific Computing Center, un IBM SP RS/6000 con 380 nodos y 16 procesadores POWER3 a 375 MHz por nodo. En esta máquina, actualmente desmantelada, las prestaciones no se veían afectadas si se aprovechaban todos los procesadores de cada nodo.

La medida de tiempos de ejecución en estas máquinas presenta una gran variabilidad, debido al uso compartido de los recursos entre distintos usuarios. En particular, se comparte la red de interconexión con otros procesos que se ejecutan de forma simultánea, lo que puede resultar en una pérdida de prestaciones. Este tipo de máquinas se explotan para obtener la máxima productividad, y no resulta factible realizar experimentos de prestaciones en condiciones controladas. Además, el tiempo de ejecución disponible se encuentra limitado, y no resulta posible realizar suficientes pruebas para obtener medidas estadísticas de calidad. Por lo tanto, se ha optado por realizar un número pequeño de experimentos y tomar como representativo el menor de los tiempos obtenidos, que suponemos será el más cercano a las condiciones ideales sin compartir recursos con otros

[1] `http://www.grycap.upv.es/usuario/odin.htm`

usuarios. Los experimentos se han repetido un mínimo de tres veces, pero se ha llegado hasta diez repeticiones en los casos donde los resultados no parecían estar en concordancia con las previsiones teóricas o el resto de mediciones.

El principal índice de prestaciones de los algoritmos paralelos es el *speed-up*, es decir, la aceleración que se obtiene al utilizar varios procesadores con respecto de la versión secuencial. Idealmente, el tiempo de ejecución se debe reducir en un factor p si se utilizan p procesadores. El speed-up se calcula de la forma tradicional como

$$S_p = \frac{T_s}{T_p} ,$$

donde T_s es el tiempo en secuencial del algoritmo más rápido y T_p es el tiempo en paralelo con p procesadores.

Existe un límite en las prestaciones paralelas al utilizar problemas de tamaño fijo, debido a que el tiempo empleado en la parte del algoritmo paralelizable disminuye al repartirlo entre un número mayor de procesadores, mientras que el tiempo empleado en la parte secuencial se mantiene constante. Para evitar este efecto se ha recurrido a un problema tridiagonal de dimensión proporcional al número de procesadores, con elementos aleatorios entre 0 y 1. Con esta matriz, la parte paralelizable en la implementación de SLEPc, es decir, el producto matriz vector y resto de operaciones con los vectores de la base, crece de forma casi lineal con la dimensión. Un inconveniente de trabajar con matrices de dimensión variable es que al aumentar la dimensión también aumenta el número de valores propios, haciendo que estén más juntos y por tanto necesitando más iteraciones para converger. Asumiendo que el coste computacional por iteración crece de forma lineal con la dimensión, y que el número de iteraciones es variable, definimos el speed-up escalado para p procesadores como

$$S = \frac{\frac{T_s \times p}{I_s}}{\frac{T_p}{I_p}} ,$$

donde T_p e I_p son el tiempo y número de iteraciones con p procesadores, T_s e I_s son el tiempo y número de iteraciones del algoritmo más rápido en secuencial. Otro parámetro interesante para medir la eficacia de la implementación paralela es el número de operaciones en coma flotante (Mflop) por segundo en cada procesador.

Parámetro de ejecución	Valor	Método
`-eps_orthog_type`	`mgs`	Gram-Schmidt modificado
	`cgs`	Gram-Schmidt clásico
`-eps_orthog_refinement`	`never`	sin refinamiento
	`always`	con refinamiento
	`ifneeded`	con refinamiento iterativo

Tabla 4.1 Opciones para seleccionar el procedimiento de ortogonalización.

4.1. Procedimiento de Gram-Schmidt

El procedimiento de ortogonalización es la parte más importante que es común a los métodos implementados, por lo tanto se ha decidido encapsularlo dentro de una clase independiente, llamada IP. Este clase modela el producto interior, teniendo en cuenta si el problema es generalizado definido positivo. Todos los objetos EPS y SVD construyen automáticamente un objeto IP de forma transparente para el usuario.

El usuario puede seleccionar la variante de Gram-Schmidt con los parámetros de ejecución `-eps_orthog_type` y `-eps_orthog_refinement` mostrados en la tabla 4.1. Para las variantes iterativas se puede controlar el valor de η con el parámetro `-eps_orthog_eta`. Los parámetros para controlar la ortogonalización en un objeto SVD son los mismos, salvo que empiezan por `-svd` en lugar de `-eps`. Estos parámetros se pueden especificar en el programa con la función `IPSetOrthogonalization`. Esta función necesita una referencia al objeto IP que se debe obtener con `EPSGetIP` o `SVDGetIP`.

Las versiones de Gram-Schmidt clásico con refinamiento incluyen la técnica de estimación de la norma, explicada en el capítulo anterior, por tanto los procedimientos de ortogonalización disponibles son:

- Gram-Schmidt modificado.
- Gram-Schmidt modificado con refinamiento.
- Gram-Schmidt modificado con refinamiento iterativo.
- Gram-Schmidt clásico.
- Gram-Schmidt clásico con refinamiento y estimación de la norma.
- Gram-Schmidt clásico con refinamiento iterativo y estimación de la norma.

La implementación del método de Gram-Schmidt clásico se ha realizado con las primitivas de PETSc `VecMAXPY` y `VecMDot`, optimizadas para trabajar con un conjunto de vectores. Estas rutinas consideran al conjunto de vectores como una matriz, lo que permite aprovechar mejor la jerarquía de memorias mediante un producto matriz vector. Como estas rutinas obtienen mejores prestaciones que un bucle de operaciones `VecAXPY` o `VecDot`, el método de Gram-Schmidt modificado no resulta competitivo con respecto del método clásico.

4.2. Método de Arnoldi con reinicio explícito

El método de Arnoldi con reinicio explícito (algoritmo 3.12) está implementado en la clase EPSARNOLDI. El usuario puede seleccionar este método con el parámetro de ejecución `-eps_type arnoldi` o con la función `EPSSetType`.

El bucle interior del método utiliza el producto matriz-vector de PETSc, y la rutina de ortogonalización del objeto IP, para construir una descomposición de Arnoldi. El tamaño de esta descomposición viene determinado por el usuario, mediante la función `EPSSetDimensions` o la opción `-eps_ncv`. Si el usuario no especifica este valor se utiliza la fórmula

$$\text{mín}(n, \text{máx}(2nev, nev+15)) ,$$

donde n es el tamaño del problema y nev es el número de valores propios requeridos con la función `EPSSetDimensions` o la opción `-eps_nev`. Si durante este proceso un vector de Arnoldi resulta combinación lineal de los anteriores, entonces el subespacio de Krylov está agotado, lo que se conoce en la literatura como *breakdown*. En este caso se trabaja solamente con los vectores linealmente independientes obtenidos hasta el momento, y el reinicio se realiza con un vector aleatorio ortonormal a los vectores convergidos. Si no es posible obtener este último vector, el método termina indicando con el flag `EPS_DIVERGED_BREAKDOWN` el motivo de la parada.

La matriz proyectada obtenida tiene forma de Hessenberg superior, y primero se transforma a forma real de Schur con la función DHSEQR de la librería LAPACK. Después se utiliza la función DTREXC para reordenar la forma de Schur de forma numéricamente estable [BD93], y colocar los valores de Ritz deseados al principio de la matriz. Por último, DTREVC calcula los vectores propios de la matriz proyectada, para obtener los vectores de Ritz y sus correspondientes estimaciones de error. Los valores de Ritz situados al principio de la matriz que

Opción	Parámetro de ejecución	Criterio de ordenación
EPS_LARGEST_MAGNITUDE	-eps_largest_magnitude	De mayor a menor $\|\lambda\|$
EPS_SMALLEST_MAGNITUDE	-eps_smallest_magnitude	De menor a mayor $\|\lambda\|$
EPS_LARGEST_REAL	-eps_largest_real	De mayor a menor Re(λ)
EPS_SMALLEST_REAL	-eps_smallest_real	De menor a mayor Re(λ)
EPS_LARGEST_IMAGINARY	-eps_largest_imaginary	De mayor a menor Im(λ)
EPS_SMALLEST_IMAGINARY	-eps_smallest_imaginary	De menor a mayor Im(λ)

Tabla 4.2 Opciones de ordenación en el reinicio para controlar la convergencia con la función EPSSetWhichEigenpairs.

han convergido se bloquean, y se utiliza el primer vector no convergido para reiniciar el método.

Esta ordenación de la forma de Schur se controla con las opciones de la tabla 4.2 y la función EPSSetWhichEigenpairs. Esto permite modificar el comportamiento del método de Arnoldi, que de forma natural converge a los valores propios dominantes. En general, la velocidad de convergencia a los valores propios interiores no resulta competitiva con respecto de la aplicación de transformaciones espectrales.

La principal ventaja de utilizar la forma real de Schur es que no es necesario trabajar con vectores de complejos si el problema es real con soluciones complejas. Como estos valores propios complejos son siempre pares conjugados, se puede almacenar la parte real del vector en el espacio asociado al primer valor conjugado y la parte imaginaria en el espacio asociado al segundo valor. Esto complica en gran medida el código pero permite trabajar solamente con aritmética real, evitando la limitación del lenguaje C estándar original que no puede trabajar con números complejos de forma portable. Si PETSc está compilado con números complejos se emplean las rutinas equivalentes ZHSEQR, ZTREXC y ZTREVC para trabajar con la matriz proyectada. En este caso el código es mucho más simple, porque todos los vectores se almacenan utilizando números complejos.

Esta forma de almacenar los vectores es transparente para el usuario mediante la función EPSGetEigenpair, que calcula los vectores propios a partir de los vectores de Schur solamente si el usuario los solicita explícitamente. Esta función utiliza cuatro parámetros por referencia para devolver un par propio. Si PETSc está configurado para trabajar con números complejos se utilizan solamente dos parámetros, ignorando los otros dos. En el caso contrario, dado que no se

dispone de aritmética compleja, las partes reales e imaginarias del par propio se devuelven por separado.

En la clase EPSARNOLDI también está implementado el procedimiento de ortogonalización clásica de Gram-Schmidt con refinamiento y normalización retrasados. Para ello se sustituye la iteración de Arnoldi básica por su versión modificada con refinamiento y normalización retrasados (algoritmo 3.4). Este procedimiento puede seleccionarse con la opción `-eps_arnoldi_delayed` o la función `EPSArnoldiSetDelayed`.

4.2.1. Experimentos numéricos

Los resultados de esta sección se han obtenido calculando los 10 valores propios dominantes de las 232 matrices de la batería de pruebas con un tamaño de la base fijo de 30 vectores, tolerancia 10^{-7} y 1.000 iteraciones como máximo.

La figura 4.1 permite comparar el nivel de ortogonalidad entre los vectores de Arnoldi utilizando distintas variantes del procedimiento de Gram-Schmidt. En estas gráficas cada punto en el eje horizontal corresponde a una determinada matriz de la batería de pruebas y el eje vertical corresponde al nivel de ortogonalidad. Este nivel se mide como el máximo del valor $\|I - V_m^* V_m\|_F$ en los reinicios del método de Arnoldi.

En estas gráficas se observa que Gram-Schmidt modificado no es capaz de mantener la ortogonalidad, y que para obtener un nivel cercano a la precisión de la máquina es necesario utilizar refinamiento. El procedimiento de Gram-Schmidt clásico iterativo, con estimación de la norma, obtiene un nivel de ortogonalidad similar al modificado con refinamiento, con un coste menor en comunicaciones y operaciones. La iteración de Arnoldi modificada, con normalización y refinamiento retrasados, no consigue un nivel de ortogonalidad tan bueno, pero es mejor que con Gram-Schmidt modificado. Con este último procedimiento se pueden distinguir claramente dos casos, uno en el que el nivel es muy bueno y el algoritmo converge sin problemas, y otro en que el algoritmo no mantiene la ortogonalidad. En este último caso el algoritmo no es capaz de converger de forma satisfactoria.

4.2.2. Análisis de prestaciones

Para comparar las prestaciones del método de Arnoldi con las diferentes versiones de los procedimientos de ortogonalización, se han realizado pruebas variando el número de procesadores. En estas pruebas se ha configurado el

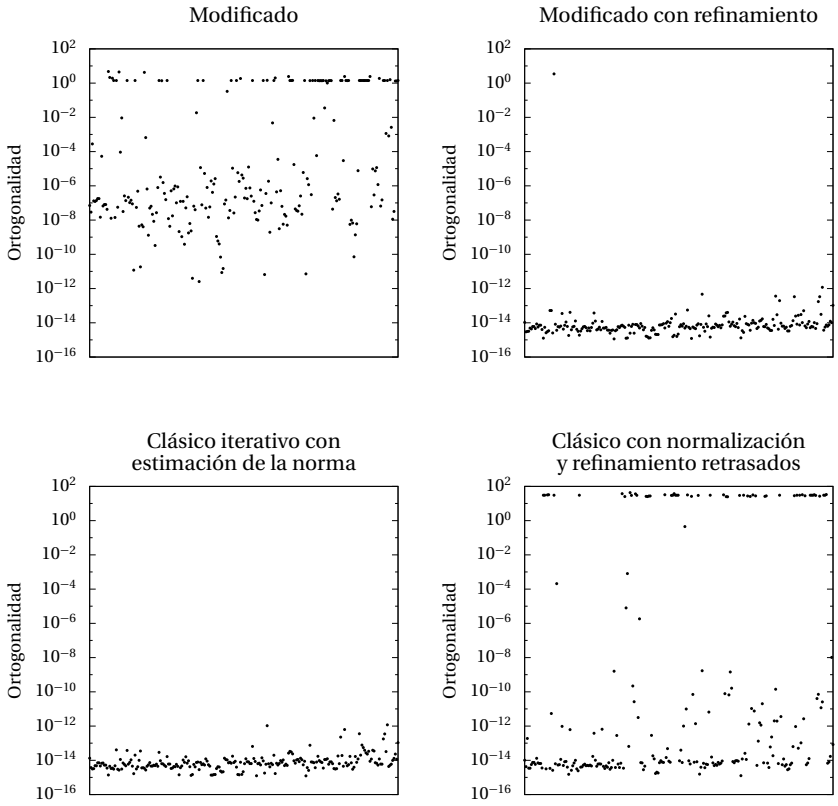

Figura 4.1 Nivel de ortogonalidad máximo entre los vectores de la base del subespacio, calculando 10 valores propios para cada una de las matrices de la batería de pruebas, con el método de Arnoldi con reinicio explícito y distintos procedimientos de Gram-Schmidt.

método para calcular 10 valores propios con una base de 50 vectores, con el resto de parámetros iguales a la batería de pruebas de la sección anterior. Un aspecto muy importante del problema utilizado para comparar los algoritmos, es que el producto matriz vector implementado en PETSc resulte eficiente en paralelo.

La matriz elegida de la batería de pruebas es la AF23560, debido a que su distribución de elementos no nulos es cercana a una matriz banda. Esta matriz es la más grande de la colección NEP con una dimensión de 23.560 y un total de 484.256 elementos no nulos. Proviene de un problema de valores propios para el análisis de estabilidad transitoria de ecuaciones de Navier-Stokes para el flujo en un ala.

En las tablas 4.3 y 4.4 se comparan los tiempos de ejecución secuenciales y número de productos matriz vector para Gram-Schmidt clásico, con un paso de refinamiento y con refinamiento iterativo. No se han realizado pruebas con Gram-Schmidt modificado sin refinamiento, porque este procedimiento no garantiza la ortogonalidad en todos los casos. Tampoco ha parecido necesario hacer pruebas con Gram-Schmidt modificado y refinamiento, puesto que Gram-Schmidt clásico obtiene el mismo nivel de ortogonalidad con un coste en comunicaciones mucho menor. En estas tablas se puede observar que los métodos con estimación de la norma son los más rápidos y, por lo tanto, se utilizarán como tiempo secuencial de referencia para calcular el speed-up. El método de Arnoldi con Gram-Schmidt clásico refinamiento retrasado y normalización retrasada (algoritmo 3.4) resulta mucho más lento en secuencial, debido a que realiza más operaciones.

Las gráficas de las figuras 4.2 y 4.3 muestran que la técnica de estimación de la norma obtiene además mejores prestaciones paralelas que la versión con la norma exacta. También se comprueba que el método de Arnoldi con Gram-Schmidt clásico refinamiento retrasado y normalización retrasada (algoritmo 3.4) es el más rápido con un número alto de procesadores, a pesar de ser el más lento en secuencial. En todos los casos la eficiencia paralela de los métodos baja al aumentar el número de procesadores. Esto es debido a que la cantidad de cálculo en cada procesador disminuye al repartir el trabajo entre los procesadores, y por lo tanto el tiempo empleado en las partes no paralelizadas y en las comunicaciones del algoritmo se hacen dominantes.

Para probar el comportamiento con más procesadores se han repetido los experimentos con la matriz PRE2 perteneciente a la colección de la Universidad de Florida [Dav92]. Esta matriz es mucho más grande que la anterior, con una dimensión de 659.033 y un total de 5.834.044 elementos no nulos agrupados en

		Refinamiento	Refinamiento y estimación de la norma	Normalización y refinamiento retrasados
Odin	**Tiempo**	3,64	3,59	5,45
	Productos	232	232	232
Seaborg	**Tiempo**	7,38	7,31	7,91
	Productos	232	232	232
MareNostrum	**Tiempo**	4,39	4,43	5,24
	Productos	278	278	278

Tabla 4.3 Tiempo de ejecución secuencial (en segundos) y número de productos matriz vector, para calcular 10 valores propios de la matriz AF23560, con el método de Arnoldi y distintas variantes de Gram-Schmidt clásico con un paso de refinamiento.

		Refinamiento iterativo	Refinamiento iterativo y estimación de la norma
Odin	**Tiempo**	3,65	3,61
	Productos	232	232
Seaborg	**Tiempo**	6,17	6,04
	Productos	232	232
MareNostrum	**Tiempo**	4,39	4,34
	Productos	278	278

Tabla 4.4 Tiempo de ejecución secuencial (en segundos) y número de productos matriz vector, para calcular 10 valores propios de la matriz AF23560, con el método de Arnoldi y distintas variantes de Gram-Schmidt clásico con refinamiento iterativo.

