UNIVERSIDAD NACIONAL DE ASUNCIÓN

Facultad Politécnica

Evaluación de Gradiente Conjugado por Bloques para un Lado Derecho

por

Elias Maciel

Proyecto final de grado presentado como
requisito para obtener el grado de
Licenciado

en

Ciencias Informáticas

Orientadores:

Profesor Christian Schaerer, Tutor
Profesor Pedro Torres

Otoño 2017

Resumen

Evaluación de Gradiente Conjugado por Bloques para un Lado Derecho

por

Elias Maciel

Licenciatura en Ciencias Informáticas
con énfasis en Programación de Computadoras

Universidad Nacional de Asunción, San Lorenzo

Profesor Christian Schaerer, Tutor

El método de Gradiente Conjugado (CG) es un algoritmo empleado en la resolución de sistemas lineales, en donde la matriz que se busca resolver es simétrica definida positiva (SPD). CG posee un cuello de botella computacional provocado por el bajo rendimiento logrado con rutinas de multiplicación de matriz dispersa por vector en los sistemas actuales. De manera a aumentar el rendimiento, se han propuesto versiones por bloques, en donde en lugar de multiplicar la matriz dispersa por un solo vector se realiza la multiplicación por múltiples vectores simultáneamente. Para el CG por bloques (BCG) estudiado, se logra generar un mayor subespacio de búsqueda en cada iteración en comparación a la formulación clásica de este método. En este trabajo estudiamos el BCG para un solo lado derecho con una estrategia para la combinación de soluciones arrojadas por el método por bloques. De forma a conocer el rendimiento del BCG propuesto, realizamos un conjunto de pruebas con el fin de contrastar estos resultados con los de CG. Comparamos ambos algoritmos en términos de cantidad de iteraciones ejecutadas para llegar a la convergencia y tiempo de ejecución. Los resultados muestran que en la mayoría de los casos se puede lograr disminuir la cantidad de iteraciones a medida que se aumenta el tamaño de bloque. Sin embargo, para las configuraciones experimentales establecidas, en la mayoría de los casos obtuvimos tiempos de ejecución superiores para BCG en comparación a CG.

Abstract

Evaluation of Block Conjugate Gradient for a Single Right-Hand Side

by

Elias Maciel

Bachelor of Science in Information Technology
with Emphasis in Computer Programming

National University of Asuncion, San Lorenzo

Professor Christian Schaerer, Chair

Conjugate Gradient method (CG) is an algorithm used to solve linear systems in which the matrix to be solved is symmetric positive definite (SPD). CG has a computational bottleneck due to the low performance attained with routines that perform sparse matrix-vector multiplication in modern systems. In order to increase the performance, there have been proposed block versions, for which instead of multiplying the sparse matrix for a single vector the multiplication is carried out with multiple vectors simultaneously. For the studied Block CG (BCG), it can be generated a greater search subspace in each iteration in comparison to the classic formulation of this method. In this work we study the BCG method for a single right-hand side with a strategy to combine the solutions generated by the block method. In order to know the performance of the proposed BCG, we have conducted a number of tests to contrast the obtained results to those of CG. We compared both algorithms in terms of number of iterations executed to achieve convergence and elapsed time. The results show that in the majority of cases the number of iterations can be reduced as the block size is increased. However, for the experimental settings established, in most cases we obtained higher elapsed times for BCG compared to CG.

A mi madre

Índice general

Índice general II

Índice de figuras III

Índice de cuadros IV

1 Introducción 1
1.1. Contribuciones . . . 2
1.2. Objetivos . . . 2
1.3. Organización del trabajo . . . 2

2 Fundamento teórico 4
2.1. Cuestiones computacionales . . . 4
2.2. Revisión de la Literatura y Estado del Arte . . . 6

3 Algoritmos 9
3.1. Gradiente Conjugado . . . 9
3.2. Gradiente Conjugado por Bloques . . . 14

4 Experimentos numéricos 19
4.1. Construcción de una matriz SPD . . . 19
4.2. Configuración experimental . . . 21
4.3. Resultados . . . 23

5 Conclusiones 28

A Códigos 29
A.1. `cg.m` . . . 29
A.2. `bcg.m` . . . 31

Bibliografía 33

Índice de figuras

2.1. Intensidad aritmética para distintos kernels y métodos 5
2.2. Gráfica del modelo *Roofline* 6

4.1. Norma relativa del residuo en función al número de iteración para cfd1, ex3 y bundle1 23
4.2. Norma relativa del residuo en función al número de iteración para poisson10k, fv2 y qa8fm 24
4.3. Norma relativa del residuo en función al número de iteración para cvxbqp1, apache1, thermal1 25
4.4. Norma relativa del residuo en función al número de iteración para G2_circuit . . 26

Índice de cuadros

2.1. Intensidad Aritmética de SpMV y SpMM 7

4.1. Nombre e información estructural de cada una de las matrices utilizadas en las pruebas 22
4.2. Tiempos de ejecución de CG y BCG para las matrices seleccionadas 26

Agradecimientos

Primeramente agradezco al profesor Christian Schaerer por enseñarme Álgebra Lineal y por todos los conocimientos transmitidos.

También a Pedro Torres y a Juan Carlos Cabral por contribuir con la matemática y con los experimentos en este trabajo.

En general, al Laboratorio de Computación Científica Aplicada por brindarme acceso a sus instalaciones.

Finalmente, a mi madre y padre por el soporte que me dieron durante todos estos años.

Capítulo 1

Introducción

En problemas científicos y de ingeniería como análisis estructural, análisis de circuitos, métodos de elementos finitos [30, 32], surgen sistemas de ecuaciones lineales donde la matriz que representa al sistema es simétrica definida positiva (SPD). En muchos casos estas matrices son dispersas y de gran porte. Por estos motivos un método directo como factorización LU o Cholesky para la resolución de estos sistemas puede ser prohibitivo.

El método más popular para la resolución de este tipo de sistemas es el de Gradiente Conjugado (CG) [30], propuesto por Hestenes y Stiefel [18], el cual es un método basado en el *subespacio de Krylov*. A pesar de que este método garantiza la convergencia en n pasos con aritmética exacta, donde n es el orden de la matriz, posee un cuello de botella en términos computacionales que es la operación de multiplicación de matriz dispersa por vector (SpMV). Esta operación es necesaria para generar el subespacio de búsqueda de la solución. El motivo particular por el cual esta operación degrada el rendimiento general de este algoritmo es que alcanza una baja intensidad aritmética. Los detalles de este problema serán presentados y discutidos en el Capítulo 2.

Una variante del CG que ha sido estudiada en trabajos como [27, 6], es la versión por bloques (Block CG o BCG), en donde se busca resolver un sistema de ecuaciones lineales con múltiples lados derechos. Computacionalmente, esta versión se caracteriza por emplear de mejor manera los recursos de memoria, realizando más operaciones en relación a los datos transportados. De esta forma se alcanza una intensidad aritmética superior en comparación al CG.