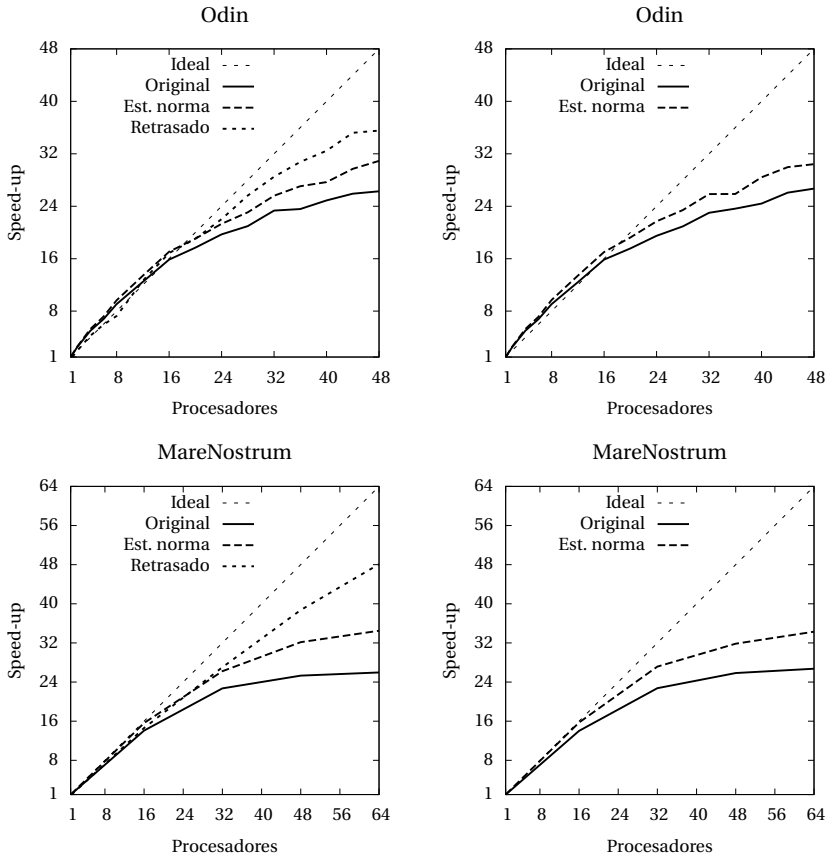

Figura 4.2 Prestaciones paralelas, en Odin y MareNostrum, de las distintas variantes del método de Arnoldi, con Gram-Schmidt clásico con refinamiento (izquierda) y con refinamiento iterativo (derecha), para calcular 10 valores propios de la matriz AF23560.

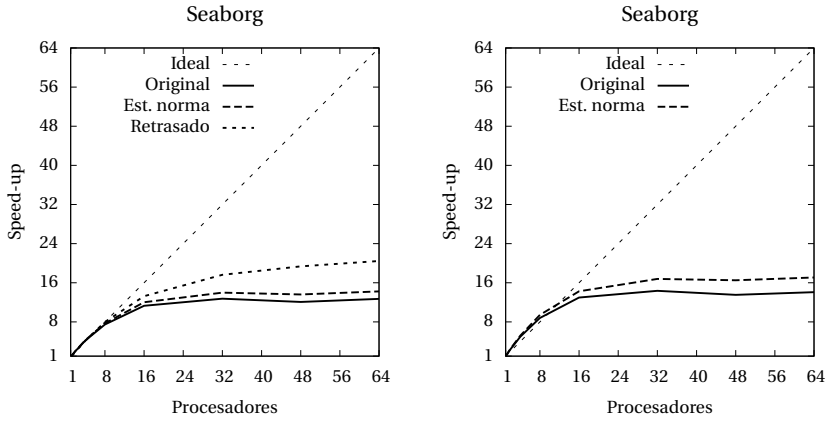

Figura 4.3 Prestaciones paralelas, en Seaborg, de las distintas variantes del método de Arnoldi, con Gram-Schmidt clásico con refinamiento (izquierda) y con refinamiento iterativo (derecha), para calcular 10 valores propios de la matriz AF23560.

una estructura banda. Procede del análisis en frecuencia de grandes circuitos analógicos no-lineales.

Los tiempos de ejecución y número de productos matriz vector para las ejecuciones en secuencial de los distintos algoritmos se muestran en las tablas 4.5 y 4.6. Una diferencia importante de este caso con respecto del anterior, es que el número de iteraciones varía ligeramente con el algoritmo y la máquina utilizada. A pesar de este efecto el procedimiento con refinamiento y estimación de la norma resulta el más rápido de todos. Esta variación también existe entre las ejecuciones del mismo algoritmo con distinto número de procesadores. De hecho, la versión secuencial con normalización y refinamiento retrasado no converge en MareNostrum, mientras que las versiones paralelas funcionan correctamente.

Esta variación en el número de iteraciones se debe a la sensibilidad del algoritmo de Arnoldi al nivel de ortogonalidad, y que los distintos procedimientos de Gram-Schmidt clásico con refinamiento no calculan exactamente el mismo resultado en precisión finita. Es bien conocido [Bjö96] que un paso del procedimiento clásico de Gram-Schmidt produce errores de cancelación, y que el refinamiento utiliza este resultado para corregirlos a un nivel aceptable. Estos

		Refinamiento	Refinamiento y estimación de la norma	Normalización y refinamiento retrasados
Odin	**Tiempo**	910,36	900,86	1058,43
	Productos	1896	1896	2022
Seaborg	**Tiempo**	2261,13	2097,60	2535,50
	Productos	1938	1852	2020
MareNostrum	**Tiempo**	835,58	712,47	–
	Productos	1990	1866	–

Tabla 4.5 Tiempo de ejecución secuencial (en segundos) y número de productos matriz vector, para calcular 10 valores propios de la matriz PRE2, con el método de Arnoldi y distintas variantes de Gram-Schmidt clásico con un paso de refinamiento.

		Refinamiento iterativo	Refinamiento iterativo y estimación de la norma
Odin	**Tiempo**	753,50	718,20
	Productos	1938	1938
Seaborg	**Tiempo**	1797,59	1743,41
	Productos	1896	1940
MareNostrum	**Tiempo**	823,04	753,57
	Productos	1990	1866

Tabla 4.6 Tiempo de ejecución secuencial (en segundos) y número de productos matriz vector, para calcular 10 valores propios de la matriz PRE2, con el método de Arnoldi y distintas variantes de Gram-Schmidt clásico con refinamiento iterativo.

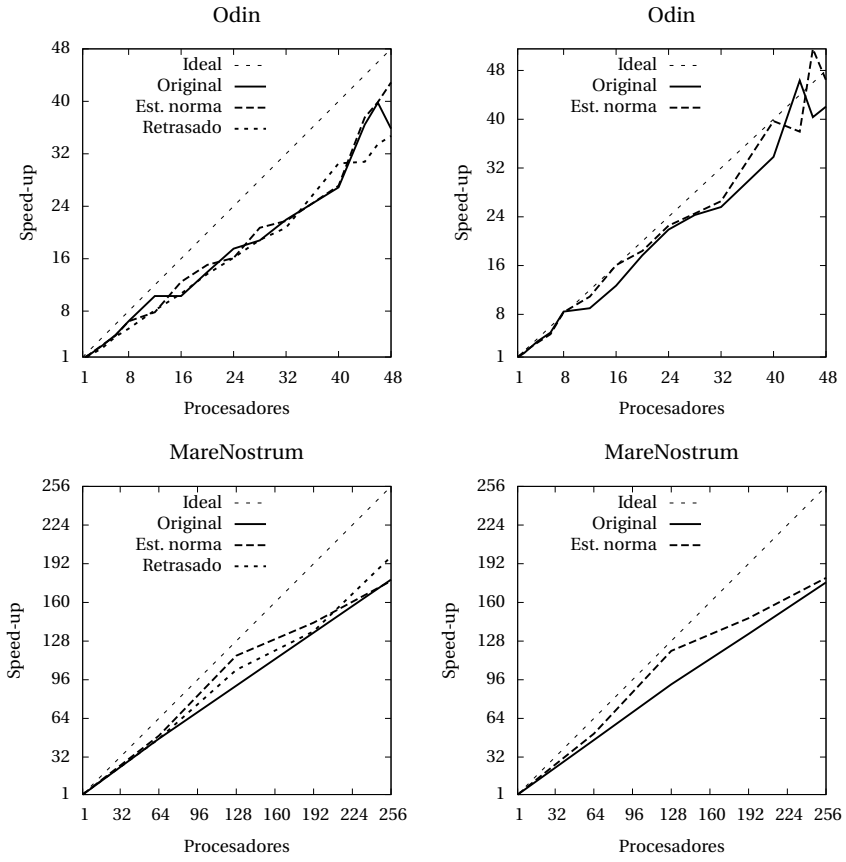

Figura 4.4 Prestaciones paralelas, en Odin y MareNostrum, de las distintas variantes del método de Arnoldi, con Gram-Schmidt clásico con refinamiento (izquierda) y con refinamiento iterativo (derecha), para calcular 10 valores propios de la matriz PRE2.

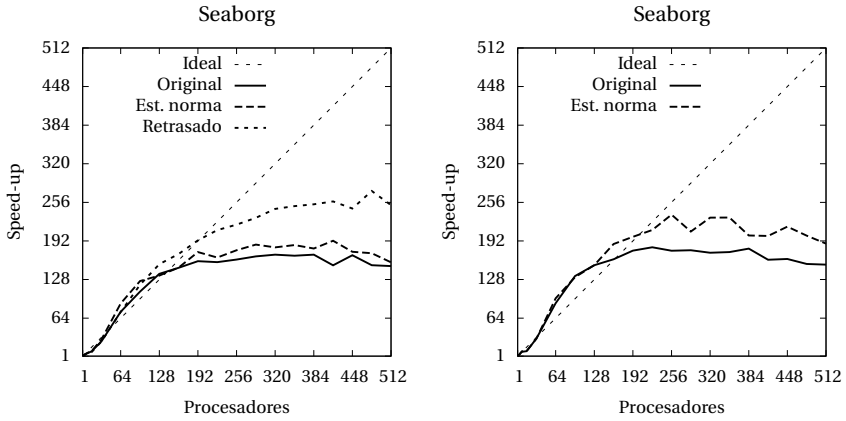

Figura 4.5 Prestaciones paralelas, en Seaborg, de las distintas variantes del método de Arnoldi, con Gram-Schmidt clásico con refinamiento (izq.) y con refinamiento iterativo (der.), para calcular 10 valores propios de la matriz PRE2.

errores dependen de la representación en coma flotante y del orden de las operaciones realizadas. Aunque en las máquinas actuales la representación en coma flotante sigue el estándar IEEE 754, el orden de las operaciones depende del compilador y las opciones de optimización utilizadas. Además, en la implementación de PETSc, el orden de las operaciones en el producto escalar varía con el número de procesadores, puesto que primero se realiza la suma de los productos de la porción de los vectores correspondientes a cada procesador y después se suman estos resultados parciales.

En las figuras 4.4 y 4.5 se muestran las gráficas de speed-up con la matriz PRE2 en las tres máquinas utilizadas. En Odin y MareNostrum se observa que los distintos procedimientos de ortogonalización obtienen prestaciones similares, con variaciones en el speed-up debidas sobre todo a los distintos números de iteraciones. A pesar de estas variaciones, en Seaborg se observa una clara mejora para los procedimientos de estimación de la norma, y sobre todo para la versión modificada con normalización y refinamiento retrasados.

Para medir el speed-up escalado se ha utilizado una matriz tridiagonal de dimensión $10000 \times p$, donde p es el número de procesadores. Los elementos de esta matriz son números aleatorios entre 0 y 1, generados por una función

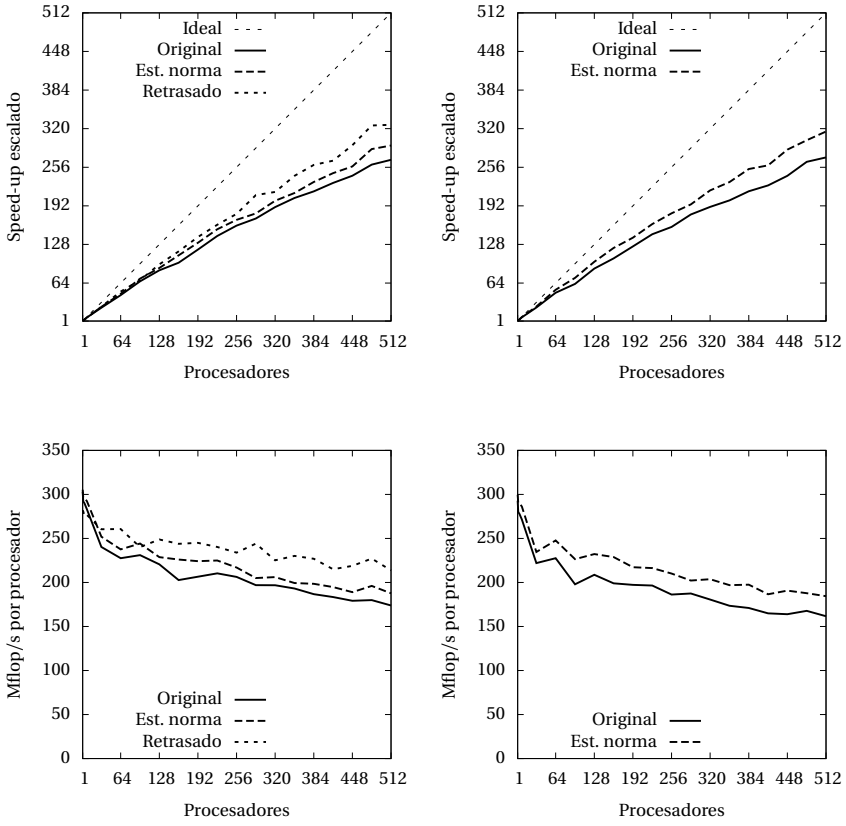

Figura 4.6 Prestaciones paralelas, en Seaborg, de las distintas variantes del método de Arnoldi, con Gram-Schmidt clásico con refinamiento (izquierda) y con refinamiento iterativo (derecha), para calcular 10 valores propios de la matriz tridiagonal sintética.

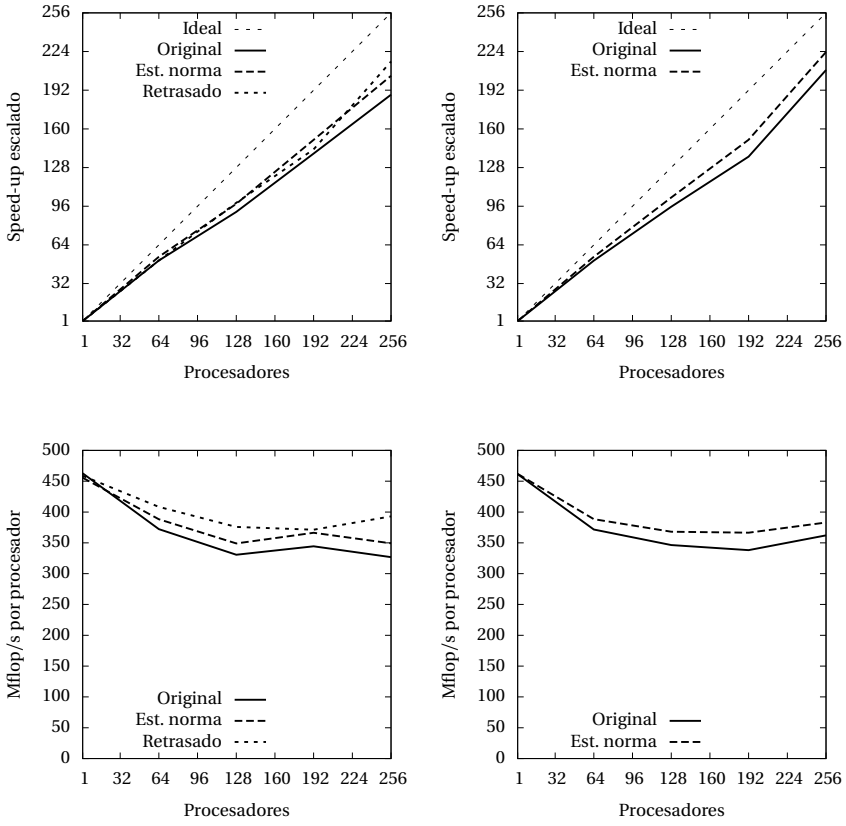

Figura 4.7 Prestaciones paralelas, en MareNostrum, de las distintas variantes del método de Arnoldi, con Gram-Schmidt clásico con refinamiento (izquierda) y con refinamiento iterativo (derecha), para calcular 10 valores propios de la matriz tridiagonal sintética.

Opción	Parámetro	Método
EPSLANCZOS_REORTHOG_LOCAL	local	Local
EPSLANCZOS_REORTHOG_FULL	full	Completa
EPSLANCZOS_REORTHOG_SELECTIVE	selective	Selectiva
EPSLANCZOS_REORTHOG_PERIODIC	periodic	Periódica
EPSLANCZOS_REORTHOG_PARTIAL	partial	Parcial
EPSLANCZOS_REORTHOG_DELAYED	delayed	Completa con retraso de la normalización

Tabla 4.7 Opciones de reortogonalización en el método de Lanczos con la función EPSLanczosSetReorthog o el parámetro de ejecución -eps_lanczos_reorthog.

pseudo-aleatoria. Las gráficas de la figuras 4.6 y 4.7 muestran que con esta matriz todos los algoritmos estudiados presentan una buena escalabilidad y eficiencia en ambas máquinas, con una ligera ventaja para los procedimientos con estimación de la norma y normalización y refinamiento retrasados.

4.3. Método de Lanczos con reinicio explícito

El método de Lanczos con reinicio explícito (algoritmo 3.13) se ha implementado en la clase EPSLANCZOS. El usuario puede seleccionar este método con el parámetro de ejecución -eps_type lanczos o con la función EPSSetType. Al igual que en el método de Arnoldi, se utilizan primitivas de PETSc para paralelizar las operaciones con los vectores de Lanczos y la rutina DSTEVR de LAPACK [DPV04] para calcular los valores y vectores propios de la matriz proyectada. En este caso la implementación resulta mucho más sencilla, puesto que la matriz proyectada y los valores propios son siempre reales. Además, la ordenación de los valores propios para el reinicio se realiza mediante una sencilla permutación. Esta permutación está implementada aprovechando el código realizado para el método de Arnoldi, con la salvedad de que las opciones para ordenar por la parte imaginaria no son aplicables. Al igual que ocurre con Arnoldi, este mecanismo de ordenación no es competitivo para calcular los valores propios interiores.

Esta implementación del método de Lanczos dispone de los esquemas de reortogonalización mostrados en la tabla 4.7. Esta reortogonalización se realiza mediante el objeto IP, que se encarga de utilizar un producto interior para mantener la simetría en los problemas generalizados. Aunque el usuario tiene

plena libertad para elegir entre los procedimientos disponibles, es necesario que garanticen la ortogonalidad a la máxima precisión para que los métodos semiortogonales funcionen correctamente. Los dos esquemas de reortogonalización completa reutilizan la iteración básica y modificada realizadas para el método de Arnoldi.

Los esquemas de reortogonalización periódica y parcial se basan en reortogonalizar el vector de Lanczos calculado cuando el nivel de ortogonalidad con alguno de los vectores anteriores es inferior a $\sqrt{\epsilon}$. En el esquema periódico se reortogonaliza contra todos los vectores anteriores, mientras que en el esquema parcial solamente contra aquellos cuyo nivel de ortogonalidad sea inferior a $\epsilon^{3/4}$. En lugar de calcular el nivel de ortogonalidad, se utiliza una estimación ω_{ik} que aproxima los elementos de la matriz $W_j = V_j^* V_j$ para la iteración j de Lanczos. Esta función se define de forma recurrente como

$$\omega_{k,k} = 1 \,,$$

$$\omega_{k,k-1} = \psi_k \,,$$

$$\beta_{j+1}\omega_{j+1,k} = \beta_{k+1}\omega_{j,k+1} + (\alpha_k - \alpha_j)\omega_{j,k} + \beta_k\omega_{j,k-1} - \beta_j\omega_{j-1,k} + \vartheta_{j,k} \,,$$

para $1 \leq k < j$ con $\omega_{j,k+1} = \omega_{k+1,j}$ y $\omega_{k,0} = 0$. El valor $\psi_k = |v_k^* v_{k-1}|$ representa el nivel de ortogonalidad entre dos vectores de Lanczos consecutivos y $\vartheta_{j,k}$ modela los errores de redondeo cometidos en un paso de la iteración de Lanczos. Estos valores no se calculan explícitamente, sino que se suelen estimar mediante distribuciones aleatorias. La literatura sobre el tema [Grc81, Sim84b, BDD+00] presenta ligeras variaciones en la forma de calcular estas estimaciones, aunque están de acuerdo en que $\psi_k = O(\epsilon)$ y $\vartheta_{j,k} = O(\epsilon\|A\|_2)$. Se ha decidido implementar la recurrencia con las estimaciones utilizadas en PROPACK [Lar98], $\psi_k = \sqrt{n}\frac{\epsilon}{2}$ y $\vartheta_{j,k} = \psi_k\|A\|_2$, donde n es la dimensión de A, puesto que no requieren números aleatorios complicados de obtener de forma fiable en un ordenador. En lugar de calcular la norma $\|A\|_2$, se utiliza como aproximación el valor de Ritz más grande calculado hasta el momento. Como en la primera iteración no se dispone todavía de valores de Ritz, se utiliza el máximo de $|\alpha_i| + \beta_i + \beta_{i+1}$, heurística basada en los círculos de Gershgorin [GVL96, pág. 320].

El esquema de reortogonalización selectiva necesita estimar la norma del residuo de cada uno de los valores de Ritz en cada iteración de Lanczos. Para evitar resolver un problema de valores propios tridiagonal completo en cada iteración de Lanczos, se han propuesto métodos que calculan de forma económica la norma del residuo a partir de los valores de la iteración anterior [PR81, PNO85].

Estas técnicas no se han considerado en SLEPc, debido a que el tamaño de la tridiagonal es relativamente pequeño y acotado gracias a la utilización del reinicio. Por lo tanto, la estimación del residuo se ha implementado resolviendo el problema de valores propios tridiagonal con la librería LAPACK de forma secuencial. La aplicación de reinicio permite dos alternativas diferentes para implementar la reortogonalización selectiva, según las acciones realizadas en el momento de detectar una pérdida de ortogonalidad. La primera, y más sencilla, es parar la iteración de Lanczos y forzar el reinicio. La segunda, es mantener explícitamente la ortogonalidad entre los vectores de Lanczos en las iteraciones de Lanczos restantes. Esta última alternativa proporciona una convergencia más rápida y por lo tanto, es la implementada en SLEPc.

En el esquema de reortogonalización local la corrección propuesta en [CW85] no funciona, debido a que el tamaño de la matriz proyectada es demasiado pequeño. En su lugar, esta implementación calcula explícitamente la norma del residuo de los pares propios supuestamente convergidos, para comprobar que la estimación del método de Lanczos es correcta y así eliminar la aparición de valores propios espúreos. Los valores propios convergidos dentro de un reinicio se consideran repetidos si caen dentro del intervalo definido por la precisión requerida por el usuario. De estos valores repetidos se descartan todos menos uno, que se deja bloqueado como convergido. Como el algoritmo no garantiza la ortogonalidad de este vector con respecto de los anteriores, se le aplica el procedimiento de Gram-Schmidt para asegurar la ortogonalidad entre los vectores propios convergidos. Si el mismo valor propio vuelve a converger en iteraciones sucesivas, se considera como valor propio múltiple.

En los problemas generalizados, cuando la matriz que define el producto interior es semidefinida positiva, es necesario primero corregir el vector inicial, y después corregir los vectores propios convergidos para evitar las componentes no deseadas que surjan por errores de redondeo, técnica conocida como *purificación* (sección 3.2.2).

Aunque se han propuesto técnicas avanzadas para ahorrar operaciones en el proceso de corrección, esta implementación utiliza la forma más sencilla, que consiste es aplicar el operador definido por la transformación espectral y normalizar el resultado. Este proceso es transparente al usuario y se realiza en la función `EPSGetEigenpair` solamente si se solicitan los vectores propios.

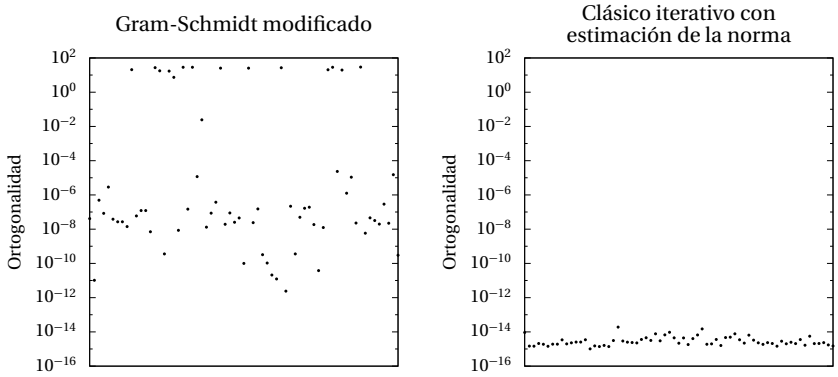

Figura 4.8 Nivel de ortogonalidad máximo entre los vectores de Lanczos, calculando 10 valores propios para cada una de las matrices de la batería de pruebas, con el método de Lanczos con reinicio explícito y reortogonalización completa con Gram-Schmidt modificado o clásico iterativo con estimación de la norma.

4.3.1. Experimentos numéricos

La implementación del método de Lanczos se ha validado con las 67 matrices simétricas reales de la batería de pruebas. El número de vectores y la condición de parada se ha configurado de la misma forma que con el método de Arnoldi.

La figura 4.8 muestra el nivel de ortogonalidad medido de la misma forma que en la sección anterior, como el máximo del valor $\|I - V_m^* V_m\|_F$ en los reinicios del método de Lanczos. En esta figura se compara la reortogonalización completa con el procedimiento de Gram-Schmidt modificado y con el procedimiento clásico iterativo con estimación de la norma, y se observa claramente que el primero no es capaz de garantizar la ortogonalidad en todos los casos, al igual que ocurre en el método de Arnoldi, mientras que el segundo consigue un nivel cercano a la precisión máxima. En el resto de esta sección se trabajará en exclusiva con el procedimiento de Gram-Schmidt clásico iterativo con estimación de la norma, puesto que en la sección anterior se ha comprobado que es el más eficiente de los propuestos que garantizan la ortogonalidad en todos los casos.

Los niveles de ortogonalidad utilizando este procedimiento, con distintos esquemas de reortogonalización, se muestran en la figura 4.9, donde se aprecia

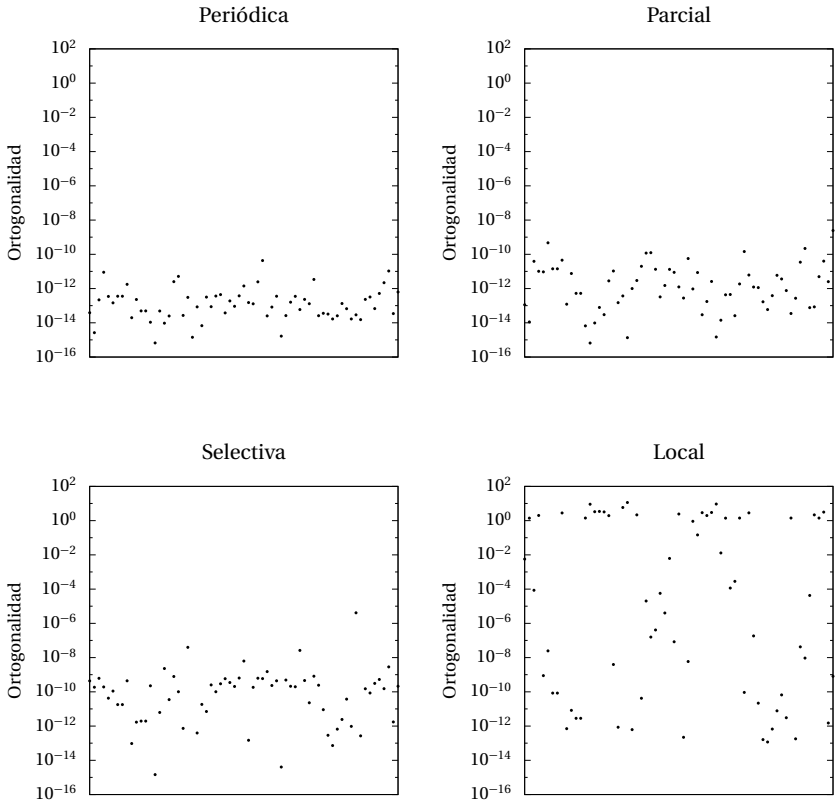

Figura 4.9 Nivel de ortogonalidad máximo entre los vectores de Lanczos, calculando 10 valores propios para cada una de las matrices de la batería de pruebas, con el método de Lanczos con reinicio explícito y distintas variantes de reortogonalización.

que se mantiene la semiortogonalidad en todos los esquemas excepto en el de reortogonalización local. A pesar de esta pérdida de ortogonalidad, la convergencia del método de Lanczos con el esquema local no resulta afectada. Sin embargo, para que cada vector propio convergido sea ortogonal con el resto, es necesario reortogonalizar de forma explícita en el momento que se detecta la convergencia.

El procedimiento de Gram-Schmidt clásico, con normalización y refinamiento retrasado, no se va a tener en cuenta en los resultados de esta sección debido a que solamente parece compatible con el esquema de reortogonalización completa, y por lo tanto sus prestaciones serán inferiores a los esquemas semiortogonales.

4.3.2. Análisis de prestaciones

La comparación de la eficiencia paralela del método de Lanczos, con distintos esquemas de reortogonalización, se ha realizado con las matrices CRYSTK03 y AF_SHELL1 pertenecientes a la colección de la Universidad de Florida [Dav92]. La matriz CRYSTK03 tiene una dimensión de 24.696 y un total de 1.751.178 elementos no nulos. Forma parte de un problema de valores propios generalizado procedente del análisis en elementos finitos de las vibraciones de un cristal. La matriz AF_SHELL1 tiene una dimensión de 504.855 y un total de 17.562.051 elementos no nulos. Procede de un problema de formación de hojas de metal en la industria del automóvil. Los elementos de ambas matrices son reales con una estructura en banda, por lo que el producto matriz vector resulta muy eficiente en PETSc.

En las tablas 4.8 y 4.9 se muestran los tiempos de ejecución secuenciales y el número de productos matriz vector del método de Lanczos con reinicio explícito y los distintos esquemas de reortogonalización. El método se ha configurado para calcular los 10 valores propios más grandes con una base de 50 vectores, utilizando el procedimiento clásico de Gram-Schmidt iterativo con estimación de la norma. Un aspecto interesante es que el número de productos es mayor con el esquema de reortogonalización local, debido a que se calcula la norma del residuo explícitamente, lo que necesita productos matriz vector extra. Este esquema resulta el más rápido, porque el coste de realizar estos productos matriz vector queda compensado por el ahorro de la reortogonalización.

Las figuras 4.10 y 4.11 muestran el speed-up de los distintos esquemas con las dos matrices CRYSTK03 y AF_SHELL1. Con la matriz más pequeña todos los algoritmos pierden eficiencia con un número no muy grande de procesadores. En

		Completa	Local	Periódica	Parcial	Selectiva
Odin	**Tiempo**	5,10	3,25	4,68	4,62	3,68
	Productos	184	194	184	184	184
MareNostrum	**Tiempo**	4,79	3,02	4,37	4,32	3,15
	Productos	184	194	184	184	184

Tabla 4.8 Tiempo de ejecución secuencial (en segundos) y número de productos matriz vector, para calcular 10 valores propios de la matriz CRYSTK03, con el método de Lanczos y distintas variantes de reortogonalización.

		Completa	Local	Periódica	Parcial	Selectiva
Odin	**Tiempo**	132,66	77,33	122,14	121,72	83,19
	Productos	265	275	265	265	265
MareNostrum	**Tiempo**	120,35	68,11	108,73	108,73	72,28
	Productos	265	275	265	265	265

Tabla 4.9 Tiempo de ejecución secuencial (en segundos) y número de productos matriz vector, para calcular 10 valores propios de la matriz AF_SHELL1, con el método de Lanczos y distintas variantes de reortogonalización.

ambas matrices el esquema de reortogonalización local es el más rápido, tanto en secuencial como en paralelo. Las prestaciones de la reortogonalización completa, periódica y parcial son muy similares, lo que hace suponer que el ahorro de operaciones en estos esquemas semiortogonales no es muy significativo. El esquema de reortogonalización selectiva obtiene peores prestaciones con la matriz AF23560 en Odin, debido a que el tiempo necesario para la estimación de la norma del residuo (proceso secuencial) en cada iteración no se compensa con el ahorro en reortogonalizaciones (proceso paralelo). Las prestaciones del esquema selectivo mejoran, bien con una implementación de LAPACK más eficiente como la de MareNostrum, bien con un problema más grande, quedando incluso por encima de los otros métodos semiortogonales.

De la misma forma que con el método de Arnoldi, el speed-up obtenido con el método de Lanczos está limitado por el tamaño de la matriz. Para medir el speed-up con más procesadores se ha recurrido a la matriz tridiagonal de dimensión 10000 por procesador, pero en esta ocasión los valores aleatorios se han elegido de forma que la matriz sea simétrica. Las gráficas de la figura 4.12 muestran que las variantes con reortogonalización local y selectiva son claramente superiores a los otros esquemas. Paradójicamente, los dos esquemas

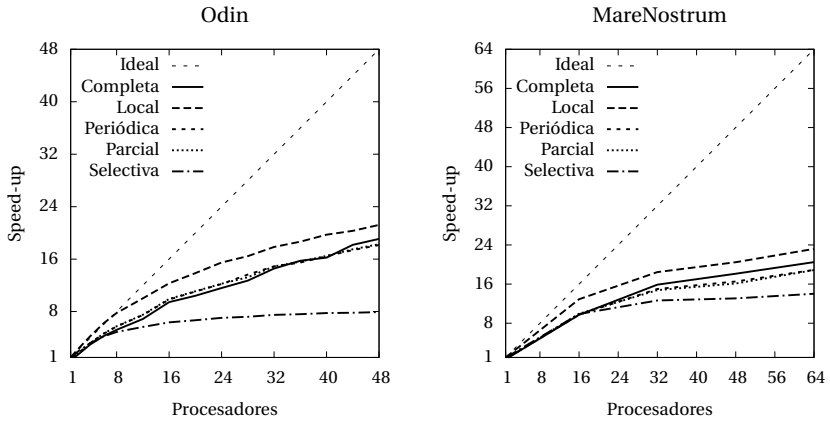

Figura 4.10 Prestaciones paralelas, en Odin y MareNostrum, de las distintas variantes del método de Lanczos para calcular 10 valores propios de la matriz CRYSTK03.

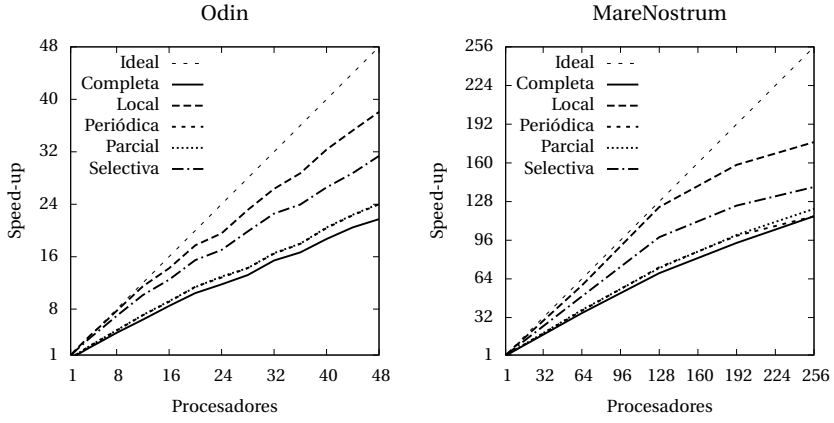

Figura 4.11 Prestaciones paralelas, en Odin y MareNostrum, de las distintas variantes del método de Lanczos para calcular 10 valores propios de la matriz AF_SHELL1.

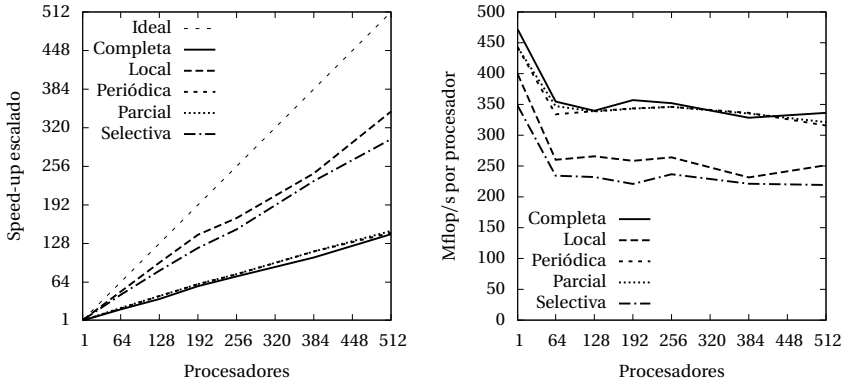

Figura 4.12 Prestaciones paralelas, en MareNostrum, de las distintas variantes del método de Lanczos para calcular 10 valores propios de la matriz tridiagonal sintética.

más eficientes obtienen peores valores de Mflop/s, esto se debe a que realizan menos reortogonalizaciones que los otros esquemas. La implementación en SLEPc del procedimiento clásico de Gram-Schmidt resulta más eficiente que otras operaciones, gracias al empleo de rutinas tipo producto matriz densa por vector, que obtienen una mayor productividad en operaciones de coma flotante, aprovechando mejor la arquitectura del computador. Por lo tanto, realizar un mayor número de reortogonalizaciones produce un incremento de la tasa de Mflop/s, aunque el proceso resulte más lento en su totalidad. A pesar de esta ventaja, los esquemas de reortogonalización completa, periódica y parcial no resultan competitivos debido al elevado número de operaciones en coma flotante realizadas.

4.4. Método de Krylov-Schur

El método de Krylov-Schur se ha implementado en la clase EPSKRYLOVSCHUR. Este método puede ser seleccionado con el parámetro de ejecución `-eps_type krylovschur` o con la función `EPSSetType`. La implementación de este método tiene muchos puntos en común con la del método de Arnoldi,

en particular, la iteración básica, el procedimiento de ortogonalización y parte del tratamiento de la matriz proyectada.