Adicionalmente, como en la versión por bloques se operan con varios vectores en simultáneo por iteración, en cada paso se logra generar un subespacio de búsqueda mayor y, por ende, enriquecido. Esto puede reducir considerablemente la cantidad de iteraciones necesarias para llegar a la convergencia. Los detalles de la generación de los subespacios de búsqueda para CG y BCG se tratan en el Capítulo 3.

Por otra parte, en el trabajo de Chronopoulos y Kucherov [10] se propone un algoritmo para encontrar una combinación óptima de soluciones obtenidas por algún método por bloques para determinar la solución de un sistema con un único lado derecho. Esto en lugar de que cada solución del método por bloques se corresponda a un lado derecho en particular.

En este trabajo estudiamos una implementación de CG por bloques en conjunción con el algoritmo de combinación de soluciones para la resolución de sistemas de ecuaciones lineales con un único lado derecho. Impulsados por el enriquecimiento del subespacio de búsqueda que se logra con los métodos por bloques, y por la intensidad aritmética superior que se puede alcanzar mediante la operación con múltiples vectores.

La evaluación del algoritmo propuesto se realiza desde dos perspectivas. La cantidad de iteraciones ejecutadas para llegar a la convergencia y el tiempo de ejecución con las plataformas y configuraciones experimentales establecidas. Ambos criterios, en contraste a la implementación clásica del CG.

1.1. Contribuciones

Desarrollamos un algoritmo basado en el método de Gradiente Conjugado por bloques para la solución de matrices simétricas positivas definidas para un lado derecho mediante combinación de soluciones y escribimos el código de implementación del algoritmo en MATLAB.

1.2. Objetivos

Objetivo General

Conocer el desempeño del algoritmo de Gradiente Conjugado utilizando una estrategia por bloques y combinación de soluciones para un único lado derecho en comparación al método de Gradiente Conjugado clásico.

Objetivos Específicos

- Diseñar el algoritmo e implementarlo en un lenguaje de programación.
- Establecer criterios para medir y comparar el desempeño del algoritmo propuesto en contraste a su contraparte clásica.

1.3. Organización del trabajo

El trabajo se organiza de la siguiente forma. En el Capítulo 2 se presentan los fundamentos teóricos para la formulación de los algoritmos teniendo en cuenta las aplicaciones de los métodos propuestos y su desempeño en términos computacionales. Adicionalmente, realizamos una revisión de la literatura y del estado del arte con relación a la reformulación de algoritmos de manera a lograr rendimientos superiores.

Los algoritmos de Gradiente Conjugado clásico y Gradiente Conjugado por Bloques con combinación de soluciones propuesto en este trabajo se presentan y demuestran en el Capítulo 3.

En el Capítulo 4 detallamos las configuraciones establecidas para los experimentos y los resultados obtenidos. Además, comparamos los resultados de los métodos estudiados en base a un conjunto de criterios, específicamente, el número de iteraciones realizadas para lograr la convergencia y el tiempo de ejecución. Finalmente, se dan conclusiones generales y se discuten posibles trabajos futuros en el Capítulo 5.

Capítulo 2

Fundamento teórico

2.1. Cuestiones computacionales

Memory Wall

Es la continua disparidad entre las velocidades de la CPU y los distintos niveles de la jerarquía de memoria. Esto se debe principalmente a la división de las industrias de microprocesadores y dispositivos de memoria. Las primeras apuntaban a aumentar la velocidad de cómputos y las segundas apuntaban a tener mayor capacidad [24, 35].

Para dar una perspectiva sobre esta tendencia, desde el año 1980 al año 2000 la velocidad de la CPU aumentó a razón de 60 % por año, mientras que el tiempo de acceso a la DRAM (*Dynamic RAM*) aumentó a razón de menos del 10 % en el mismo período de tiempo [9].

En la actualidad, el costo de energía y tiempo para el transporte de datos (también llamado comunicación) entre los niveles de memoria y el procesador o entre procesadores en una red es mayor en órdenes de magnitud que la ejecución de operaciones de punto flotante [8]. La tasa de crecimiento de esta diferencia es exponencial y se estima que aumente en sistemas futuros [15].

Intensidad Aritmética

Como la comunicación es costosa resulta conveniente realizar la mayor cantidad de operaciones posibles con los datos transportados. En base a este aspecto se define la Intensidad Aritmética, que es la relación entre el número de computaciones efectuadas y la cantidad de datos transportados por sobre los cuales se opera. Generalmente se mide en $\frac{FLOPS}{Byte}$.

En la Figura 2.1 se observa que la multiplicación de matriz dispersa por vector denso (SpMV) alcanza valores muy bajos, se logra hasta $1\frac{FLOPS}{Byte}$. Mientras que para operaciones matriz-matriz (Dense Linear Algebra–BLAS 3) se logran hasta $10\frac{FLOPS}{Byte}$. Este número es un límite superior para el caso matriz dispersa por múltiples vectores (SpMM) que se utiliza en los métodos por bloques cuando la matriz es dispersa.

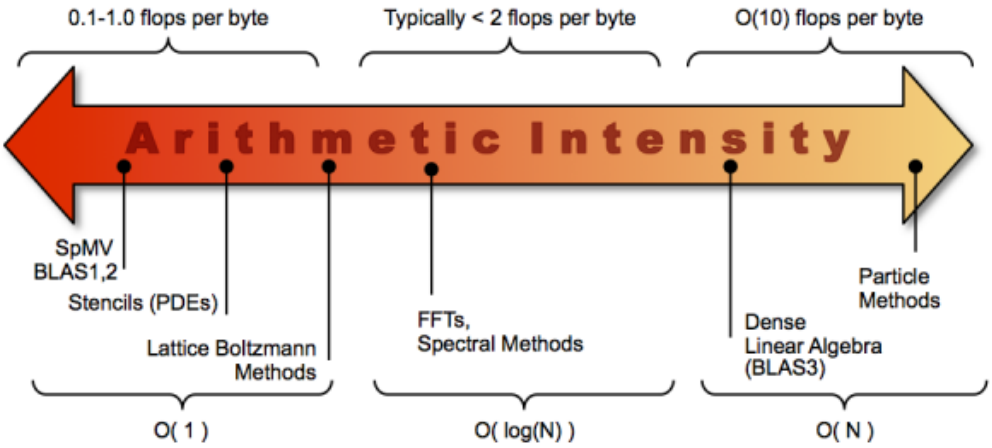


Figura 2.1: Intensidad aritmética para distintos kernels y métodos.
Fuente: `https://crd.lbl.gov/departments/computer-science/PAR/research/roofline`

Modelo *Roofline*

Es un modelo que indica el rendimiento del procesador en base a las operaciones de punto flotante que puede alcanzar a efectuar por unidad de tiempo (típicamente GFLOPS/s) en función a la intensidad arimética [28, 34]. Se aplica principalmente para conocer el rendimiento pico real de un procesador (a diferencia del rendimiento pico teórico) teniendo como parámetro la intensidad aritmética lograda con un determinado algoritmo.

Para que una unidad de procesamiento alcance su rendimiento máximo se requiere que los datos a ser procesados y las instrucciones estén disponibles para el procesador en los niveles más altos de la jerarquía de memoria. Es decir, la unidad de procesamiento no debe quedar ociosa esperando a que se transporten los datos entre los niveles de memoria.

La Figura 2.2 muestra la gráfica del rendimiento de un procesador en base a la intensidad aritmética obtenida. La recta horizontal (Peak performance) representa el límite práctico para el rendimiento de la unidad del procesamiento. Adicionalmente, se muestran los rendimientos alcanzados con un procesador corriendo dos algoritmos con intensidades aritméticas diferentes.

En el ejemplo de la Figura 2.2, Alg1 es un algoritmo limitado por el ancho de banda (*memory-bound computation*), que está relacionado con el costo de comunicación. Mientras que con Alg2 se logra utilizar toda la capacidad de procesamiento de la CPU (*cpu-bound computation*).

Debido a que con operaciones matriz-matriz podemos obtener una mayor intensidad aritmética y así utilizar en mayor medida los recursos de procesamiento, buscamos acelerar la convergencia del algoritmo de gradiente conjugado, el cual está limitado por el cuello de botella de la multiplicación de matriz dispersa por vector. Además, un algoritmo de Krylov por bloques como el propuesto puede generar un subespacio de búsqueda mayor y exhibe

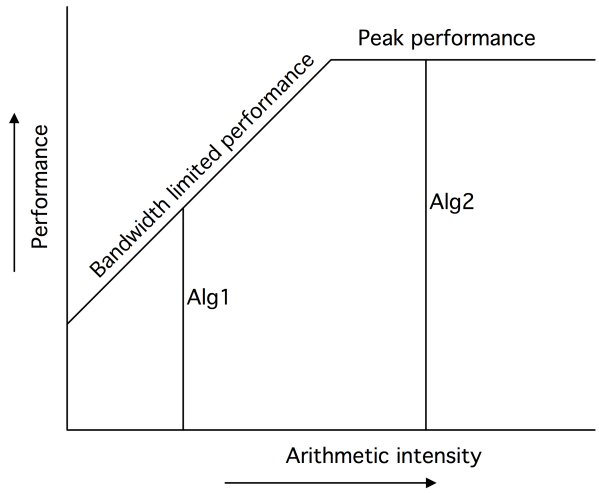


Figura 2.2: Gráfica del modelo *Roofline* para dos algoritmos, Alg1 y Alg2.
Fuente: Introduction to High Performance Scientific Computing. [16, p. 42].

mejores propiedades para una implementación paralela.

Para tener una perspectiva y mayores detalles sobre la diferencia en rendimiento de las rutinas sobre matrices dispersas mencionadas se puede consultar el trabajo de Liu y colaboradores [23], en donde se reporta que una matriz dispersa se puede multiplicar por otra matriz densa con un tamaño de bloque que varía entre 8 a 16 en el mismo tiempo que multiplicar la misma matriz dispersa por un vector 2 veces.

2.2. Revisión de la Literatura y Estado del Arte

Optimización de SpMV

En los trabajos [19], [33] y [3] se explora el comportamiento del kernel SpMV, que consiste en una multiplicación de la forma $y = A * x$, donde A es una matriz dispersa, x es el vector fuente e y es el vector de destino, el cual almacena el resultado de la multiplicación.

Los trabajos citados desarrollan estrategias para la optimización de esta operación, teniendo en cuenta su ubicuidad y la cantidad de métodos numéricos que lo emplean. Específicamente, se propone la utilización de bloques a nivel de registro (para minimizar lecturas/escrituras) y caché (para minimizar *cache misses*) [14] así como la multiplicación de la matriz dispersa por múltiples vectores (SpMM), ya que de esta forma la operación se amortiza por cada vector adicional, pero está limitado por la capacidad de los niveles más altos de la memoria del sistema.

En [1] se discuten transformaciones de datos y optimizaciones a nivel de compilador. Teniendo como indicadores la cantidad de datos transportados y la cantidad de instrucciones ejecutadas en ambas rutinas. Esto se sumariza en el Cuadro 2.1.

	SpMV	ejecutar SpMV k veces	SpMM
Flops	$2 * nnz$	$2k * nnz$	$2k * nnz$
Palabras movidas	$nnz + 2n$	$k * nnz + 2k * n$	$nnz + 2k * n$

Cuadro 2.1: Intensidad Aritmética de SpMV y SpMM. nnz es la cantidad de elementos no ceros de la matriz dispersa y k es el tamaño de bloque para SpMM.

Otro factor a considerar es el patrón de dispersión de la matriz, el cual influye significativamente en el rendimiento de los kernels. Esto se debe a que con matrices de patrones regulares como las matrices banda, los accesos a memoria también son regulares y de esta manera se toman ventaja de los conceptos de localidad espacial y temporal. Contrariamente, con matrices cuyas estructuras están próximas a ser aleatorias se realizan lecturas y escrituras irregulares a memoria [20].

Una optimización a niveles de paralelismo para SpMM se trata en [2]. También se resalta la dependencia de este kernel en el rendimiento del transporte de datos y se presentan *speedups* considerables para el caso con mútiples procesadores y nodos con memorias distribuidas. Y que se concluye indicando que el rendimiento de la comunicación entre nodos es el factor determinante para el rendimiento general en grandes concurrencias.

Algoritmos *communication-avoiding* o de evasión de comunicación

El hecho que la comunicación resulte más costosa que las operaciones de punto flotante ha provocado la reformulación de numerosos métodos que estaban diseñados para obtener los resultados en el menor número de pasos posibles, a que realicen la menor cantidad de transportes de datos posible. Esto es conocido como *communication-avoiding*. Actualmente, la complejidad de comunicación es un mejor criterio para determinar el costo de un algoritmo en términos de energía y tiempo que estimar el número de instrucciones de procesador a ejecutar [7, 3, 13, 5, 4].

Por ejemplo, en [26] se busca minimizar la comunicación en GMRES, el cual es otro *solver* basado en el subespacio de Krylov. Y en [31] se propone una implementación paralela para la descomposición de autovectores y autovalores de una matriz simétrica que ejecuta asintóticamente menos comunicación en comparación a otros métodos propuestos para este problema.

Así mismo, en [11] se presentó el método de Gradiente Conjugado de s pasos (*s-step* CG). Impulsados por las propiedades paralelas y la localidad de datos de las cuales se puede obtener ventaja, de manera a alcanzar rendimientos superiores.

En el trabajo de Carson [7] se utilizan los términos *communication-avoiding* y *s-step* de manera intercambiable.

Influencia de los autovalores para el método CG

La tasa de convergencia para el algoritmo CG se ve notablemente influenciada por la distribución y la magnitud de los autovalores de la matriz que se busca resolver. Esta influencia se estudia en el trabajo de Jennings [21], donde se utilizan los polinomios generados por el propio CG, los polinomios de Chevishev y otros polinomios auxiliares de manera a conocer el desempeño del algoritmo y estimar el número de pasos y la tasa de convergencia para una matriz dada.

En el siguiente capítulo discutiremos los conceptos presentados en este capítulo en el contexto del CG.

Capítulo 3

Algoritmos

3.1. Gradiente Conjugado

El algoritmo de Gradiente Conjugado (CG) es un método desarrollado por Hestenes y Stiefel [18] empleado en la resolución de sistemas de ecuaciones lineales

$$Ax = b, \tag{3.1}$$

en donde A es una matriz simétrica definida positiva (SPD) de $n \times n$, y donde x es el vector solución y b es el lado derecho, ambos vectores de dimensión n.