En este caso, como la matriz proyectada no tiene forma de Hessenberg superior, primero se reduce con la funciones de LAPACK DGEHRD y DORGHR, y después se trabaja con las funciones implementadas en el método de Arnoldi. Un aspecto importante es la selección del número de vectores que se mantienen en la base durante el reinicio. A falta de una teoría general para ello se ha optado por una heurística bastante simple: elegir la mitad de los vectores de la base cuyos valores de Ritz estén más cerca de la parte del espectro deseada. Para seleccionar estos vectores y controlar la convergencia del método se aprovecha el código desarrollado para ello en el método de Arnoldi. Esta heurística puede verse como un equilibrio entre dos factores opuestos, por un lado cuantos más vectores se conserven en el reinicio menos información se pierde, y por el otro es necesario calcular nuevos vectores de Arnoldi para que el método avance.

La versión Hermitiana de Krylov-Schur, Lanczos con reinicio grueso, se ha implementado en la misma clase dado que no existen muchas diferencias en su implementación, principalmente la utilización de la rutina DSYEVR de LAPACK para encontrar los valores y vectores propios de la matriz proyectada. No se ha implementado ningún método semiortogonal debido a que la complejidad extra del código no compensaría el ahorro de tiempo de ejecución. En particular, no existe un equivalente claro para el reinicio grueso de la recurrencia que estima el nivel de ortogonalidad entre los vectores de Lanczos. Además, los experimentos numéricos parecen sugerir que es necesario reortogonalizar con los vectores de reinicio en todas las iteraciones, negando en gran parte el beneficio de los métodos semiortogonales.

La obtención de los vectores propios mediante la rutina `EPSGetEigenpair` aprovecha la implementación realizada para los métodos de Arnoldi y Lanczos. En los casos no Hermitianos se calculan los vectores propios a partir de los vectores de Schur tal y como se describe en la sección 4.2. Y en los casos Hermitianos generalizados se utiliza la purificación descrita anteriormente para el método de Lanczos en la sección 4.3.

4.4.1. Experimentos numéricos

Para comprobar la relación entre la implementación de Krylov-Schur y el método de Arnoldi con reinicio implícito de ARPACK se ha recurrido a la batería de pruebas utilizada previamente. En la figura 4.13 se muestra una comparativa del número de productos matriz vector realizados por Krylov-Schur, ARPACK y

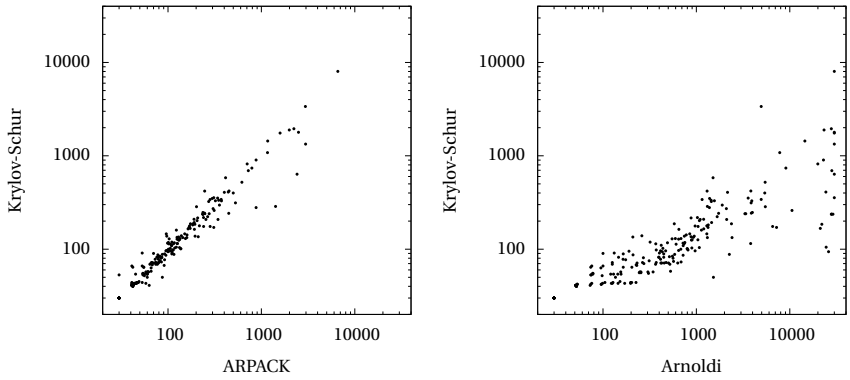

Figura 4.13 Comparativa entre Krylov-Schur, ARPACK y Arnoldi en cuanto al número de productos matriz vector para las matrices de la batería de pruebas.

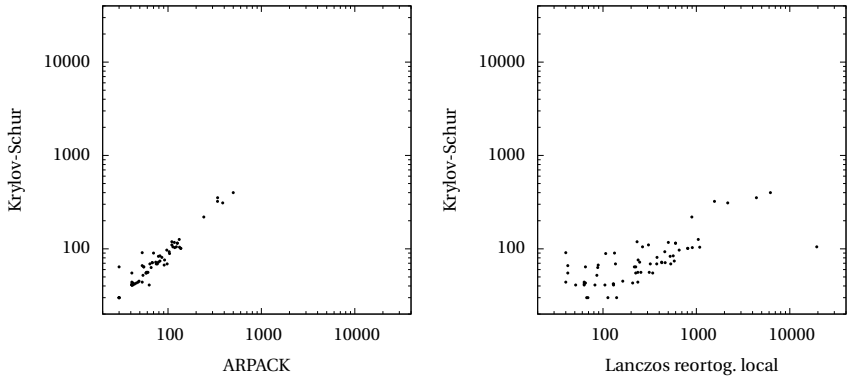

Figura 4.14 Comparativa entre Krylov-Schur, ARPACK y Lanczos con reortogonalización local en cuanto al número de productos matriz vector para las matrices simétricas de la batería de pruebas.

Arnoldi para las distintas matrices de la batería de prueba. En estas gráficas el eje vertical mide el número de productos realizados por Krylov-Schur, el eje horizontal los realizados por ARPACK o Arnoldi y cada punto representa una matriz de la batería de prueba. Se ha utilizado una escala logarítmica para distinguir mejor los puntos individuales. En la comparativa de Krylov-Schur con ARPACK se observar que en general los puntos se agrupan en una recta con pendiente de 45 grados, es decir, que los dos métodos necesitan un número similar de productos matriz vector para converger. Esto es compatible con la descripción teórica que propone que ambos métodos construyen descomposiciones equivalentes. También queda claro en estas gráficas que el método de Arnoldi con reinicio explícito necesita muchas más iteraciones para resolver el mismo problema que Krylov-Schur.

Las gráficas de la figura 4.14 comparan las versiones Hermitianas de Krylov-Schur, ARPACK y Lanczos con reortogonalización local para las matrices simétricas de la batería de pruebas. En la sección anterior se ha mostrado que esta variante es la más eficiente de Lanczos, sin embargo realiza muchos más productos matriz vector y no es competitiva con Krylov-Schur. Al igual que para el caso no Hermitiano, ARPACK y Krylov-Schur parecen equivalentes en número de productos matriz vector.

Debido a que la versión de Krylov-Schur a bloques de Anasazi no es capaz de completar satisfactoriamente la batería de pruebas, no se ha podido incluir en esta comparativa.

4.4.2. Análisis de prestaciones

Las medidas de prestaciones se han realizado con las matrices AF23560 y PRE2 utilizadas anteriormente para el método de Arnoldi. En primer lugar es necesario establecer el tamaño óptimo de bloque en estos problemas para Anasazi. Para ello se han realizado pruebas en secuencial calculando 10 valores propios variando únicamente el tamaño de bloque. Dado que este tamaño tiene que ser divisor exacto del tamaño de la base, se ha elegido una de 60 vectores que permite utilizar bloques de 1 a 6 vectores. Los resultados de estas pruebas en Odin se muestran en las tablas 4.10 y 4.11. En ambos casos se observa que el coste en iteraciones y tiempo aumenta significativamente con el tamaño de bloque y que para un bloque de tamaño 5 el método no converge en un tiempo razonable. El gran inconveniente de los métodos de Krylov a bloques es que necesitan más productos matriz vector para obtener un subespacio con el mismo grado en A que los métodos de vector único [Saa92, cap. 6]. Los métodos a bloques

Tamaño de bloque	Reinicios	Productos matriz vector	Tiempo de ejecución
1	2	112	2,47
2	3	160	4,02
3	3	156	3,70
4	4	204	5,26
5	×	×	×
6	6	300	8,57

Tabla 4.10 Tiempos de ejecución secuenciales, número de reinicios y productos matriz vector realizados por Anasazi, para obtener 10 valores propios de la matriz AF23560, variando el tamaño de bloque.

Tamaño de bloque	Reinicios	Productos matriz vector	Tiempo de ejecución
1	14	724	423,76
2	34	1710	1005,41
3	63	3036	1873,91
4	125	6012	3484,18
5	×	×	×
6	385	18492	11201,33

Tabla 4.11 Tiempos de ejecución secuenciales, número de reinicios y productos matriz vector realizados por Anasazi, para obtener 10 valores propios de la matriz PRE2, variando el tamaño de bloque.

puede compensar este inconveniente con una implementación con rutinas tipo producto de matrices que aprovechan mejor la jerarquía de memoria, pero obviamente la implementación de Anasazi no lo consigue en estos casos. Como el tamaño de bloque 1 es el más eficiente en estos dos casos se usará en los análisis siguientes. Esto convierte efectivamente la implementación a bloques de Anasazi en un método de vector único como SLEPc o ARPACK.

Los tiempos de ejecución secuencial y número de reinicios y productos matriz vector para ARPACK y las implementaciones de Krylov-Schur de SLEPc y Anasazi se detallan en las tablas 4.12 y 4.13 para las matrices AF23560 y PRE2. Los métodos se han configurado para obtener 10 valores propios con una tolerancia de 10^{-7} empleando una base de 30 vectores y el resto de parámetros por defecto.

		ARPACK	SLEPc	Anasazi
Odin	**Tiempo**	1,16	1,50	2,64
	Reinicios	5	6	8
	Productos	100	94	178
MareNostrum	**Tiempo**	1,07	1,08	-
	Reinicios	5	6	-
	Productos	100	94	-

Tabla 4.12 Tiempo de ejecución secuencial (en segundos), número de reinicios y productos matriz vector, para calcular 10 valores propios de la matriz AF23560, con ARPACK y con el método de Krylov-Schur implementado en SLEPc y Anasazi.

		ARPACK	SLEPc	Anasazi
Odin	**Tiempo**	306,53	214,87	514,32
	Reinicios	27	35	60
	Productos	526	494	1270
MareNostrum	**Tiempo**	175,21	183,85	-
	Reinicios	27	35	-
	Productos	526	494	-

Tabla 4.13 Tiempo de ejecución secuencial (en segundos), número de reinicios y productos matriz vector, para calcular 10 valores propios de la matriz PRE2, con ARPACK y con el método de Krylov-Schur implementado en SLEPc y Anazasi.

Las prestaciones de ARPACK y SLEPc son similares en ambas máquinas, con la excepción de la matriz PRE2 en Odin donde ARPACK resulta un poco más lento que SLEPc. Con ambas matrices las prestaciones de Anasazi en Odin son claramente inferiores, tanto en tiempo como número de iteraciones. Dadas las pobres prestaciones de Anasazi tanto en secuencial como en paralelo (como se verá a continuación) y las limitaciones de disponibilidad de MareNostrum no se han realizado pruebas en esta máquina.

Las prestaciones paralelas de las tres implementaciones se comparan en las gráficas 4.15 y 4.16, donde se observa claramente que la implementación de Anasazi no resulta en absoluto competitiva. Con estas dos matrices la implementación de SLEPc es más eficiente que ARPACK, especialmente en MareNostrum. Para extender el análisis a más procesadores se ha recurrido a la tridiagonal de tamaño variable utilizada anteriormente para el método de Arnoldi. En este

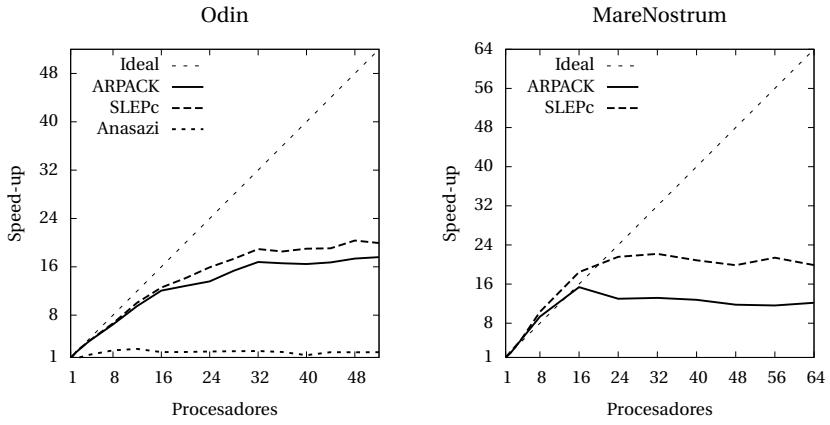

Figura 4.15 Prestaciones paralelas, en Odin y MareNostrum, de ARPACK y los métodos de Krylov-Schur de SLEPc y Anasazi, para calcular 10 valores propios de la matriz AF23560.

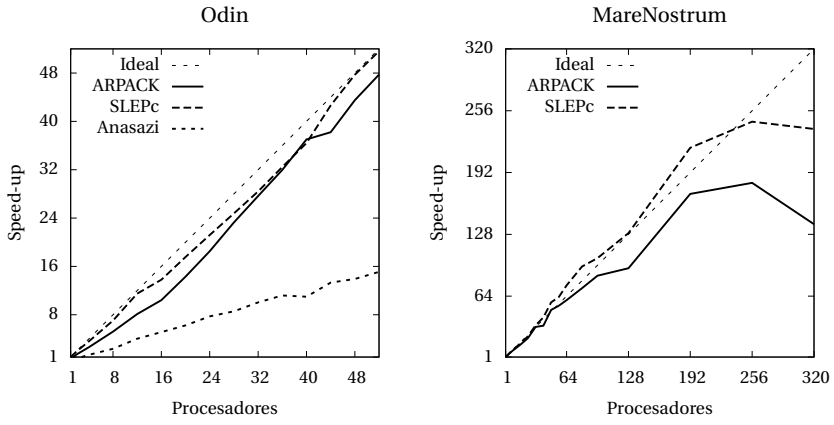

Figura 4.16 Prestaciones paralelas, en Odin y MareNostrum, de ARPACK y los métodos de Krylov-Schur de SLEPc y Anasazi, para calcular 10 valores propios de la matriz PRE2.

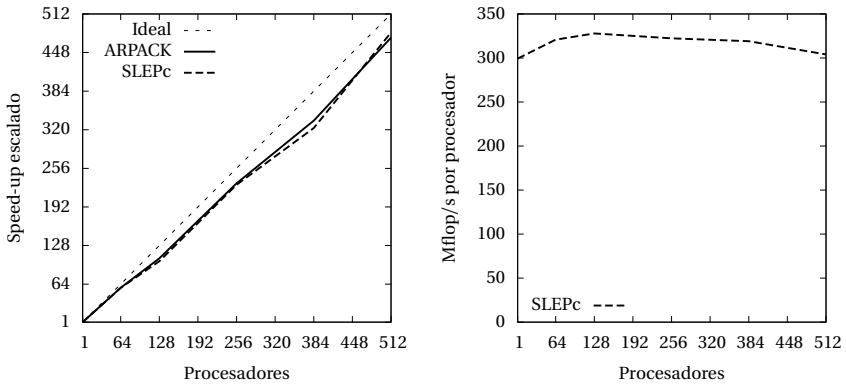

Figura 4.17 Prestaciones paralelas, en MareNostrum, de ARPACK y el método de Krylov-Schur de SLEPc, para calcular 10 valores propios de la matriz tridiagonal sintética.

caso se ha utilizado un tamaño más grande, 100.000 elementos por procesador, para que el tiempo de ejecución resulte significativo. La figura 4.17 muestra que tanto ARPACK como SLEPc obtienen un excelente speed-up escalado. ARPACK no dispone de mecanismos para medir los Mflop por segundo, por lo que no se incluye en la gráfica.

4.5. Método de Golub-Kahan-Lanczos con reinicio explícito

El método de Golub-Kahan-Lanczos con reinicio explícito (algoritmo 3.14) se ha implementado en la clase SVDLANCZOS. El usuario puede seleccionar este método con el parámetro de ejecución `-svd_type lanczos` o con la función `SVDSetType`. Al igual que con los métodos para valores propios, esta implementación utiliza primitivas de PETSc para paralelizar los vectores de Lanczos y la matriz proyectada se replica en todos los procesadores. Para calcular la SVD de la matriz proyectada se utiliza la rutina DBDSDC de LAPACK, que implementa el método de divide y vencerás para matrices bidiagonales.

La implementación en SLEPc del método de Golub-Kahan-Lanczos solamente dispone de reortogonalización completa para evitar los problemas de pérdida de ortogonalidad inherentes a los métodos de Lanczos. A la vista de los resultados mostrados en la sección 4.3.2, se ha decidido que el esfuerzo de implementar esquemas semiortogonales no se vería compensado por las prestaciones que se obtendrían. Además, tampoco parece interesante abordar otras técnicas, porque gracias a los resultados de la sección anterior, sabemos que el reinicio explícito no resulta competitivo con respecto del reinicio grueso.

Los procedimientos de Gram-Schmidt utilizados para mantener la ortogonalidad son los mismos utilizados para valores propios, disponibles en la clase IP. Los parámetros de ejecución para controlar la ortogonalización son los mismos que para el problema de valores propios, con la excepción de empezar por `-svd_orthog` en lugar de `-eps_orthog`.

La versión con reortogonalización por un solo lado está disponible con la opción `-svd_lanczos_oneside` o la función `SVDLanczosSetOneSide`. La implementación se ha realizado sustituyendo la iteración básica de Gloub-Kahan-Lanczos por el algoritmo 3.11. Esta versión no solamente reduce a la mitad el coste de la reortogonalización sino que permite ahorrar una gran cantidad de memoria al utilizar solamente dos vectores P. Además se ahorra una operación de multireducción retrasando la normalización del vector p_j hasta la ortogonalización del vector q_{j+1}. En este caso los vectores singulares por la izquierda se calculan a partir del valor y vector singular por la derecha como en el caso del producto cruzado (sección 1.2.1). Aunque los vectores por la derecha sean ortogonales, es posible que los vectores por la izquierda obtenidos de esta forma no lo sean debido a errores de redondeo si la matriz está muy mal condicionada. Para evitar esta pérdida de ortogonalidad, se aplica entre los vectores por la izquierda el mismo procedimiento de Gram-Schmidt empleado en el algoritmo a medida que se obtienen a partir de los vectores por la derecha. Este cálculo extra solamente se realiza si el usuario solicita los vectores por la izquierda a través de la función `SVDGetSingularTriplet`.

4.5.1. Experimentos numéricos

La validación del algoritmo se ha realizado con las 164 matrices no simétricas de la batería utilizada en las secciones anteriores. El método se ha configurado para que calcule los 10 valores singulares más grandes con una tolerancia de 10^{-7} y 1000 iteraciones como máximo. Al igual que en las pruebas anteriores se ha utilizado una base de 30 vectores.

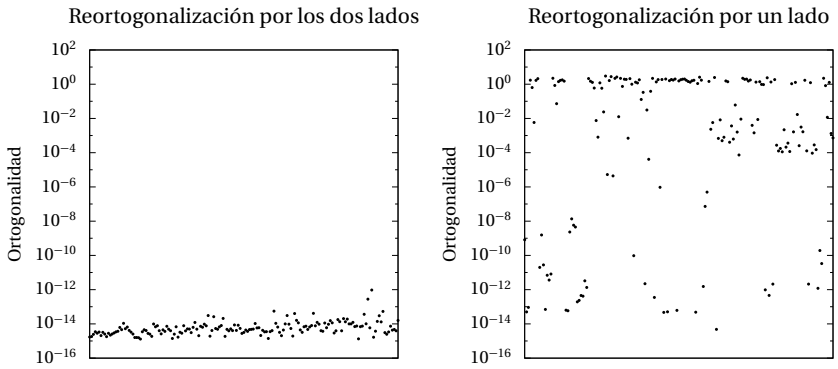

Figura 4.18 Nivel de ortogonalidad máximo entre los vectores P, calculando 10 valores singulares para cada una de las matrices de la batería de pruebas, con el método de Golub-Kahan-Lanczos con reinicio explícito y reortogonalización completa por los dos lados (izquierda) y por un lado (derecha).