Este algoritmo genera un subespacio de búsqueda para la solución llamado *subespacio de Krylov*, el cual es generado multiplicando la matriz A sucesivamente por un vector v, de la siguiente forma

$$\mathcal{K}_i(A, v) = span\{v, Av, A^2v, ..., A^{i-1}v\}. \tag{3.2}$$

Este método puede ser considerado como un método directo debido a que con aritmética exacta se garantiza la convergencia en un máximo de n pasos (coincidente con la dimensión del vector x). Sin embargo, típicamente se lo implementa de forma iterativa. Esto se debe a que generalmente con matrices de gran porte se logra la convergencia en mucho menos de n iteraciones.

En particular, este algoritmo busca el punto mínimo de la forma cuadrática

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c, \tag{3.3}$$

que se alcanza cuando $Ax = b$.

El proceso iterativo comienza con una estimación x_0 para la solución, que puede ser el vector cero. Basados en este parámetro determinamos el residuo inicial $r_0 = b - Ax_0$ y en general

$$r_i = b - Ax_i. \tag{3.4}$$

El error e_i es la diferencia entre la estimación para la solución en una iteración dada y la solución real x, es decir

$$e_i = x_i - x. \tag{3.5}$$

Expandiendo la ecuación (3.4) con (3.5) tenemos que

$$r_i = Ax - Ax_i = -Ae_i. \tag{3.6}$$

En cada iteración, la solución x_i se va actualizando avanzando una magnitud α_i hacia una dirección de búsqueda p_i tal que

$$x_{i+1} = x_i + \alpha_i p_i \tag{3.7}$$

y mediante la ecuación (3.5) obtenemos $x_{i+1} - x = x_i - x + \alpha p_i$ y

$$e_{i+1} = e_i + \alpha_i p_i. \tag{3.8}$$

Ahora se debe determinar el valor de α_i y explicar cómo se construyen las direcciones de búsqueda p_i en cada iteración de manera a mostrar la efectividad de este procedimiento.

La estrategia consiste en seleccionar direcciones de búsqueda *conjugadas* o *A-ortogonales*, esto significa que $p_i^T A p_j = 0$ para $i \neq j$. Una forma de interpretar esta expresión es que p_i es ortogonal a p_j en el producto interno determinado por A. Lo que se logra con las direcciones de búsqueda A-ortogonales es no volver a buscar la solución en una dirección por la cual ya se ha buscado en iteraciones anteriores.

Determinación de $\boldsymbol{\alpha}$. Como el error es la diferencia entre una solución estimada y la solución real, el error es una combinación lineal de direcciones de búsqueda A-ortogonales aún no generadas, por tanto

$$p_i^T A e_{i+1} = 0, \tag{3.9}$$

así deducimos que

$$\begin{aligned} p_i^T A e_{i+1} &= p_i^T A(e_i + \alpha_i p_i) \quad \text{(por la ecuación (3.8))} \\ 0 &= p_i^T A e_i + \alpha_i p_i^T A p_i \\ \alpha_i &= -\frac{p_i^T A e_i}{p_i^T A p_i}. \end{aligned}$$

Ya que no tenemos el término de error (si se conoce el término e_i ya tendríamos la solución del sistema) pero sí el residuo, por la ecuación (3.6), reemplazamos el factor Ae_i por su equivalente $-r_i$ en la expresión para hallar α_i, entonces

$$\alpha_i = \frac{p_i^T r_i}{p_i^T A p_i}. \tag{3.10}$$

Determinación de $\boldsymbol{\beta}$. En cuanto a las direcciones de búsqueda, estas son construidas a partir de vectores linealmente independientes $\{u_0, u_1, ..., u_{n-1}\}$ mediante el *proceso Gram-Schmidt conjugado*. Este proceso genera una dirección de búsqueda p_i de tal forma a que sea A-ortogonal a las todas las direcciónes de búsqueda anteriores. Se comienza con $p_0 = u_0$ y para $i > 0$

$$p_i = u_i + \sum_{k=0}^{i-1} \beta_{ik} p_k. \tag{3.11}$$

Con el fin de determinar el valor de β para la construcción de una dirección de búsqueda dada debemos tener en cuenta la A-ortogonalidad de estas direcciones. Si multiplicamos cada lado de la ecuación (3.11) por $p_j^T A$ tenemos que

$$\begin{aligned} p_j^T A p_i &= p_j^T A u_i + \sum_{k=0}^{i-1} \beta_{ik} p_j^T A p_k \\ 0 &= p_j^T A u_i + \beta_{ij} p_j^T A p_j, \qquad j < i \\ \beta_{ij} &= -\frac{p_j^T A u_i}{p_j^T A p_j}, \end{aligned}$$

por simetría de A

$$\beta_{ij} = -\frac{u_i^T A p_j}{p_j^T A p_j}. \tag{3.12}$$

El problema con este proceso es que se necesitan almacenar todas las direcciones de búsqueda anteriores para obtener una nueva p_i, el cual tiene una complejidad espacial de $\mathcal{O}(n^2)$. Esto es, la complejidad relacionada al requerimiento de almacenamiento de la memoria del sistema. Además, la cantidad de operaciones para completar el proceso de Gram-Schmidt conjugado es de $\mathcal{O}(n^3)$ (en general, la cantidad de operaciones efectuadas con el proceso clásico de Gram-Schmidt para una matriz A de m por n es de $\mathcal{O}(mn^2)$).

La solución a este problema es utilizar los residuos en vez de las vectores u_i como base para construir las direcciones p_i. Como el error se compone de direcciones de búsqueda A-ortogonales que aún no se han generado (ecuación (3.9)) tenemos que $-p_i^T A e_j = 0$ para $j > i$ y por tanto

$$p_i^T r_j = 0 \quad \text{para } j > i. \tag{3.13}$$

Tomando el producto interno del residuo para una iteración j y la ecuación (3.11)

$$p_i^T r_j = u_i^T r_j + \sum_{k=0}^{i-1} \beta_{ik} p_k^T r_j \tag{3.14}$$

$$0 = u_i^T r_j \quad \text{(por la ecuación (3.13))}, \tag{3.15}$$

y como se utilizan los residuos en vez de u_i,

$$r_i^T r_j = 0. \tag{3.16}$$

Por (3.14) y (3.16)

$$\begin{aligned} p_i^T r_j &= u_i^T r_j \\ u_i^T r_j &= r_i^T r_j. \end{aligned} \tag{3.17}$$

Así se garantiza que el residuo actual es ortogonal a todas las direcciones p_i anteriores, solamente necesitamos almacenar la dirección actual para construir la siguiente en cada

iteración. Por lo indicado anteriormente la ecuación (3.12) se convierte en

$$\beta_{ij} = -\frac{r_i^T A p_j}{p_j^T A p_j}. \tag{3.18}$$

Adicionalmente, para demostrar que no se requiere almacenar todas las direcciones de búsqueda ya construidas se puede utilizar la expresión para obtener el residuo utilizando la ecuación (3.8)

$$\begin{aligned} r_{i+1} &= -Ae_{i+1} \\ &= -A(e_i + \alpha_i p_i) \\ &= -Ae_i - \alpha_i A p_i \\ &= r_i - \alpha_i A p_i. \end{aligned} \tag{3.19}$$

Evaluando el producto interno de r_i por (3.19) obtenemos

$$r_i^T r_{j+1} = r_i^T r_j - \alpha_j r_i^T A p_j,$$

despejando el numerador de la expresión para β_{ij},

$$r_i^T A p_j = \frac{1}{\alpha_j}(r_i^T r_j - r_i^T r_{j+1}), \tag{3.20}$$

entonces

$$r_i^T A p_j = \begin{cases} \frac{1}{\alpha_i} r_i^T r_i, & i = j \\ -\frac{1}{\alpha_{i-1}} r_i^T r_i, & i = j+1 \\ 0, & \text{para otro caso.} \end{cases} \tag{3.21}$$

Como la ecuación (3.12) indica que β_{ij} está definida para $i > j$, evaluamos el segundo caso, en donde $i = j + 1$,

$$\begin{aligned} \beta_{ij} = \beta_{i,i-1} &= -\frac{-r_i^T r_i}{\frac{p_{i-1}^T r_{i-1}}{p_{i-1} A p_{i-1}} p_{i-1}^T A p_{i-1}} \\ &= \frac{r_i^T r_i}{p_{i-1}^T r_{i-1}}. \end{aligned} \tag{3.22}$$

Para simplificar, podemos llamar β_i a $\beta_{i,i-1}$.

La ecuación (3.22) nos dice que para constuir una nueva dirección de búsqueda solamente se requiere almacenar el residuo y la dirección anteriores a la iteración actual. La complejidad de de tiempo y espacio en cada iteración se reduce de $\mathcal{O}(n^2)$ a $\mathcal{O}(m)$, donde m es la cantidad de elementos no ceros de la matriz A [30].

Debido a que en cada iteración debemos computar el cuadrado de la norma del residuo r_i para el numerador en β_i podemos reemplazar los términos en α_i y en β_i para utilizar este producto interno en lugar de $p_i^T r_i$ de manera a optimizar el proceso iterativo. Entonces, por las ecuaciones en (3.17) obtenemos

$$\alpha_i = \frac{r_i^T r_i}{p_i^T A p_i}, \tag{3.23}$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}. \tag{3.24}$$

Agrupando todos los pasos se define el método de Gradiente Conjugado, el cual se muestra en el Algoritmo 1.

Algorithm 1 Gradiente Conjugado

$p_0 = r_0 = b - Ax_0$
for $i = 0, 1, ...,$ *hasta la convergencia* **do**
    $\alpha_i = (r_i, r_i)/(p_i, Ap_i)$
    $x_{i+1} = x_i + \alpha_i p_i$
    $r_{i+1} = r_i - \alpha_i A p_i$
    $\beta_i = (r_{i+1}, r_{i+1})/(r_i, r_i)$
    $p_{i+1} = r_{i+1} + \beta_i p_i$
end for

Una implementación iterativa más eficiente que el Algoritmo 1 para lenguaje de programación se propone en [22], el cual se muestra en el Algoritmo 2.

Algorithm 2 Gradiente Conjugado Clásico - Implementación

procedure CG(A, x, b, ϵ, $kmax$)
    $r = b - Ax, \rho_0 = \|r\|_2^2, k = 1.$
    while $\sqrt{\rho_{k-1}} > \epsilon\|b\|_2$ and $k < kmax$ **do**
        if $k = 1$ **then**
            $p = r$
        else
            $\beta = \rho_{k-1}/\rho_{k-2}$
            $p = r + \beta p$
        end if
        $w = Ap$
        $\alpha = \rho_{k-1}/p^T w$
        $x = x + \alpha p$
        $r = r - \alpha w$
        $\rho_k = \|r\|_2^2$
        $k = k + 1$
    end while
end procedure

Para declarar convergencia se busca un vector x de forma a reducir la norma-2 del residuo en un factor ϵ (típicamente en el orden de 10^{-6}) con respecto a la norma-2 del lado derecho b. Es decir, para determinar la llegada a la convergencia se debe cumplir $\|b - Ax\|_2 \leq \epsilon \|b\|_2$. También se establece un parámetro $kmax$ que indica la cantidad máxima de iteraciones a ejecutar. Estos son los criterios de parada del Algoritmo.

3.2. Gradiente Conjugado por Bloques

Este método se desarrolló en los trabajos de O'Leary [27] y Broyden [6] entre otros, y en los cuales se pueden encontrar mayores detalles de este método. Está orientado a la resolución de sistemas lineales con múltiples lados derechos. Es decir, resolver

$$AX = B, \tag{3.25}$$

donde A es una matriz SPD, X y B son matrices de n por m. Típicamente, con $m \ll n$. El valor de m también se conoce como tamaño de bloque.

Las soluciones aproximadas para B se obtienen mediante

$$X_{i+1} = X_i + P_i \Lambda_i, \tag{3.26}$$

en donde Λ_i es la matriz de magnitudes para las direcciones de búsqueda P_i. Las magnitudes se determinan teniendo en cuenta la condición de que la siguiente matriz de residuos debe ser ortogonal al subespacio generado por P_i, de la misma forma que en la ecuación (3.13) de la versión clásica, entonces

$$\begin{aligned} P_i^T R_{i+1} &= 0 \\ P_i^T (R_i - A P_i \Lambda_i) &= 0, \end{aligned} \tag{3.27}$$

despejando Λ_i se obtiene

$$\Lambda_i = (P_i^T A P_i)^{-1} P_i^T R_i. \tag{3.28}$$

A su vez, P_i es un subespacio de búsqueda de m dimensiones que se construye de manera similar a su contraparte clásica, tal que

$$P_{i+1} = R_{i+1} + P_i \Psi_i. \tag{3.29}$$

La matriz Ψ_i se determina haciendo que el subespacio P_{i+1} sea A-ortogonal al subespacio P_i generado en la iteración anterior, de esta forma

$$P_{i+1}^T A P_i = 0, \tag{3.30}$$

entonces

$$\Psi_i = -(P_i^T A P_i)^{-1} P_i^T A R_{i+1}. \tag{3.31}$$

Teniendo las mismas condiciones para las direcciones de búsqueda y los residuos que en la versión clásica, se deducen las siguientes igualdades

$$\begin{aligned} P_i^T A P_j &= 0 \quad \text{para } i \neq j \\ R_i^T R_j &= 0 \quad \text{para } i \neq j \\ P_i^T R_j &= 0 \quad \text{para } i < j, \end{aligned} \tag{3.32}$$

y con estas ecuaciones simplificamos las expresiones finales para Λ_i y Ψ_i,

$$\begin{aligned} \Lambda_i &= (P_i^T A P_i)^{-1} R_i^T R_i \\ \Psi_i &= (R_i^T R_i)^{-1} R_{i+1}^T R_{i+1}. \end{aligned} \tag{3.33}$$

La iteración se realiza de la misma forma que en la versión clásica. En este caso se declara convergencia si $\|r_j\|_2 \leq \epsilon \|b_j\|_2$ para todo $j \in [1, m]$, tal que r_j y b_j son columnas de R y B respectivamente. Esto se muestra en el Algoritmo 3, desarrollado en [17].