Las gráficas de la figura 4.18 muestran el nivel de ortogonalidad entre los vectores p_j para las dos variantes del método implementado. En ambos casos se ha utilizado el procedimiento clásico de Gram-Schmidt con refinamiento y estimación de la norma que se ha demostrado como el más eficiente de los planteados en esta tesis. De la misma forma que en gráficas anteriores, cada punto en el eje horizontal corresponde a una determinada matriz de la batería de pruebas y el eje vertical corresponde al nivel de ortogonalidad. Este nivel se mide como el máximo del valor $\|I - P_m^* P_m\|_F$ en cada uno de los reinicios. No se han incluido gráficas del nivel de ortogonalidad entre los vectores q_j, dado que el procedimiento de ortogonalización utilizado garantiza la ortogonalidad hasta la precisión máxima posible. La gráfica de la izquierda corresponde al método con ortogonalización por los dos lados, donde se observa como la ortogonalidad es buena, tal y como es previsible al ortogonalizar los vectores p_j de forma explícita en cada iteración. La gráfica de la derecha muestra que la ortogonalidad entre los vectores p_j se pierde en el método con ortogonalización por un solo lado, aunque este efecto no impide la convergencia hacia los valores singulares requeridos.

		Producto cruzado	Golub-Kahan-Lanczos	GKL por un lado
Odin	**Tiempo**	4,38	5,69	4,42
	Reinicios	11	11	11
	Productos matriz vector	562	562	562
	Productos escalares	10452	15351	10173
MareNostrum	**Tiempo**	3,35	4,17	3,39
	Reinicios	11	11	11
	Productos matriz vector	562	562	562
	Productos escalares	10452	15351	10173

Tabla 4.14 Tiempo de ejecución secuencial (en segundos) y número de operaciones, para calcular 10 valores singulares de la matriz AF23560, con el método de Golub-Kahan-Lanczos con reinicio explícito.

4.5.2. Análisis de prestaciones

Las medidas de prestaciones se han realizado en Odin y MareNostrum con las matrices AF23560 y PRE2 utilizadas anteriormente para el método de Arnoldi. Los métodos se han configurado para obtener 10 valores singulares con una tolerancia de 10^{-7} empleando una base de 30 vectores y el resto de parámetros por defecto. Las tablas 4.14 y 4.15 permiten comparar los tiempos de ejecución en secuencial y número de operaciones entre los métodos de Golub-Kahan-Lanczos con ortogonalización por uno y dos lados. También se han realizado pruebas calculando los valores propios del producto cruzado con el método de Lanczos con reortogonalización completa. No se han incluido pruebas con los valores propios de la matriz cíclica dado que su coste es siempre superior a estos tres métodos. En las tablas se puede observar que el método de Golub-Kahan-Lanczos es bastante más lento que el producto cruzado, dado que realiza más productos escalares. Sin embargo, las prestaciones mejoran mucho al utilizar la ortogonalización por un solo lado, haciendo al método casi tan eficiente como el producto cruzado pero con la ventaja de ser más estable.

Las figuras 4.19 y 4.20 muestran el speed-up de los tres métodos tomando como referencia el más rápido en secuencial, que es el producto cruzado en

		Producto cruzado	Golub-Kahan-Lanczos	GKL por un lado
Odin	**Tiempo**	612,56	938,09	700,34
	Reinicios	69	80	78
	Productos matriz vector	3542	4076	3972
	Productos escalares	65397	109298	71892
MareNostrum	**Tiempo**	755,25	990,81	733,32
	Reinicios	93	90	90
	Productos matriz vector	4618	4566	4494
	Productos escalares	87201	124301	82399

Tabla 4.15 Tiempo de ejecución secuencial (en segundos) y número de operaciones, para calcular 10 valores singulares de la matriz PRE2, con el método de Golub-Kahan-Lanczos con reinicio explícito.

todos los casos excepto uno. Los métodos se han configurado de la misma forma que para las ejecuciones secuenciales. Al igual que en casos anteriores, el tiempo de proceso con la matriz AF23560 es demasiado pequeño, haciendo que las prestaciones empeoren con un número considerable de procesadores. Con la matriz PRE2 más grande se obtienen buenos resultados con aproximadamente 200 procesadores. En ambos casos se observa que las mejores prestaciones del producto cruzado se mantienen al aumentar el número de procesadores. También se observa que el método con ortogonalización por un solo lado es más eficiente que el método con ortogonalización por los dos lados, llegando en ocasiones a ser tan rápido como el producto cruzado.

Para extender los resultados a más procesadores se ha recurrido a una tridiagonal sintética no simétrica similar a la utilizada con el método de Arnoldi. Esta tridiagonal tiene una dimensión de 10000 columnas y filas por cada procesador con sus elementos aleatorios. En la figura 4.21 se observa que los tres métodos probados tienen una escalabilidad muy buena con hasta 500 procesadores. Debido a que la ortogonalización está implementada con funciones de tipo producto matriz vector que aprovechan mejor la jerarquía de memoria, el método de Golub-Kahan-Lanczos con reortogonalización por los lados obtiene

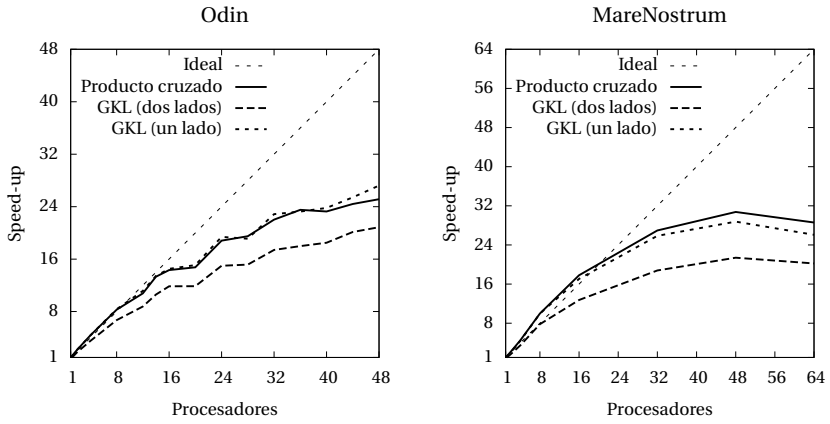

Figura 4.19 Prestaciones paralelas, en Odin y MareNostrum, del método de Golub-Kahan-Lanczos con reinicio explícito, para calcular 10 valores singulares de la matriz AF23560.

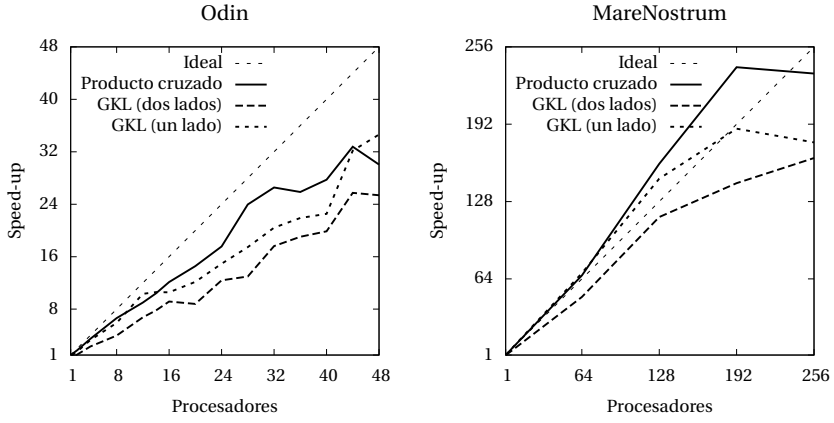

Figura 4.20 Prestaciones paralelas, en Odin y MareNostrum, del método de Golub-Kahan-Lanczos con reinicio explícito, para calcular 10 valores singulares de la matriz PRE2.

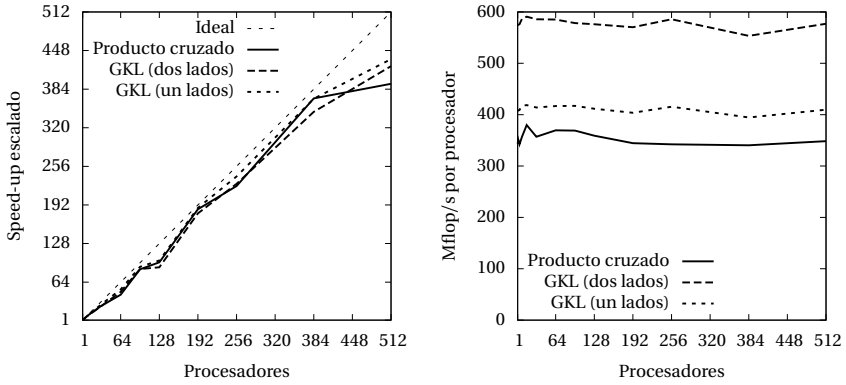

Figura 4.21 Prestaciones paralelas, en MareNostrum ,del método de Golub-Kahan-Lanczos con reinicio explícito, para calcular 10 valores singulares de la matriz tridiagonal sintética.

mejores resultados con la métrica de Mflops/s. Este efecto es similar al ocurrido en la sección 4.3.2 para las distintas variantes de reortogonalización del método de Lanczos para valores propios.

4.6. Método de Golub-Kahan-Lanczos con reinicio grueso

El método de Golub-Kahan-Lanczos con reinicio grueso se ha implementado en la clase SVDTRLANCZOS. El usuario puede seleccionar este método con el parámetro de ejecución `-svd_type trlanczos` o con la función `SVDSetType`. Esta implementación está basada en la del mismo método con reinicio explícito, compartiendo parte del código para construir la bidiagonal. Al igual que en el reinicio explícito, la paralelización de las operaciones con los vectores de Lanczos se realiza mediante primitivas de PETSc y la matriz proyectada está replicada en todos los procesadores. En este caso la matriz proyectada no es bidiagonal y se utiliza la rutina DGESDD para matrices generales de LAPACK.

El procedimiento de Gram-Schmidt utilizado en el método dispone de las mismas posibilidades que la versión con reinicio explícito, controladas por la función `IPSetOrthogonalization` y los parámetros de ejecución que empiezan por `-svd_orthog`. La versión con reortogonalización por un solo lado está disponible con la opción `-svd_trlanczos_oneside` o la función `SVDTRLanczosSetOneSide`. De esta forma se ahorra una considerable cantidad de cálculo, pero a diferencia de la versión con reinicio explícito, en este caso no se ahorra memoria al ser necesario almacenar los dos conjuntos de vectores. Además, al finalizar las iteraciones se aplica el procedimiento de Gram-Schmidt a los vectores por la izquierda obtenidos dado que el algoritmo no garantiza su ortogonalidad en casos donde la matriz está muy mal condicionada.

De la misma forma que con el método de Lanczos con reinicio grueso para valores propios, se ha utilizado la heurística de mantener la mitad de los vectores de Lanczos para realizar el reinicio. No se han considerado variantes semi-ortogonales para este método, debido a que la complejidad extra no parece compensar la mejora de prestaciones, y además, no resulta sencillo deducir una función de recurrencia que estime la ortogonalidad de forma robusta en todos los casos.

4.6.1. Experimentos numéricos

La validación de este método se ha realizado con la misma batería de pruebas utilizada en la sección anterior para el método con reinicio explícito. El método se ha configurado de la misma forma para que calcule los 10 valores singulares más grandes con una base de 30 vectores.

Las gráficas de la figura 4.22 muestran el nivel de ortogonalidad entre los vectores p_j calculado de la misma forma que en la sección anterior para las variantes del método con reinicio grueso y reortogonalización por uno y dos lados. El procedimiento de ortogonalización utilizado es el clásico de Gram-Schmidt con refinamiento y estimación de la norma al igual que en la sección anterior. En la gráfica de la izquierda se observa que la ortogonalidad se mantiene gracias a que el algoritmo aplica el procedimiento de forma explícita a los vectores p_j. En la variante con ortogonalización por un lado (gráfica de la derecha) se produce una pérdida de ortogonalidad menor que con el método con reinicio explícito (ver la gráfica de la derecha en la figura 4.18). Este efecto se debe al reinicio grueso, que en la práctica solamente construye la mitad de vectores de Lanczos que el reinicio explícito si ambos métodos utilizan el mismo tamaño de base.

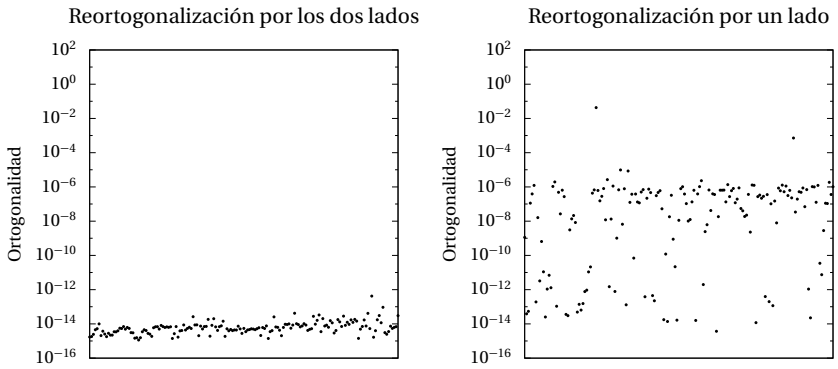

Figura 4.22 Nivel de ortogonalidad máximo entre los vectores P, calculando 10 valores singulares para cada una de las matrices de la batería de pruebas, con el método de Golub-Kahan-Lanczos con reinicio grueso y reortogonalización completa por los dos lados (izquierda) y por un lado (derecha).

4.6.2. Análisis de prestaciones

Las medidas de prestaciones se han realizado en Odin y MareNostrum con las matrices AF23560 y PRE2 utilizadas anteriormente para el método de Golub-Kahan-Lanczos con reinicio explícito. Al igual que en la sección anterior, los métodos se han configurado para obtener 10 valores singulares con una tolerancia de 10^{-7} empleando una base de 30 vectores y el resto de parámetros por defecto. Las tablas 4.16 y 4.17 permiten comparar los tiempos de ejecución en secuencial y número de operaciones entre los métodos de Golub-Kahan-Lanczos con reinicio grueso y ortogonalización por uno y dos lados. También se han realizado pruebas calculando los valores propios del producto cruzado con el método de Lanczos con reortogonalización completa y reinicio grueso (implementado en SLEPc dentro del método de Krylov-Schur). No se han incluido pruebas con los valores propios de la matriz cíclica puesto que su coste es siempre superior a estos tres métodos. En las tablas se puede observar que el método de Golub-Kahan-Lanczos es bastante más lento que el producto cruzado, dado que realiza más productos escalares. Sin embargo, las prestaciones mejoran mucho al utili-

		Producto cruzado	Golub-Kahan-Lanczos	GKL por un lado
Odin	**Tiempo**	1,09	1,59	1,17
	Reinicios	3	3	3
	Productos matriz vector	116	116	116
	Productos escalares	2358	3544	1911
MareNostrum	**Tiempo**	0,82	1,13	0,87
	Reinicios	3	3	3
	Productos matriz vector	116	116	116
	Productos escalares	2358	3544	1911

Tabla 4.16 Tiempo de ejecución secuencial (en segundos) y número de operaciones, para calcular 10 valores singulares de la matriz AF23560, con el método de Lanczos con reinicio grueso.

		Producto cruzado	Golub-Kahan-Lanczos	GKL por un lado
Odin	**Tiempo**	74,73	111,33	81,06
	Reinicios	9	9	9
	Productos matriz vector	300	300	300
	Productos escalares	6748	9910	5108
MareNostrum	**Tiempo**	68,31	104,21	74,84
	Reinicios	9	9	9
	Productos matriz vector	300	300	300
	Productos escalares	6748	9910	5108

Tabla 4.17 Tiempo de ejecución secuencial (en segundos) y número de operaciones, para calcular 10 valores singulares de la matriz PRE2, con el método de Lanczos con reinicio grueso.

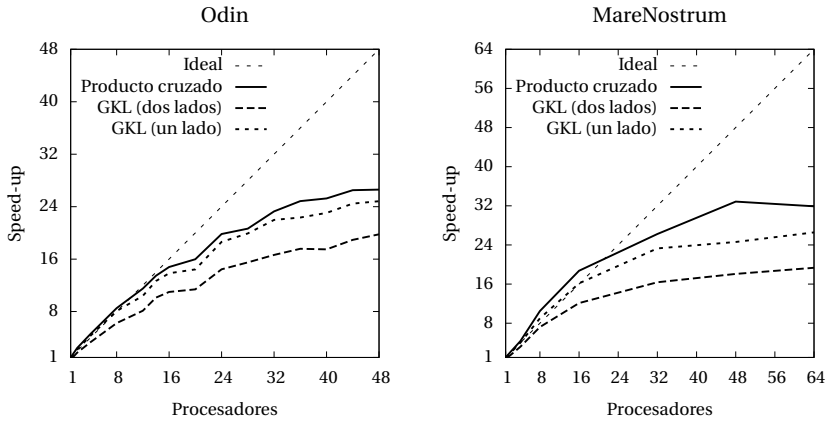

Figura 4.23 Prestaciones paralelas, en Odin y MareNostrum, del método de Golub-Kahan-Lanczos con reinicio grueso, para calcular 10 valores singulares de la matriz AF23560.

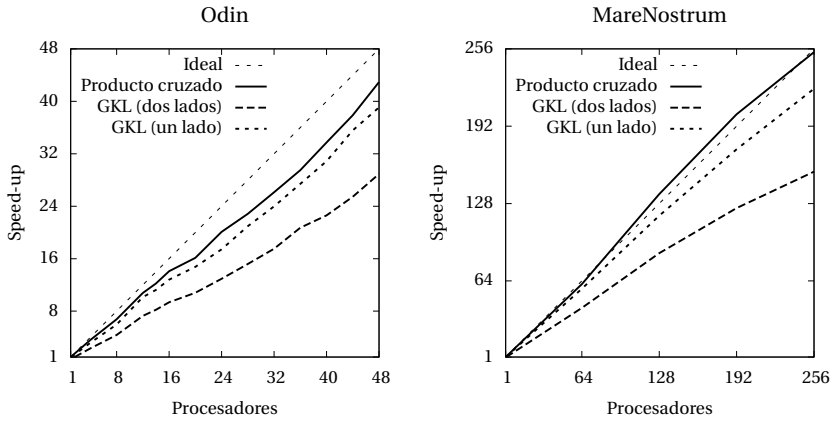

Figura 4.24 Prestaciones paralelas, en Odin y MareNostrum, del método de Golub-Kahan-Lanczos con reinicio grueso, para calcular 10 valores singulares de la matriz PRE2.

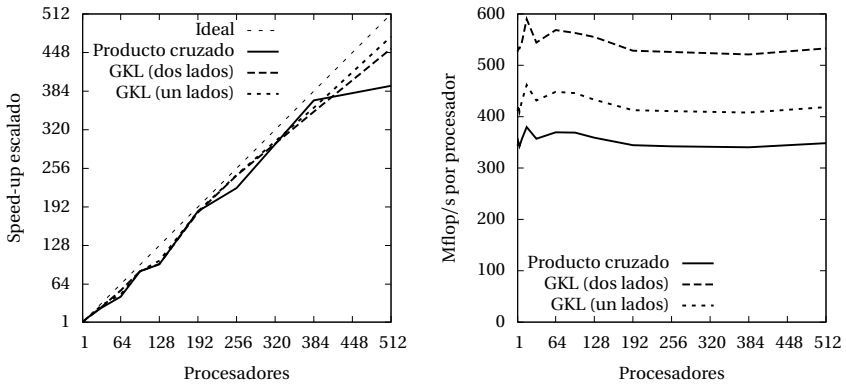

Figura 4.25 Prestaciones paralelas, en MareNostrum, del método de Golub-Kahan-Lanczos con reinicio grueso, para calcular 10 valores singulares de la matriz tridiagonal sintética.

zar la ortogonalización por un solo lado, haciendo al método competitivo pero sin los problemas de estabilidad asociados al producto cruzado.

Las figuras 4.23 y 4.24 muestran el speed-up de los tres métodos tomando como referencia el producto cruzado que es el más rápido en secuencial en todos los casos. Los métodos se han configurado de la misma forma que para las ejecuciones secuenciales. Al igual que en casos anteriores, el tiempo de proceso con la matriz AF23560 es demasiado pequeño, haciendo que las prestaciones empeoren con un número considerable de procesadores. Con la matriz PRE2 más grande se obtienen buenos resultados con más de 200 procesadores. En ambos casos se observa que las mejores prestaciones del producto cruzado se mantienen al aumentar el número de procesadores. También se observa que el método con ortogonalización por un solo lado es más eficiente que el método con ortogonalización por los dos lados, resultando competitivo con el producto cruzado.

Para extender los resultados a más procesadores se ha recurrido a la tridiagonal sintética no simétrica utilizada en la sección anterior con el reinicio explícito. Esta tridiagonal tiene una dimensión de 10.000 columnas y filas por cada procesador con sus elementos aleatorios. En la figura 4.25 se observa que los tres

métodos probados tienen una escalabilidad muy buena con hasta 500 procesadores. Al igual que ocurre con el reinicio explícito, los métodos menos eficientes obtienen mejores resultados con la métrica de Mflops/s debido a que realizan más ortogonalizaciones, y por lo tanto más operaciones de tipo producto matriz vector que aprovechan mejor la jerarquía de memoria.

4.7. Paralelización híbrida con OpenMP y MPI

Las máquinas paralelas actuales son en su mayoría de tipo cluster, donde cada nodo dispone de un pequeño número de procesadores con memoria compartida. Sin embargo, PETSc y SLEPc utilizan un modelo puro de memoria distribuida que no aprovecha de forma óptima estas arquitecturas. En estas máquinas se puede utilizar un modelo de ejecución híbrido, utilizando memoria compartida entre los procesadores de un mismo nodo y simultáneamente un modelo de memoria distribuida entre los procesadores de distintos nodos.

Para implementar la parte de memoria compartida de este modelo híbrido en SLEPc se ha elegido OpenMP, que es un estándar consolidado y que garantiza portabilidad a gran variedad de plataformas. Además, resulta más sencillo de programar que el estándar POSIX para hilos múltiples.

Aunque el estándar MPI-2 incluye soporte para hilos de ejecución múltiples, este soporte no es obligatorio y no está disponible todavía en la mayoría de las implementaciones. Esto implica que las primitivas MPI no pueden ser invocadas dentro de una región paralela de OpenMP. Para evitar este problema, el enfoque ha sido restringir el uso de OpenMP a las secciones de código puramente computacionales.

Dado que OpenMP permite paralelizar de forma gradual, dicha paralelización se centrará inicialmente en las operaciones de mayor coste. En la tabla 4.18 se desglosa por operaciones de PETSc el tiempo de ejecución secuencial del método de Krylov-Schur en Odin y en MareNostrum para calcular los 10 valores propios más grandes de la matriz PRE2. El resto de parámetros del método se ha configurado de la misma forma que las pruebas de la sección 4.4.2, incluyendo el procedimiento de ortogonalización. En esta tabla se observa que en la ejecución secuencial más del 50 % del tiempo de cálculo corresponde a las operaciones de PETSc `VecMAXPY` y más del 20 % a `VecMDot`. Esta distribución del tiempo de ejecución es típica para los métodos de Krylov, donde la mayor parte del cálculo se concentra en la ortogonalización de los vectores si la matriz del problema es lo suficientemente dispersa.

	Porcentaje del tiempo secuencial	
Operación	**Odin**	**MareNostrum**
`VecMAXPY`	56,8	57,5
`VecMDot`	21,6	26,1
`MatMult`	9,7	9,5
`VecAXPY`	2,7	2,1
`VecScale`	1,5	0,8
`VecCopy`	1,9	1,2
`VecSet`	3,8	0,9
Otras	2,0	1,9

Tabla 4.18 Desglose por operaciones del tiempo de ejecución secuencial.

La ley de Amdahl nos permite calcular el speed-up máximo S_p como

$$S_p = \frac{1}{(1-F)+\frac{F}{p}}$$

donde F es la fracción de tiempo secuencial y p el número de procesadores. Paralelizando estas dos operaciones se podría llegar con los dos procesadores de Odin a un speed-up de 1,6 dentro del nodo. Por otro lado, en MareNostrum se obtendría un speed-up de 2,7 con los cuatro procesadores de cada nodo, que aunque está lejos de ser óptimo es comparable al speed-up obtenido utilizando solamente MPI.

4.7.1. Detalles de implementación

El modelo de ejecución de OpenMP tiene un hilo principal por nodo que cuando llega a las regiones paralelas activa tantos hilos de ejecución como haya indicado el usuario (en este caso tantos como procesadores tiene un nodo). Al finalizar la región paralela solamente queda en ejecución el hilo principal, que es el único encargado de realizar las llamadas a MPI necesarias. De esta forma el paralelismo realizado con OpenMP es transparente al resto del código de la librería que utiliza MPI.

AXPY múltiple

La operación `VecMAXPY` de PETSc se utiliza para implementar la segunda parte del procedimiento de Gram-Schmidt (líneas $v_{j+1} = v_{j+1} - V_j h_{1:j,j}$ y $v_{j+1} = v_{j+1} - V_j c$ del algoritmo 3.3). Esta operación es una versión optimizada para m vectores de la típica operación AXPY y calcula la suma:

$$y = y + \sum_{i=1}^{m} \alpha_i x_i$$

donde $y, x_i \in \mathbb{R}^n$ y $\alpha \in \mathbb{R}^m$. Una forma sencilla de paralelizar esta operación con OpenMP sin problemas de dependencia de datos es la siguiente:

para $i = 1, 2, \ldots, n$ (en paralelo)
  para $j = 1, 2, \ldots, m$
    $y_i = y_i + \alpha_j \cdot x_{i,j}$
  fin
fin

Al paralelizar el bucle externo, el efecto es que cada hilo opera sobre un subconjunto de filas de los vectores implicados. Esta distribución de la carga está bien balanceada porque n es mucho más grande que el número de procesadores. Sin embargo sus prestaciones son bastante pobres en la práctica porque los elementos de los vectores x_i no se acceden de forma contigua lo que produce un mal aprovechamiento de la memoria caché. Para corregir este efecto se ha añadido un bucle interno que recorre los vectores en bloques de tamaño k:

para $i = 0, k, 2k, \ldots, n-1$ (en paralelo)
  para $j = 1, \ldots, m$
    para $l = i+1, i+2, \ldots, i+k$
      $y_l = y_l + \alpha_j \cdot x_{l,j}$
    fin
  fin
fin

Con un tamaño de bloque adecuado todos los elementos de vector y permanecen en la memoria caché durante el bucle interno. De esta forma no existe penalización por las múltiples escrituras en y y los vectores x_i se acceden con un patrón secuencial dentro del bucle interno. Es necesario fijar el tamaño del bloque de antemano para que el compilador sea capaz de generar código óptimo. Se ha elegido un tamaño de 1.024 después de realizar varios experimentos en las máquinas probadas.

Producto escalar múltiple

La primera parte del procedimiento de Gram-Schmidt (líneas $h_{1:j,j} = V_j^* u_{j+1}$ y $c = V_j^* u_{j+1}$ del algoritmo 3.3) está implementada con la operación `VecMDot` de PETSc. Esta operación es también una versión optimizada para m vectores del producto escalar de vectores que calcula el vector $z \in \mathbb{R}^m$:

$$z_i = y^T x_i \quad i = 1, 2, \ldots, m$$

donde $y, x_i \in \mathbb{R}^n$. Al igual que en el caso anterior esta operación resulta trivial de paralelizar con OpenMP. Pero en este caso es necesario una multireducción de MPI para combinar los resultados parciales entre los nodos:

para $i = 1, \ldots, m$ (en paralelo)
  $z_i = 0$
  para $j = 1, \ldots, n$
    $z_i = z_i + y_j \cdot x_{i,j}$
  fin
fin
sumar z entre todos los nodos

Sin embargo, esta versión no tiene la carga bien balanceada debido a que m puede ser menor que el número de procesadores. Para equilibrar la carga hay que repartir los vectores de la misma forma que con la operación anterior:

$z = 0$
para $j = 0, k, 2k, \ldots, n-1$ (en paralelo)
  para $i = 1, \ldots, m$
    para $l = j+1, j+2, \ldots, j+k$
      $z_i = z_i + y_l \cdot x_{i,l}$
    fin
  fin
fin
sumar z entre los procesadores de un nodo
sumar z entre todos los nodos

Para mejorar el patrón de acceso a memoria se ha utilizado la misma técnica de acceso a bloques. Desgraciadamente con este enfoque aparece una condición de carrera en la actualización de z_i. Para evitarla se pueden utilizar las primitivas de sincronización de OpenMP para crear una sección crítica, pero a costa de

un impacto muy grave en las prestaciones. Como el tamaño del vector z es pequeño, se ha decidido trabajar con una copia local a cada hilo de ejecución y después combinar los resultados parciales dentro del nodo antes de hacer la multireducción con MPI como se muestra en el algoritmo.

4.7.2. Análisis de prestaciones

Para estudiar las prestaciones de esta implementación híbrida se ha utilizado el método de Krylov-Schur para calcular los 10 valores propios más grandes de la matriz PRE2. El resto de parámetros del método se ha configurado de la misma forma que las pruebas de la sección 4.4.2, incluyendo el procedimiento de ortogonalización. Primero se ha probado con un proceso MPI por nodo tal y como se ha realizado en las secciones anteriores. Después se ha lanzado con tantos procesos MPI por nodo como procesadores tiene cada nodo, dos en Odin y cuatro en MareNostrum. Por último, se ha probado la implementación presentada en esta sección con tantos hilos de ejecución como procesadores tiene cada nodo.

La gráfica de la izquierda en la figura 4.26 muestra el speed-up obtenido en Odin con dos hilos de ejecución por nodo, donde se observa una clara mejora con respecto de la versión con dos procesos MPI por nodo. En la gráfica de la derecha se observa una mayor eficiencia paralela con la versión OpenMP aunque no se llega a aprovechar totalmente las posibilidades de la máquina.

En MareNostrum se han realizado las pruebas con cuatro hilos de ejecución por nodo, cuyos resultados se reflejan en las gráficas de la figura 4.27. A pesar de que la paralelización con OpenMP solamente se aplica a una parte del código, el speed-up y la eficiencia paralela no resultan muy inferiores a la versión con cuatro procesos MPI por nodo.

Este modelo híbrido funciona mejor que el de memoria distribuida pura en máquinas donde la implementación de las comunicaciones no es eficiente cuando hay varios procesos MPI dentro de un mismo nodo, como es el caso de Odin. En máquinas con una implementación más eficiente, como MareNostrum, se consiguen prestaciones comparables a pesar de haber paralelizado con OpenMP solamente dos operaciones.

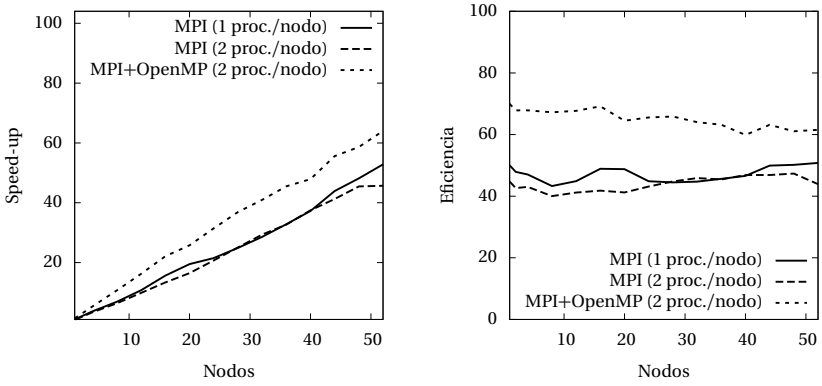

Figura 4.26 Prestaciones paralelas, en Odin, del método de Krylov-Schur para calcular 10 valores propios de la matriz PRE2, con diferentes configuraciones de procesos e hilos.

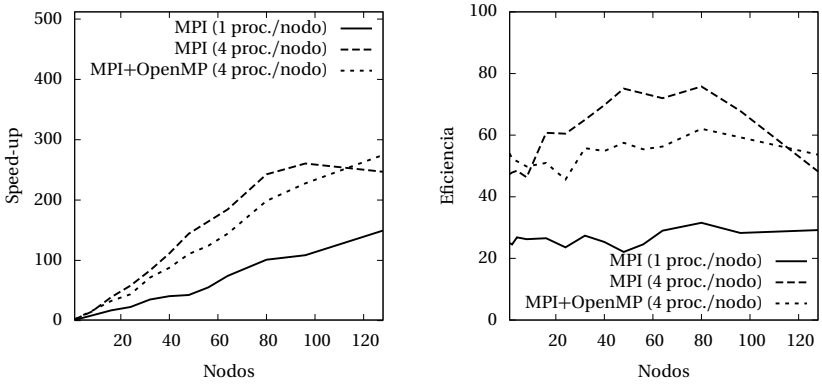

Figura 4.27 Prestaciones paralelas, en MareNostrum, del método de Krylov-Schur para calcular 10 valores propios de la matriz PRE2, con diferentes configuraciones de procesos e hilos.

Capítulo 5

Conclusiones

En esta tesis se ha revisado el estado del arte de los métodos para problemas dispersos de valores propios y singulares. Se han estudiado los métodos de Arnoldi y Lanczos , con especial atención a los procedimientos de ortogonalización. En este contexto se han aplicado técnicas de combinación de comunicaciones para mejorar las prestaciones paralelas del procedimiento de Gram-Schmidt clásico. Estas mejoras no se habían propuesto anteriormente para métodos de valores propios y singulares. Se ha aportado un nuevo esquema de ortogonalización con refinamiento que reduce al mínimo las comunicaciones necesarias. También se han revisado las técnicas de reinicio para mejorar y modificar la convergencia de los métodos de Krylov. Por último, se ha presentado una implementación paralela con paso de mensajes de los métodos de Arnoldi y Lanczos con reinicio explícito, y el método de Krylov-Schur para problemas dispersos de valores propios. Esta implementación de Lanczos incorpora distintas estrategias de reortogonalización, permitiendo su comparación de forma sencilla, característica que no estaba disponible en ninguna otra implementación. Además, se ha implementado los métodos de Golub-Kahan-Lanczos con reinicio explícito y grueso para problemas dispersos singulares, de los que no existe ninguna otra implementación paralela con paso de mensajes. Estos métodos proporcionan una alternativa eficiente a la utilización del producto cruzado para el cálculo de la SVD.

La implementación realizada en SLEPc, de cada uno de los métodos, se ha validado mediante una batería de pruebas con matrices procedentes de aplicaciones reales. Las prestaciones paralelas se han medido en máquinas tipo cluster,

como MareNostrum, comprobando una buena escalabilidad incluso con un número muy grande de procesadores. Primero se ha comprobado que la implementación del método de Krylov-Schur en SLEPc obtiene mejores prestaciones que la de Trilinos, el único paquete disponible con el mismo método. Finalmente, esta implementación tiene unas prestaciones competitivas con respecto de ARPACK, el referente hasta la fecha para este tipo de software. Además, el método de Krylov-Schur presenta ventajas numéricas con respecto del reinicio implícito de ARPACK, y SLEPc proporciona una interfaz de usuario más potente y sencilla que la de ARPACK.

5.1. Producción científica

El trabajo realizado en esta tesis se ha presentado en varios congresos y ha dado lugar a un par de publicaciones en revistas indexadas por el JCR. Una primera versión del procedimiento clásico de Gram-Schmidt con refinamiento retrasado (sección 3.1.4) se presentó en el congreso PARCO 2005 con el título *A parallel variant of the Gram-Schmidt process with reorthogonalization*[HRT06b]. Las mejoras presentadas en esta tesis del procedimiento de Gram-Schmidt han sido publicadas en el artículo *Parallel Arnoldi eigensolvers with enhanced scalability via global communications rearrangement* de la revista *Parallel Computing* [HRT07c]. En este artículo se utiliza el método de Arnoldi con reinicio explícito para comparar las prestaciones de las distintas variantes de Gram-Schmidt presentadas, de la misma forma que en la sección 4.2.2. En el congreso VECPAR 2006 se presentaron las variantes del método de Lanczos con reinicio explícito *Evaluation of several variants of explicitly restarted Lanczos eigensolvers and their parallel implementations* [HRT07a] haciendo una comparativa similar a la sección 4.3.2 de las distintas estrategias de reortogonalización . En las Jornadas de Paralelismo del 2006 se presentó un resumen de lo publicado hasta ese momento, *Paralelización de Métodos de Krylov para Cálculo de Valores Propios* [HRT06a] combinando el método de Arnoldi, las variantes del método de Lanczos y las mejoras para Gram-Schmidt clásico.

La implementación del método de Krylov-Schur se presentó en el congreso ICIAM 2007 bajo el título *A parallel Krylov-Schur implementation for large Hermitian and non-Hermitian eigenproblems* [HRT07d], comparando sus prestaciones con ARPACK de forma parecida a la sección 4.4.2. En las Jornadas de Paralelismo del 2007 se presentó *Paralelización Híbrida con OpenMP y MPI de Métodos de Krylov para el Cálculo de Valores Propios* [HRT07b] donde se expuso

el modelo de paralelismo híbrido y los resultados de la sección 4.7. El método de Golub-Kahan-Lanczos para problemas de valores singulares junto con los resultados de las secciones 4.5.2 y 4.6.2 dieron lugar al trabajo *A robust and efficient parallel SVD solver based on restarted lanczos bidiagonalization* presentado en el congreso Harrachov 2007 [HRT07e]. La ampliación de este trabajo a un artículo del mismo título, con especial énfasis en el método de Golub-Kahan-Lanczos con reortogonalización por un lado y reinicio grueso, ha sido publicado en la revista *Electronic Transactions on Numerical Analysis* [HRT08].