Ya que el subespacio de búsqueda es generado a partir de un conjunto de vectores linealmente independientes en cada iteración, es importante remarcar que con aritmética exacta la máxima cantidad de iteraciones necesarias para encontrar la solución es de $[(n+m-1)/m]$.

Algorithm 3 Gradiente Conjugado por Bloques para múltiples lados derechos

1: $P_0 = P_0 = B - AX_0$
2: **for** $i = 0, 1, 2, ...$ **do**
3: $\quad \Lambda_i = (P_i^T A P_i)^{-1} R_i^T R_i$
4: $\quad X_{i+1} = X_i + P_i \Lambda_i$
5: $\quad R_{i+1} = R_i - A P_i \Lambda_i$
6: $\quad$ **if** $\|r_{i+1}^j\| < \epsilon \|b_j\| \ \forall j \in [1, m]$ **then**
7: $\quad\quad$ **break**
8: $\quad$ **end if**
9: $\quad \Psi_i = (R_i^T R_i)^{-1} R_{i+1} r_{i+1}$
10: $\quad P_{i+1} = R_{i+1} + P_i \Psi_i$
11: **end for**

Solución $AX = B$ con $b_i = b \ \forall i = 1 \ldots m$. Como en este trabajo se busca acelerar la convergencia para un solo lado derecho, a continuación se describen los pasos para obtener el promedio de soluciones para un método iterativo de Krylov por bloques que se demuestran en el artículo de Chronopoulos y Kucherov [10]. En el trabajo citado se indica que la combinación óptima de las soluciones columnas de X para un solo lado derecho está dada por la expresión

$$\hat{x} = \frac{1}{e^T \xi} X \xi, \tag{3.34}$$

tal que

$$\xi = (R^T R)^{-1} e, \tag{3.35}$$

en donde $e = (1_1, \ldots, 1_m)$.

Teniendo un solo lado derecho b y la matriz de soluciones X_i podemos obtener una matriz de residuos para la iteración i de la siguiente forma

$$R = be^T - AX. \tag{3.36}$$

Multiplicando la ecuación (3.36) por un vector ξ de dimensión m (a determinar) y escalando ambos lados de la ecuación por el producto interno $e^T\xi$ obtenemos

$$\frac{1}{e^T\xi}R\xi = b - A\left(\frac{1}{e^T\xi}X\xi\right). \tag{3.37}$$

De esta manera el residuo obtenido de la solución promedio $\hat{x}$ es igual a

$$\begin{aligned}\hat{r} &= b - A\hat{x} \\ &= \frac{1}{e^T\xi}R\xi.\end{aligned} \tag{3.38}$$

Algorithm 4 Gradiente Conjugado por Bloques para un lado derecho mediante combinación de soluciones

$P_0 = P_0 = B - AX_0$
for $i = 0, 1, \ldots$ **do**
  $\Lambda_i = (P_i^T AP_i)^{-1} R_i^T R_i$
  $X_{i+1} = X_i + P_i\Lambda_i$
  $R_{i+1} = R_i - AP_i\Lambda_i$
  ▷ control de convergencia
  $R_{i+1} = Q_R W$ ▷ descomposición QR
  $W^T\eta = e$ ▷ resolver por sustitución regresiva para η
  $W\xi = \eta$ ▷ resolver por sustitución regresiva para ξ
  $\hat{x}_{i+1} = \frac{1}{e^T\xi}X_{i+1}\xi$ ▷ solución promedio
  $\hat{r}_{i+1} = \frac{1}{e^T\xi}R_{i+1}\xi$ ▷ residuo promedio
  if $\|\hat{r}_{i+1}\| < \epsilon\|b\|$ **then**
    break
  end if
  ▷ fin del control de convergencia
  $\Psi_i = (R_i^T R_i)^{-1} R_{i+1}R_{i+1}$
  $P_{i+1} = R_{i+1} + P_i\Psi_i$
end for

Calculando la norma al cuadrado del residuo promedio tenemos que

$$\left\|\frac{1}{e^T\xi}R\xi\right\|^2 = \frac{1}{(e^T\xi)^2}\xi^T R^T R\xi, \tag{3.39}$$

y por la desigualdad de Cauchy-Schwarz

$$(e^T\xi)^2 \leq (\xi^T(R^TR)\xi)(e^T(R^TR)^{-1}e) \tag{3.40}$$

se deduce que

$$\left\|\frac{1}{e^T\xi}R\xi\right\|^2 \geq \frac{\xi^TR^TR\xi}{(\xi^TR^TR\xi)(e^T(R^TR)^{-1}e)}, \tag{3.41}$$

así se determina que el valor mínimo para el cuadrado de la norma del residuo promedio es

$$\frac{1}{e^T(R^TR)^{-1}e},$$

que solo se puede obtener si se cumple la condición indicada en la ecuación (3.35). De esta forma ξ se elige de manera a que

$$\xi = \text{argmin}_\xi \, \|\hat{r}\|_2^2. \tag{3.42}$$

Si tomamos la ecuación (3.35), para hallar ξ se debe resolver

$$\xi = (R^TR)^{-1}e \tag{3.43}$$

Es importante que la matriz R sea *full rank* de forma a que R^TR se pueda invertir en cada iteración. Por eso X_0, la matriz de soluciones iniciales, debe componerse de columnas linealmente independientes. Otro detalle a tener en cuenta es que debido a que el proceso iterativo puede hacer que las columnas de R se tornen cercanas a ser linealmente dependientes, es necesario factorizar la matriz R mediante la descomposición QR, entonces

$$\begin{aligned} R &= Q_RW \\ R^TR &= (Q_RW)^T(Q_RW) \\ &= W^TW, \end{aligned} \tag{3.44}$$

y como la matriz W es triangular, es necesario ejecutar solamente dos sustituciones regresivas para determinar ξ, precisamente

$$\begin{aligned} W^T\eta &= e \\ W\xi &= \eta. \end{aligned} \tag{3.45}$$

Una vez que computamos ξ, podemos obtener el residuo $\hat{r}$ que utilizaremos para determinar la llegada a la convergencia, de la misma manera que en la implementación de la versión clásica. Es decir, se itera hasta que

$$\|\hat{r}\|_2 \leq \epsilon\|b\|_2.$$

Debido a que el control de convergencia es más costoso que en la versión clásica, ya que debemos computar una descomposición QR y resolver dos sistemas triangulares, se añade un parámetro de paso, que indica en cada cuántas iteraciones se realiza el control de convergencia.

La implementación del BCG propuesto se estructura de manera similar al Algoritmo 2, la cual se muestra en el Algoritmo 5.

Algorithm 5 Gradiente Conjugado por bloques para un solo lado derecho con combinación de soluciones - Implementación

procedure BCG(A, X, b, ϵ, $kmax$, $step$)
  $e = [1_1 \dots 1_m]^T, R = be^T - AX, \rho_0 = R^T R, k = 1.$
  while $k < kmax$ **do**
    if $k \bmod step = 0$ **then**
      $R = Q_R W$ ▷ descomposición QR
      $W^T \eta = e$ ▷ resolver por sustitución regresiva para η
      $W\xi = \eta$ ▷ resolver por sustitución regresiva para ξ
      $r = \frac{1}{e^T \xi} R\xi$ ▷ residuo promedio
      if $\|r\|_2 \leq \epsilon \|b\|_2$ **then**
        $x = \frac{1}{e^T \xi} X\xi$ ▷ solución promedio
        break
      end if
    end if
    if $k = 1$ **then**
      $P = R$
    else
      $\Psi = \rho_{k-2}^{-1} \rho_{k-1}$
      $P = R + P\Psi$
    end if
    $Q = AP$
    $\Lambda = (P^T Q)^{-1} \rho_{k-1}$
    $X = X + P\Lambda$
    $R = R - Q\Lambda$
    $\rho_k = R^T R$
    $k = k + 1$
  end while
  if $k = kmax$ **then** ▷ no se llegó a la convergencia, obtener última solución
    $R = Q_R W$
    $W^T \eta = e$
    $W\xi = \eta$
    $x = \frac{1}{e^T \xi} X\xi$
    $r = \frac{1}{e^T \xi} R\xi$
  end if
end procedure

Capítulo 4

Experimentos numéricos

De manera a conocer el desempeño de los algoritmos estudiados y contrastar el algoritmo BCG propuesto con su contraparte clásica, ejecutamos un conjunto de pruebas sobre matrices con distintos dominios de aplicación. En la Sección 4.2 se detallan las matrices, las plataformas sobre las cuales ejecutamos las pruebas y los parámetros establecidos para cada rutina. En la Sección 4.3 se muestran figuras que muestran la tasa de convergencia para CG y BCG sobre las matrices seleccionadas. Posteriormente, se muestran los tiempos de ejecución.

4.1. Construcción de una matriz SPD

Una de las matrices utilizadas en las pruebas es la matriz poisson10k la cual surge a partir de la discretización de un problema expresado como una ecuación diferencial parcial. Precisamente, la ecuación de Poisson se expresa como

$$-a\nabla^2 u = f \quad \text{en } \Omega, \tag{4.1}$$

en donde a es una constante positiva, el Laplaciano de u se da por

$$\nabla^2 u = \sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i}, \tag{4.2}$$

f es la función del término fuente y Ω es un subconjunto abierto de $\mathbb{R}^d$, $d = 1, 2$ o 3. Se busca determinar la función desconocida $u : \Omega \to \mathbb{R}$. Esta ecuación describe numerosos fenómenos físicos de campo, como la conducción de calor, el flujo de pontencia, la diferencia de potencial, entre otros [29, 25].

Para obtener la solución se establecen condiciones sobre la frontera de Ω, que es $\partial\Omega$. Por simplicidad podemos establecer que

$$u = 0 \quad \text{en } \partial\Omega. \tag{4.3}$$

Particularmente, en este problema se estudia una superficie cuadrada ($d = 2$) determinada por

$$\Omega = (0, L) \times (0, L) \tag{4.4}$$

y la cual es dividida en ambas dimensiones por un entero positivo N. Entonces, el tamaño de paso es $h = \frac{L}{N}$. De esta forma se genera una malla de n por n, donde $n = N - 1$.

Como buscamos discretizar este problema, debemos encontrar una aproximación para el valor de $u(x_{ij})$ para cada punto x_{ij} en la superficie. Los puntos están dados por

$$x_{ij} = (ih, jh) \qquad \text{para } 0 \leq i, j \leq N.$$

La ecuación diferencial en (4.2)

$$\nabla^2 u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} \tag{4.5}$$

es aproximada mediante las diferencias

$$\frac{\partial^2 u}{\partial x_1^2} \approx \frac{\Delta^2 u}{\Delta x_1^2} = \frac{u(x_1 - h, x_2) - 2u(x_1, x_2) + u(x_1 + h, x_2)}{h^2}, \tag{4.6}$$

$$\frac{\partial^2 u}{\partial x_2^2} \approx \frac{\Delta^2 u}{\Delta x_2^2} = \frac{u(x_1, x_2 - h) - 2u(x_1, x_2) + u(x_1, x_2 + h)}{h^2}, \tag{4.7}$$

y así se obtiene el Laplaciano discreto

$$\nabla^2 u_{ij} = \frac{u_{i-1,j} + u_{i,j-1} - 4u_{ij} + u_{i+1,j} + u_{i,j+1}}{h^2}, \tag{4.8}$$

en donde $u_{ij} \approx u(x_{ij})$. Esto es, u_{ij} se aproxima al valor verdadero de $u(x_{ij})$ a medida que $h \to 0$.

Utilizando el Laplaciano discreto en la ecuación (4.1) obtenemos una ecuación de diferencia,

$$-u_{i-1,j} - u_{i,j-1} + 4u_{ij} - u_{i+1,j} - u_{i,j+1} = \frac{h^2}{a} f_{ij} \qquad \text{para } 1 \leq i, j \leq n, \tag{4.9}$$

formando de esta manera un sistema de n^2 ecuaciones con n^2 incógnitas,

$$Au = \frac{h^2}{a} f. \tag{4.10}$$

Por ejemplo, si $n = 3$, entonces $i, j \in \{1, 2, 3\}$ y obtendríamos

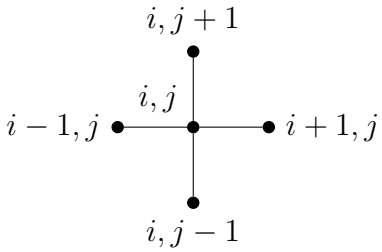
$$\left[\begin{array}{ccc|ccc|ccc}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
 & -1 & 4 & & & -1 & & & \\
\hline
-1 & & & 4 & -1 & & -1 & & \\
 & -1 & & -1 & 4 & -1 & & -1 & \\
 & & -1 & & -1 & 4 & & & -1 \\
\hline
 & & & -1 & & & 4 & -1 & \\
 & & & & -1 & & -1 & 4 & -1 \\
 & & & & & -1 & & -1 & 4
\end{array}\right]
\left[\begin{array}{c}
u_{11} \\ u_{21} \\ u_{31} \\ \hline u_{12} \\ u_{22} \\ u_{32} \\ \hline u_{13} \\ u_{23} \\ u_{33}
\end{array}\right]
= \frac{h^2}{a}
\left[\begin{array}{c}
f_{11} \\ f_{21} \\ f_{31} \\ \hline f_{12} \\ f_{22} \\ f_{32} \\ \hline f_{13} \\ f_{23} \\ f_{33}
\end{array}\right].$$

En cuanto a los puntos de frontera, tenemos que

$$u(i,0), u(0,j), u(i,n+1), u(n+1,j) = 0$$

por la condición establecida en la ecuación (4.3). Si no fueran cero, serían pasadas a lado derecho de la igualdad [32].