Por otro lado, los detalles de la implementación realizada para esta tesis están documentados en una serie de informes técnicos que amplían el manual de usuario de SLEPc [HRTV07h]. El primero [HRTV07d], describe las variantes del procedimiento de Gram-Schmidt. [HRTV07a] y [HRTV07c] detallan los métodos de Arnoldi y las variantes de Lanczos con reinicio explícito. [HRTV07b] explica la implementación de método de Krylov-Schur tanto para problemas Hermitianos como no Hermitianos. Y por último, [HRTV07e] presenta el método de Golub-Kahan-Lanczos con reinicio explícito y reinicio grueso.

5.2. Proyectos de investigación

Parte del trabajo realizado en esta tesis ha sido financiado por la Generalitat Valenciana, Dirección General de Investigación y Transferencia de Tecnología con el proyecto *Técnicas de Aceleración para Algoritmos de Cálculo de Valores Propios en SLEPc* con referencia GV06/091.

5.3. Software basado en SLEPc

El código fuente de SLEPc está disponible a través de Internet desde el año 2003, lo que ha propiciado su uso en software especializado para aplicaciones que necesitan resolver problemas de valores propios. A continuación se describe brevemente este software y su relación con SLEPc. En el caso particular de software para mallas procedentes de elementos finitos, SLEPc se utiliza si la aplicación del usuario requiere resolver un problema de valores propios.

FEniCS es un software gratuito para la resolución automática de ecuaciones diferenciales. Proporciona herramientas para mallado, ecuaciones en derivadas parciales y métodos para ecuaciones diferenciales ordinarias [Log07].

libMesh proporciona una librería para la simulación numérica de ecuaciones en derivadas parciales utilizando discretizaciones arbitrarias mediante mallas no estructuradas. El principal objetivo de esta librería es proporcionar soporte al refinamiento adaptativo de la malla, mientras se mantiene un enfoque de alto nivel [KPSC06].

Hermes (Higher-order modular finite element system) es un software para elementos finitos desarrollado por el grupo hp-FEM de la universidad de Nevada y el instituto de Termomecánica de Praga [VSZ07].

Elefant (Efficient learning, large-scale inference, and optimization toolkit) es un librería para desarrollar algoritmos de aprendizaje [GWS+07]. Está desarrollada con Python, e incorpora una interfaz para acceder a la funcionalidad de SLEPc desde este lenguaje.

SIPs es un paquete que implementa múltiples transformaciones espectrales de desplazamiento e inversión. Permite calcular de forma eficiente muchos valores propios de problemas simétricos generalizados de naturaleza dispersa y gran dimensión [ZSSZ07].

TiberCAD es un paquete de simulación para dispositivos nanoelectrónicos y optoelectrónicos basado en un enfoque a distintas escalas [AdMPSDC07]. Permite combinar varios modelos físicos en distintas regiones de una misma simulación. Utiliza SLEPc para resolver ecuaciones de Schrödinger y problemas de valores propios en electromagnetismo.

5.4. Publicaciones con referencias a SLEPc

La disponibilidad del código fuente de SLEPc ha permitido que sea utilizado para el cálculo de valores propios en trabajos de investigación. A continuación se enumeran aquellos más relevantes de los que el autor tiene constancia, clasificados según el área de aplicación.

Análisis de vibraciones

- José Miguel Alonso y Vicente Hernández: *A parallel implementation of three-dimensional modal analysis of building structures*. En *Proceedings of the Eighth International Conference on Computational Structures Technology*. Civil-Comp Press, 2006.

Ingeniería nuclear

- D. Gilbert, J. E. Roman, Wm. J. Garland y W. F. S. Poehlman: *Simulating control rod and fuel assembly motion using moving meshes*. Annals of Nuclear Energy, 35(2):291–303, 2008.

Modelos electromagnéticos y fotónicos

- G. Li y A. Xu: *Analysis of the* TE*-pass or* TM*-pass metal-clad polarizer with a resonant buffer layer*. Lightwave Technology, Journal of, 26(10):1234–1241, Mayo 2008.
- Benjamin G. Ward: *Bend performance-enhanced photonic crystal fibers with anisotropic numerical aperture*. Optics Express, 16(12):8532–8548, 2008.

Modelos de plasma

- Luca Guazzotto: *Equilibrium and Stability of Tokamak Plasmas with Arbitrary Flow*. Tesis de Doctorado, University of Rochester, Junio 2005.
- M. Kammerer, F. Merz y F. Jenko: *Exceptional points in linear gyrokinetics*. Physics of Plasmas, 15(5):052102, 2008.

Modelos físicos, ciencia de materiales y estructura electrónica

- Leslie O. Baksmaty, Constantine Yannouleas y Uzi Landman: *Rapidly rotating boson molecules with long- or short-range repulsion: An exact diagonalization study*. Physical Review A (Atomic, Molecular, and Optical Physics), 75(2):023620, 2007.
- Michael Griebel y Jan Hamaekers: *Sparse grids for the Schrödinger equation*. Mathematical Modelling and Numerical Analysis, 41(2):215–247, 2007.
- Ville Lahtinen, Graham Kells, Angelo Carollo, Tim Stitt, Jiri Vala y Jiannis K. Pachos: *Spectrum of the non-abelian phase in Kitaev's honeycomb lattice model*. Annals of Physics, 323(9):2286–2310, 2008.
- Xiang Ma y Nicholas Zabaras: *A stabilized stochastic finite element second-order projection method for modeling natural convection in random porous media*. Journal of Computational Physics, 227(18):8448–8471, 2008.

- Seungil Kim y Joseph E. Pasciak: *The computation of resonances in open system susing a perfectly matched layer.* Pendiente de publicación, 2008.
- M. Taillefumier, V. K. Dugaev, B. Canals, C. Lacroix y P. Bruno: *Chiral two-dimensional electron gas in a periodic magnetic field: Persistent current and quantized anomalous Hall effect.* Physical Review B (Condensed Matter and Materials Physics), 78(15):155330, 2008.

Modelos químicos

- D. M. Medvedev, S. K. Gray, A. F. Wagner, M. Minkoff y R. Shepard: *Advanced software for the calculation of thermochemistry, kinetics, and dynamics.* Journal of Physics: Conference Series, 16:247–251, 2005.

Otras aplicaciones

- O. Bashir, K. Willcox, O. Ghattas, B. van Bloemen Waanders y J. Hill: *Hessian-based model reduction for large-scale systems with initial-condition inputs.* International Journal for Numerical Methods in Engineering, 73(6):844–868, 2008.
- Marina Chugunova y Dmitry Pelinovsky: *Spectrum of a non-self-adjoint operator associated with the periodic heat equation.* Journal of Mathematical Analysis and Applications, 342(2):970–988, 2008.
- Charlotte Truchet, Damien Noguès y Narendra Jussien: *Un modèle markovien pour GSAT et WalkSAT – résultats préliminaires.* En Quatrièmes Journées Francophones de Programmation par Contraintes (JFPC'08), páginas 327–336, Nantes, France, Junio 2008.
- Paulo Vasconcelos, Osni Marques y Jose Roman: *Parallel eigensolvers for a discretized radiative transfer problem.* En *High Performance Computing for Computational Science - VECPAR, 2008.*

5.5. Trabajos futuros

En primer lugar, se pueden utilizar otros esquemas de almacenamiento para los vectores de la base en lugar de las primitivas proporcionadas por PETSc. De esta forma se permitiría implementar algunas operaciones de los métodos

mediante productos de matrices, obteniendo un mejor aprovechamiento de la jerarquía de memoria. También se podría reducir el coste de calcular los valores y vectores propios de la matriz proyectada en los problemas Hermitianos, aprovechando su particular estructura simétrica en el método de Krylov-Schur. La misma idea se puede aplicar al cálculo de los valores y vectores singulares en el método de Golub-Kahan-Lanczos con reinicio grueso.

Otra posibilidad es la implementación a bloques de los métodos actuales, lo que supondría un gran esfuerzo dado que actualmente PETSc no proporciona las primitivas necesarias, en particular el producto eficiente de una matriz dispersa con varios vectores. El modelo de paralización híbrido presenta el mismo inconveniente, puesto que no está disponible en la versión actual de PETSc.

Los métodos presentados en esta tesis tienen una buena velocidad de convergencia a los valores propios dominantes. Aunque las técnicas de reinicio permiten modificar la convergencia hacia otros valores, no resultan competitivas con la aplicación de transformaciones espectrales. Para evitar el coste de estas transformaciones, se puede modificar el método de Krylov-Schur para que reinicie con otro tipo de vectores, como los vectores armónicos y/o refinados, más adecuados para el cálculo de valores interiores. Por último, siempre queda la posibilidad de añadir a SLEPc otros métodos totalmente diferentes, como los métodos de Davidson o Jacobi-Davidson vistos en la sección 1.1.4.

Bibliografía

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney y D. Sorensen: *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

[Abd71] N. N. Abdelmalek: *Roundoff error analysis for Gram–Schmidt method and solution of linear least squares problems*. BIT Numerical Mathematics, 11:345–368, 1971.

[ADLK01] Patrick R. Amestoy, Iain S. Duff, Jean Yves L'Excellent y Jacko Koster: *A fully asynchronous multifrontal solver using distributed dynamic scheduling*. SIAM Journal on Matrix Analysis and Applications, 23(1):15–41, 2001.

[AdMPSDC07] M. Auf der Maur, M. Povolotskyi, F. Sacconi y A. Di Carlo: *Tibercad: A new multiscale simulator for electronic and optoelectronic devices*. Superlattices and Microstructures, 41(5-6):381–385, 2007.

[AG99] Cleve Ashcraft y Roger Grimes: *SPOOLES: An object-oriented sparse matrix library*. En *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[AHLT05] Peter Arbenz, Ulrich L. Hetmaniuk, Richard B. Lehoucq y Raymond S. Tuminaro: *A comparison of eigensolvers for large-scale 3D modal analysis using AMG-preconditioned iterative methods*. International Journal for Numerical Methods in Engineering, 64(2):204–236, 2005.

[AR06] S. Akhter y J. Roberts: *Multi-Core Programming*. Intel Press, 2006.

[Arn51] W. E. Arnoldi: *The principle of minimized iterations in the solution of the matrix eigenvalue problem*. Quarterly of Applied Mathematics, 9:17–29, 1951.

[Bag08] James Baglama: *Augmented block Householder Arnoldi method*. Linear Algebra and its Applications, 429(10):2315–2334, 2008.

[BBE+07] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matt Knepley, Lois Curfman McInnes, Barry F. Smith y Hong Zhang: *PETSc users manual*. Informe técnico ANL-95/11 - Revision 2.3.3, Argonne National Laboratory, 2007.

[BBG+07] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith y Hong Zhang: *PETSc Web page*, 2007. `http://www.mcs.anl.gov/petsc`.

[BCC+97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker y R. C. Whaley: *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[BCR98] J. Baglama, D. Calvetti y L. Reichel: *Fast Leja points*. Electronic Transactions on Numerical Analysis, 7:126–140, 1998.

[BCR03a] J. Baglama, D. Calvetti y L. Reichel: *Algorithm 827:* `irbleigs`*: A MATLAB program for computing a few eigenpairs of a large sparse Hermitian matrix*. ACM Transactions on Mathematical Software, 29(3):337–348, Septiembre 2003.

[BCR03b] J. Baglama, D. Calvetti y L. Reichel: *IRBL: An implicitly restarted block-Lanczos method for large-scale Hermitian eigenproblems*. SIAM Journal on Scientific Computing, 24(5):1650–1677, Septiembre 2003.

[BD93] Z. Bai y J. W. Demmel: *On swapping diagonal blocks in real Schur form*. Linear Algebra and its Applications, 186:75–95, Junio 1993.

[BDD+00] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe y H. van der Vorst (editores): *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2000.

[BDDD97] Z. Bai, D. Day, J. Demmel y J. Dongarra: *A test matrix collection for non-Hermitian eigenvalue problems (release 1.0)*. Informe técnico CS-97-355, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, 1997. `http://math.nist.gov/MatrixMarket`.

[BEK00] Z. Bai, T. Ericsson y T. Kowalski: *Symmetric indefinite Lanczos method*. En Z. Bai, J. Demmel, J. Dongarra, A. Ruhe y H. van der Vorst (editores): *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, páginas 249–260. Society for Industrial and Applied Mathematics, Philadelphia, 2000.

[Ber92] Michael W. Berry: *SVDPACK: A FORTRAN-77 software library for the sparse singular value decomposition*. Informe técnico UT-CS-92-159, Department of Computer Science, University of Tennessee, Junio 1992.

[BGMS97] Satish Balay, William D. Gropp, Lois Curfman McInnes y Barry F. Smith: *Efficient management of parallelism in object oriented numerical software libraries*. En E. Arge, A. M. Bruaset y H. P. Langtangen (editores): *Modern Software Tools in Scientific Computing*, páginas 163–202. Birkhäuser Press, 1997.

[BHLT07] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq y H. K. Thornquist: *Anasazi software for the numerical solution of large-scale eigenvalue problems*. Informe técnico SAND 2007-0350J, Sandia National Laboratories, 2007.

[Bjö96] Å. Björck: *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[BN07] M. Bollhöfer y Y. Notay: *JADAMILU: a software code for computing selected eigenvalues of large sparse symmetric matrices*. Computer Physics Communications, 177:951–964, 2007.

[BP02] Luca Bergamaschi y Mario Putti: *Numerical comparison of iterative eigensolvers for large sparse symmetric positive definite matrices*. Computer Methods in Applied Mechanics and Engineering, 191(45):5233–5247, Octubre 2002.

[BR05] James Baglama y Lothar Reichel: *Augmented implicitly restarted Lanczos bidiagonalization methods*. SIAM Journal on Scientific Computing, 27(1):19–42, Enero 2005.

[BR06] James Baglama y Lothar Reichel: *Restarted block Lanczos bidiagonalization methods*. Numerical Algorithms, 43(3):251–272, Marzo 2006.

[Bra93] T. Braconnier: *The Arnoldi-Tchebycheff algorithm for solving large nonsymmetric eigenproblems*. Informe técnico TR/PA/93/25, CERFACS, Toulouse, France, 1993.

[BS97] Z. Bai y G. W. Stewart: *Algorithm 776: SRRIT: A FORTRAN subroutine to calculate the dominant invariant subspace of a nonsymmetric matrix*. ACM Transactions on Mathematical Software, 23(4):494–513, Diciembre 1997.

[CDK+01] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald y R. Menon: *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[CRS94] D. Calvetti, L. Reichel y D. C. Sorensen: *An implicitly restarted Lanczos method for large symmetric eigenvalue problems*. Electronic Transactions on Numerical Analysis, 2:1–21, 1994.

[Cup80] J. J. M. Cuppen: *A divide and conquer method for the symmetric tridiagonal eigenproblem*. Numerische Mathematik, 36(2):177–195, 1980.

[CW85] Jane K. Cullum y Ralph A. Willoughby: *Lanczos Algorithms for Large Symmetric Eigenvalue Computations. Vol. 1: Theory*. Birkhaüser, Boston, MA, 1985. Reeditado por SIAM, Philadelphia, 2002.

[CWL83] J. Cullum, R. A. Willoughby y M. Lake: *A Lanczos algorithm for computing singular values and vectors of large matrices*. SIAM Journal on Scientific and Statistical Computing, 4:197–215, 1983.

[Dav75] Ernest R. Davidson: *The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices.* Journal of Computational Physics, 17(1):87–94, 1975.

[Dav92] T. Davis: *University of Florida Sparse Matrix Collection.* Informe técnico 42, NA Digest, 1992. `http://www.cise.ufl.edu/research/sparse/matrices`.

[DEK95] Jack J. Dongarra, Victor L. Eijkhout y Ajay Kalhan: *Reverse communication interface for linear algebra templates for iterative methods.* LAPACK working note 99, Department of Computer Science, University of Tennessee, Knoxville, Mayo 1995.

[Dem97] James W. Demmel: *Applied Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[DGKS76] J. W. Daniel, W. B. Gragg, L. Kaufman y G. W. Stewart: *Reorthogonalization and stable algorithms for updating the Gram–Schmidt QR factorization.* Mathematics of Computation, 30(136):772–795, Octubre 1976.

[DGL89] I. S. Duff, R. G. Grimes y J. G. Lewis: *Sparse matrix test problems.* ACM Transactions on Mathematical Software, 15(1):1–14, Marzo 1989.

[Dhi98] Inderjit Singh Dhillon: *A new O (n^2) algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem.* Tesis de Doctorado, University of California at Berkeley, 1998.

[DK90] James W. Demmel y W. Kahan: *Accurate singular values of bidiagonal matrices.* SIAM Journal on Scientific and Statistical Computing, 11(5):873–912, 1990.

[DMPV08] James W. Demmel, Osni A. Marques, Beresford N. Parlett y Christof Vomel: *Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers.* SIAM Journal on Scientific Computing, 30(3):1508–1528, 2008.

[DMS97] Jack J. Dongarra, Hans W. Meuer y Erich Strohmaier: *Top500 supercomputer sites*. Informe técnico, Supercomputer, 1997. `http://www.top500.org`.

[DPV04] Inderjit S. Dhillon, Beresford N. Parlett y Christof Vömel: *The design and implementation of the MRRR algorithm*. LAPACK working note 162, Department of Computer Science, University of Tennessee, Knoxville, Diciembre 2004. UT-CS-04-541.

[EES83] Stanley C. Eisenstat, Howard C. Elman y Martin H. Schultz: *Variational iterative methods for nonsymmetric systems of linear equations*. SIAM Journal on Numerical Analysis, 20(2):345–357, 1983.

[FN96] Roland W. Freund y Noël M. Nachtigal: *QMRPACK: a package of QMR algorithms*. ACM Transactions on Mathematical Software, 22(1):46–77, Marzo 1996.

[FSdV99] Diederik R. Fokkema, Gerard L. G. Sleijpen y Henk A. Van der Vorst: *Jacobi–Davidson style QR and QZ algorithms for the reduction of matrix pencils*. SIAM Journal on Scientific Computing, 20(1):94–125, Enero 1999.

[FV99] J. Frank y C. Vuik: *Parallel implementation of a multiblock method with approximate subdomain solution*. Applied Numerical Mathematics: Transactions of IMACS, 30(4):403–423, Julio 1999.

[FY08] Robert Falgout y Ulrike Yang: *hypre: A library of high performance preconditioners*. Computational Science —ICCS 2002, páginas 632–641, 2008.

[GE95] Ming Gu y Stanley C. Eisenstat: *A divide-and-conquer algorithm for the bidiagonal SVD*. SIAM Journal on Matrix Analysis and Applications, 16(1):79–92, 1995.

[Geu02] Roman Geus: *The Jacobi-Davidson algorithm for solving large sparse symmetric eigenvalue problems with application to the design of accelerator cavities*. Tesis de Doctorado, ETH Zürich, 2002. Núm. 14.734.

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham y Timothy S. Woodall: *Open MPI: Goals, concept, and design of a next generation MPI implementation.* En *Proceedings, 11th European PVM/MPI Users' Group Meeting*, páginas 97–104, Budapest, Hungary, Septiembre 2004.

[GGKK03] A. Grama, A. Gupta, G. Karypis y V. Kumar: *Introduction to Parallel Computing.* Addison-Wesley, segunda edición, 2003.

[GK65] G. H. Golub y W. Kahan: *Calculating the singular values and pseudo-inverse of a matrix.* Journal of the Society for Industrial and Applied Mathematics, Series B, Numerical Analysis, 2:205–224, 1965.

[GL96] William D. Gropp y Ewing Lusk: *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[GL02] Luc Giraud y Julien Langou: *When modified Gram-Schmidt generates a well-conditioned set of vectors.* IMA Journal of Numerical Analysis, 22:521–528, Octubre 2002.

[GLDS96] W. Gropp, E. Lusk, N. Doss y A. Skjellum: *A high-performance, portable implementation of the MPI message passing interface standard.* Parallel Computing, 22(6):789–828, Septiembre 1996.

[GLS99] W. Gropp, E. Lusk y A. Skjellum: *Using MPI: Portable Parallel Programming with the Message–Passing Interface.* The MIT Press, segunda edición, 1999.

[Grc81] Joseph F. Grcar: *Analyses of the Lanczos algorithm and of the approximation problem in Richardson's method.* Informe técnico 1074, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1981.

[GSF92] Giuseppe Gambolati, Flavio Sartoretto y Paolo Florian: *An orthogonal accelerated deflation technique for large symmetric eigenproblems.* Computer Methods in Applied Mechanics and Engineering, 94(1):13–23, 1992.

[GU77] G. H. Golub y R. Underwood: *The block Lanczos method for computing eigenvalues*. En J. Rice (editor): *Mathematical Software III*, páginas 364–377. Academic Press, New York, NY, USA, 1977.

[GVL96] G. H. Golub y C. F. Van Loan: *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, tercera edición, 1996.

[GWS+07] Kishor Gawande, Christfried Webers, Alex Smola, S. V. N. Vishwanathan, Simon Günter, Choon Hui Teo, Javen Qinfeng Shi, Julian McAuley, Le Song y Quoc Le: *ELEFANT user manual (revision 0.1)*. Informe técnico, NICTA, 2007. http://elefant.developer.nicta.com.au.

[GY03] Gene H. Golub y Qiang Ye: *An inverse free preconditioned Krylov subspace method for symmetric generalized eigenvalue problems*. SIAM Journal on Scientific Computing, 24(1):312–334, Enero 2003.

[HBH+05] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams y Kendall S. Stanley: *An overview of the Trilinos project*. ACM Transactions on Mathematical Software, 31(3):397–423, 2005.

[Hof89] Walter Hoffmann: *Iterative algorithms for Gram-Schmidt orthogonalization*. Computing, 41(4):335–348, 1989.

[HRT06a] V. Hernandez, J. E. Roman y A. Tomas: *Paralelización de Métodos de Krylov para Cálculo de Valores Propios*. En *Actas de las XVII Jornadas de Paralelismo*, páginas 455–460, Septiembre 2006.

[HRT06b] V. Hernandez, J. E. Roman y A. Tomas: *A parallel variant of the Gram-Schmidt process with reorthogonalization*. En Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, Oscar G. Plata, P. Tirado y Emilio L. Zapata (editores): *Proceedings of the International Conference on Parallel Computing (ParCo 2005)*, volumen 33, páginas 221–228. Central Institute for Applied Mathematics, Jülich, Germany, 2006.

[HRT07a] V. Hernandez, J. E. Roman y A. Tomas: *Evaluation of several variants of explicitly restarted Lanczos eigensolvers and their parallel implementations*. En *High Performance Computing for Computational Science - VECPAR 2006*, volumen 4395 de *Lecture Notes in Computer Science*, páginas 403–416, 2007.

[HRT07b] V. Hernandez, J. E. Roman y A. Tomas: *Paralelización Híbrida con OpenMP y MPI de Métodos de Krylov para el Cálculo de Valores Propios*. En *Actas de las XVIII Jornadas de Paralelismo*, Septiembre 2007.

[HRT07c] V. Hernandez, J. E. Roman y A. Tomas: *Parallel Arnoldi eigensolvers with enhanced scalability via global communications rearrangement*. Parallel Computing, 33(7–8):521–540, Agosto 2007.

[HRT07d] V. Hernandez, J. E. Roman y A. Tomas: *A parallel Krylov-Schur implementation for large Hermitian and non-Hermitian eigenproblems*. En *ICIAM 2007, 6th International Congress on Industrial and Applied Mathematics*, volumen 7 de *PAAM*, páginas 2020083–2020084, 2007.

[HRT07e] V. Hernandez, J. E. Roman y A. Tomas: *A robust and efficient parallel SVD solver based on restarted Lanczos bidiagonalization*. En *Harrachov 2007, Computational Linear Algebra with Applications*, página 85, Prague, Agosto 2007. Institute of Computer Science AS CR.

[HRT08] V. Hernandez, J. E. Roman y A. Tomas: *A robust and efficient parallel SVD solver based on restarted Lanczos bidiagonalization*. Electronic Transactions on Numerical Analysis, 31:68–85, 2008.

[HRTV07a] V. Hernandez, J. E. Roman, A. Tomas y V. Vidal: *Arnoldi methods in SLEPc*. Informe técnico STR-4, Universidad Politécnica de Valencia, 2007. Disponible en `http://www.grycap.upv.es/slepc/documentation/reports/str4.pdf`.

[HRTV07b] V. Hernandez, J. E. Roman, A. Tomas y V. Vidal: *Krylov-Schur methods in SLEPc*. Informe técnico STR-7, Universidad Politécnica de Valencia, 2007. Disponible en `http://www.grycap.upv.es/slepc/documentation/reports/str7.pdf`.

[HRTV07c] V. Hernandez, J. E. Roman, A. Tomas y V. Vidal: *Lanczos methods in SLEPc*. Informe técnico STR-5, Universidad Politécnica de Valencia, 2007. Disponible en `http://www.grycap.upv.es/slepc/documentation/reports/str5.pdf`.

[HRTV07d] V. Hernandez, J. E. Roman, A. Tomas y V. Vidal: *Orthogonalization routines in SLEPc*. Informe técnico STR-1, Universidad Politécnica de Valencia, 2007. Disponible en `http://www.grycap.upv.es/slepc/documentation/reports/str1.pdf`.

[HRTV07e] V. Hernandez, J. E. Roman, A. Tomas y V. Vidal: *Restarted Lanczos bidiagonalization for the SVD in SLEPc*. Informe técnico STR-8, Universidad Politécnica de Valencia, 2007. Disponible en `http://www.grycap.upv.es/slepc/documentation/reports/str8.pdf`.

[HRTV07f] V. Hernandez, J. E. Roman, A. Tomas y V. Vidal: *Single vector iteration methods in SLEPc*. Informe técnico STR-2, Universidad Politécnica de Valencia, 2007. Disponible en `http://www.grycap.upv.es/slepc/documentation/reports/str2.pdf`.

[HRTV07g] V. Hernandez, J. E. Roman, A. Tomas y V. Vidal: *SLEPc home page*. http://www.grycap.upv.es/slepc, 2007.

[HRTV07h] V. Hernandez, J. E. Roman, A. Tomas y V. Vidal: *SLEPc users manual*. Informe técnico DSIC-II/24/02 - Revision 2.3.3, D. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2007. Disponible en `http://www.grycap.upv.es/slepc/documentation/slepc.pdf`.

[HRTV07i] V. Hernandez, J. E. Roman, A. Tomas y V. Vidal: *Subspace iteration in SLEPc*. Informe técnico STR-3, Universidad Politécnica de Valencia, 2007. Disponible en `http://www.grycap.upv.es/slepc/documentation/reports/str3.pdf`.

[HRTV07j] V. Hernandez, J. E. Roman, A. Tomas y V. Vidal: *A survey of software for sparse eigenvalue problems*. Informe técnico STR-6, Universidad Politécnica de Valencia, 2007. Disponible en `http://www.grycap.upv.es/slepc/documentation/reports/str6.pdf`.

[HRV03] V. Hernandez, J. E. Roman y V. Vidal: *SLEPc: Scalable Library for Eigenvalue Problem Computations*. Lecture Notes in Computer Science, 2565:377–391, 2003.

[HRV05] V. Hernandez, J. E. Roman y V. Vidal: *SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems*. ACM Transactions on Mathematical Software, 31(3):351–362, Septiembre 2005.

[IEE93] *IEEE standard for scalable coherent interface (SCI)*. IEEE Std 1596-1992, Agosto 1993.

[Ips97] Ilse C. F. Ipsen: *Computing an eigenvector with inverse iteration*. SIAM Review, 39(2):254–291, 1997.

[Jia02] Zhongxiao Jia: *The refined harmonic Arnoldi method and an implicitly restarted refined algorithm for computing interior eigenpairs of large matrices*. Applied Numerical Mathematics: Transactions of IMACS, 42(4):489–512, 2002.

[JP93] M. T. Jones y M. L. Patrick: *The Lanczos algorithm for the generalized symmetric eigenproblem on shared-memory architectures*. Applied Numerical Mathematics: Transactions of IMACS, 12(5):377, 1993.

[Kah66] William Kahan: *Accurate eigenvalues of a symmetric tri-diagonal matrix*. Informe técnico CS-TR-66-41, Stanford University, Stanford, CA, USA, 1966.

[KC92] S. K. Kim y A. T. Chronopoulos: *An efficient parallel algorithm for extreme eigenvalues of sparse nonsymmetric matrices*. International Journal of Supercomputer Applications, 6(1):98–111, 1992.

[Kny01] A. V. Knyazev: *Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method*. SIAM Journal on Scientific and Statistical Computing, 23(2):517–541, 2001.

[KPSC06] Benjamin Kirk, John Peterson, Roy Stogner y Graham Carey: *libmesh: A C++ library for parallel adaptive mesh refinement/coarsening simulations*. Engineering with Computers, 22(3):237–254, 2006.

[Lan50] C. Lanczos: *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators.* Journal of Research of the National Bureau of Standards, 45:255–282, 1950.

[Lar98] Rasmus Munk Larsen: *Lanczos bidiagonalization with partial reorthogonalization.* Informe técnico PB-537, Department of Computer Science, University of Aarhus, Aarhus, Denmark, 1998. Disponible en `http://www.daimi.au.dk/PB/537`.

[Lar01] Rasmus Munk Larsen: *Combining implicit restart and partial reorthogonalization in Lanczos bidiagonalization.* Informe técnico, SCCM, Stanford University, 2001. Disponible en `http://soi.stanford.edu/~rmunk/PROPACK`.

[LD03] Xiaoye S. Li y James W. Demmel: *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems.* ACM Transactions on Mathematical Software, 29(2):110–140, June 2003.

[Log07] A. Logg: *Automating the finite element method.* Archives of Computational Methods in Engineering, 14(2):93–138, 2007.

[LS96a] R. B. Lehoucq y J. A. Scott: *An evaluation of software for computing eigenvalues of sparse nonsymmetric matrices.* Informe técnico MCS-P547-1195, Argonne National Laboratory, 1996.

[LS96b] R. B. Lehoucq y D. C. Sorensen: *Deflation techniques for an implicitly restarted Arnoldi iteration.* SIAM Journal on Matrix Analysis and Applications, 17(4):789–821, 1996.

[LSY98] R. B. Lehoucq, D. C. Sorensen y C. Yang: *ARPACK Users' Guide, Solution of Large-Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998.

[Mar95] O. A. Marques: *BLZPACK: Description and user's guide.* Informe técnico TR/PA/95/30, CERFACS, Toulouse, France, 1995.

[Meu87] Gérard Meurant: *Multitasking the conjugate gradient method on the CRAY X-MP/48.* Parallel Computing, 5(3):267–280, Noviembre 1987.

[MPI94] MPI Forum: *MPI: a message-passing interface standard.* International Journal of Supercomputer Applications and High Performance Computing, 8(3/4):159–416, 1994.

[MS96] K. J. Maschhoff y D. C. Sorensen: *PARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures.* Lecture Notes in Computer Science, 1184:478–486, 1996.

[MS97] Karl Meerbergen y Alastair Spence: *Implicitly restarted Arnoldi with purification for the shift-invert transformation.* Mathematics of Computation, 66(218):667–689, Abril 1997.

[MSR94] K. Meerbergen, A. Spence y D. Roose: *Shift-invert and Cayley transforms for detection of rightmost eigenvalues of nonsymmetric matrices.* BIT Numerical Mathematics, 34(3):409–423, Septiembre 1994.

[NOPEJ87] Bahram Nour-Omid, Beresford N. Parlett, Thomas Ericsson y Paul S. Jensen: *How to implement the spectral transformation.* Mathematics of Computation, 48(178):663–673, Abril 1987.

[Not02] Y. Notay: *Combination of Jacobi-Davidson and conjugate gradients for the partial symmetric eigenproblem.* Numerical Linear Algebra with Applications, 9(1):21–44, 2002.

[Pai72] C. C. Paige: *Computational variants of the Lanczos method for the eigenproblem.* Journal of the Institute of Mathematics and its Applications, 10:373–381, 1972.

[Pai76] C. C. Paige: *Error analysis of the Lanczos algorithm for tridiagonalizing a symmetric matrix.* Journal of the Institute of Mathematics and its Applications, 18(3):341–349, 1976.

[Pai80] C. C. Paige: *Accuracy and effectiveness of the Lanczos algorithm for the symmetric eigenproblem.* Linear Algebra and its Applications, 34:235–258, 1980.

[Par80] B. N. Parlett: *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, NJ, 1980. Reeditado con revisiones por SIAM, Philadelphia, 1998.

[PL93] Beresford N. Parlett y Jian Le: *Forward instability of tridiagonal QR*. SIAM Journal on Matrix Analysis and Applications, 14(1):279–316, 1993.

[PM99] Beresford N. Parlett y Osni A. Marques: *An implementation of the DQDS algorithm (positive case)*. Linear Algebra and its Applications, 309:2000, 1999.

[PNO85] B. N. Parlett y B. Nour-Omid: *The use of a refined error bound when updating eigenvalues of tridiagonals*. Linear Algebra and its Applications, 68:179–219, 1985.

[PR81] B. N. Parlett y J. K. Reid: *Tracking the progress of the Lanczos algorithm for large symmetric eigenproblems*. IMA Journal of Numerical Analysis, 1(2):135–155, 1981.

[PS79] B. N. Parlett y D. S. Scott: *The Lanczos algorithm with selective orthogonalization*. Mathematics of Computation, 33:217–238, 1979.

[Rag02] P. Raghavan: *DSCPACK: Domain-separator codes for the parallel solution of sparse linear systems*. Informe técnico CSE-02-004, Department of Computer Science and Engineering, The Pennsylvania State University, 2002.

[Rut67] H. Rutishauser: *Description of Algol* 60. Handbook for Automatic Computation, Vol. 1a. Springer-Verlag, Berlin, 1967.

[Saa84] Y. Saad: *Chebyshev acceleration techniques for solving nonsymmetric eigenvalue problems*. Mathematics of Computation, 42:567–588, 1984.

[Saa89] Youcef Saad: *Krylov subspace methods on supercomputers*. SIAM Journal on Scientific and Statistical Computing, 10(6):1200–1232, Noviembre 1989.

[Saa92] Y. Saad: *Numerical Methods for Large Eigenvalue Problems: Theory and Algorithms*. John Wiley and Sons, New York, 1992.

[SF94] A. Stathopoulos y C. F. Fischer: *A Davidson program for finding a few selected extreme eigenpairs of a large, sparse, real, symmetric matrix*. Computer Physics Communications, 79:268, 1994.

[Sim84a] H. D. Simon: *Analysis of the symmetric Lanczos algorithm with reorthogonalization methods*. Linear Algebra and its Applications, 61:101–132, 1984.

[Sim84b] H. D. Simon: *The Lanczos algorithm with partial reorthogonalization*. Mathematics of Computation, 42(165):115–142, Enero 1984.

[SJ81] William J. Stewart y Alan Jennings: *Algorithm 570: LOPSI: A simultaneous iteration method for real matrices [F2]*. ACM Transactions on Mathematical Software, 7(2):230–232, Junio 1981.

[SM07] Andreas Stathopoulos y James R. McCombs: *Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part II: Seeking many eigenvalues*. SIAM Journal on Scientific Computing, 29(5):2162–2188, 2007.

[Sor92] D. C. Sorensen: *Implicit application of polynomial filters in a k-step Arnoldi method*. SIAM Journal on Matrix Analysis and Applications, 13:357–385, 1992.

[SS99] Miloud Sadkane y Roger B. Sidje: *Implementation of a variable block Davidson method with deflation for solving large sparse eigenproblems*. Numerical Algorithms, 20(2–3):217–240, Junio 1999.

[Sta07] Andreas Stathopoulos: *Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part I: Seeking one eigenvalue*. SIAM Journal on Scientific Computing, 29(2):481–514, 2007.

[Ste01a] G. W. Stewart: *A Krylov–Schur algorithm for large eigenproblems*. SIAM Journal on Matrix Analysis and Applications, 23(3):601–614, 2001.

[Ste01b] G. W. Stewart: *Matrix Algorithms. Volume II: Eigensystems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[SV96] Gerard L. G. Sleijpen y Henk A. Van Der Vorst: *A Jacobi–Davidson iteration method for linear eigenvalue problems*. SIAM Journal on Matrix Analysis and Applications, 17:401–425, 1996.

[SWTM01] Ron Shepard, Albert F. Wagner, Jeffrey L. Tilson y Michael Minkoff: *The subspace projected approximate matrix (SPAM) modification of the Davidson method.* Journal of Computational Physics, 172(2):472–514, 2001.

[SZ00] Horst D. Simon y Hongyuan Zha: *Low-rank matrix approximation using the Lanczos bidiagonalization process with applications.* SIAM Journal on Scientific Computing, 21(6):2257–2274, Noviembre 2000.

[VSZ07] Tomas Vejchodsky, Pavel Solin y Martin Zitka: *Modular hp-FEM system HERMES and its application to Maxwell's equations.* Mathematics and Computers in Simulation, 76(1-3):223–228, 2007.

[WA98] B. Wilkinson y M. Allen: *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice-Hall, 1998.

[Wil65] J. H. Wilkinson: *The Algebraic Eigenvalue Problem.* Claredon Press, Oxford, UK, 1965.

[WS00] Kesheng Wu y Horst Simon: *Thick-restart Lanczos method for large symmetric eigenvalue problems.* SIAM Journal on Matrix Analysis and Applications, 22(2):602–616, Abril 2000.

[ZSSZ07] Hong Zhang, Barry Smith, Michael Sternberg y Peter Zapol: *SIPs: Shift-and-invert parallel spectral transformations.* ACM Transactions on Mathematical Software, 33(2):9, 2007.