Este sistema también se conoce como Laplaciano discreto de cinco puntos debido a que cada $\nabla^2 u_{ij}$ para $1 \leq i, j \leq n$ opera sobre un *stencil* consistente en cinco puntos, distribuidos de la siguiente forma



4.2. Configuración experimental

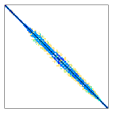
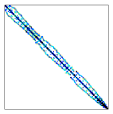
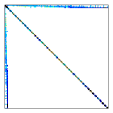
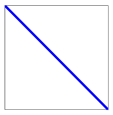
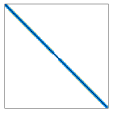
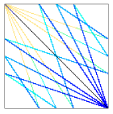
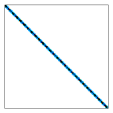
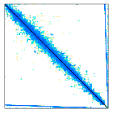
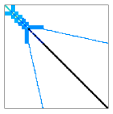
Todas las matrices utilizadas son de *The Suite Sparse Matrix Collection*, conocida anteriormente como la Colección de Matrices Dispersas de la Universidad de Florida [12]. Exceptuando la matriz `poisson10k`, la cual se generó ejecutando la función `gallery('poisson',100)` en MATLAB, la cual retorna una matriz Laplaciana 2D resultado de discretizar la ecuación de Poisson, los detalles se muestran en la Sección 4.1. En el Cuadro 4.1 se muestran los nombres de las matrices, también sus aplicaciones y estructura, dimensión y cantidad de elementos no ceros.

En cuanto a la ejecución de las funciones, para ambos algoritmos se generaron estimaciones iniciales para las soluciones de forma aleatoria con la función `randn` de MATLAB, que retorna un conjunto de números de punto flotante aleatorios con distribución normal. Para cada matriz, en la llamada a BCG se reutilizó el vector aleatorio inicial x_0 pasado como parámetro a CG más $m-1$ vectores aleatorios, todos linealmente independientes, para construir X_0. En este conjunto de pruebas $m \in \{2,3,4,5\}$ y $step = 1$ para computar el residuo promedio en cada iteración en BCG.

El valor del lado derecho b lo establecimos de manera a que sea igual a la matriz A multiplicada por un vector con todas las entradas iguales a $1/\sqrt{n}$.

Para las pruebas de tiempo con BCG variamos los parámetros m y $step$ de manera a encontrar una combinación de estos que puedan lograr tiempos de ejecución inferiores. Los valores de estos parámetros se variaron tal que $m \in \{2,3,\ldots,8\}$ y $step \in \{5,6,\ldots,20\}$.

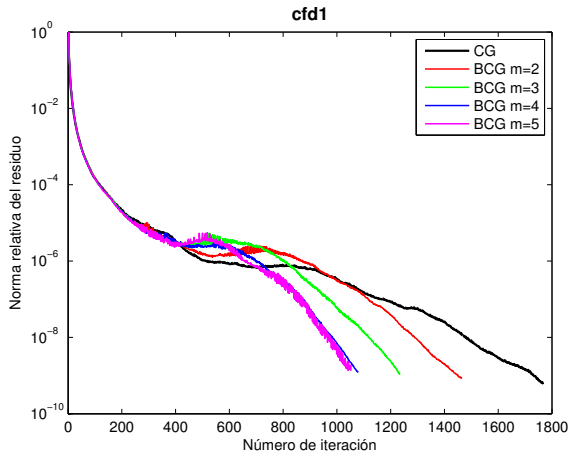
Todos los experimentos los ejecutamos en MATLAB sobre un procesador Intel Core i7-6700T @ 2.8 GHz con 8 GB de RAM.

Nombre Aplicación/Descripción	Spy plot	n	nnz
cfd1 Dinámica de fluidos computacional		70656	1825580
ex3 Dinámica de fluidos computacional		1821	52685
bundle1 Gráficas / visión computacional		10581	390741
poisson10k Laplaciano 2D		10000	49600
fv2 Problema 2D/3D		9801	87025
qa8fm Problema acústico		66127	1660579
cvxbqp1 Problema de optimización		50000	349968
apache1 Problema estructural		80800	311492
thermal1 Problema térmico		82654	574458
G2_circuit Simulación de circuito		150102	726674

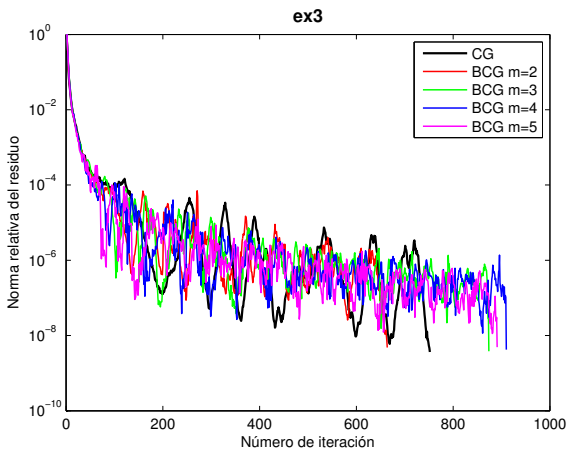
Cuadro 4.1: Nombre e información estructural de cada una de las matrices utilizadas en las pruebas.

4.3. Resultados

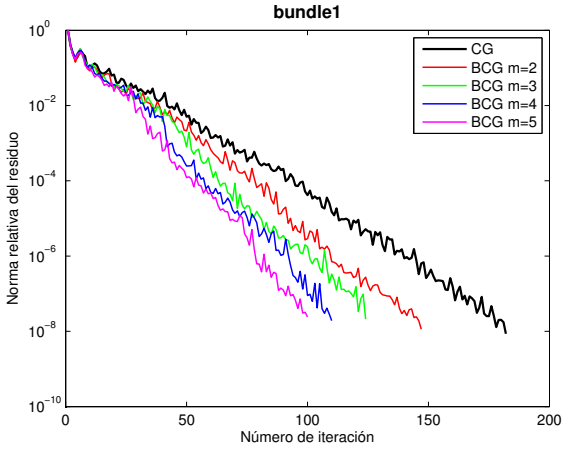
Las Figuras 4.1, 4.2, 4.3 y 4.4 muestran los gráficos de las tasas de convergencia para las matrices especificadas en el Cuadro 4.1 en base a la variación de la norma-2 del residuo en cada iteración con respecto a la norma-2 del residuo inicial.



(a) cfd1.



(b) ex3.

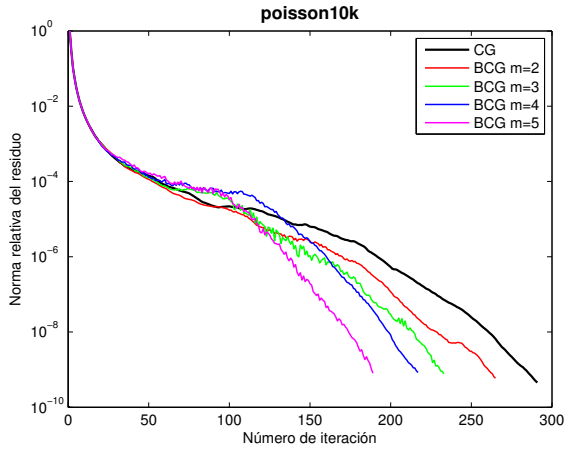


(c) bundle1.

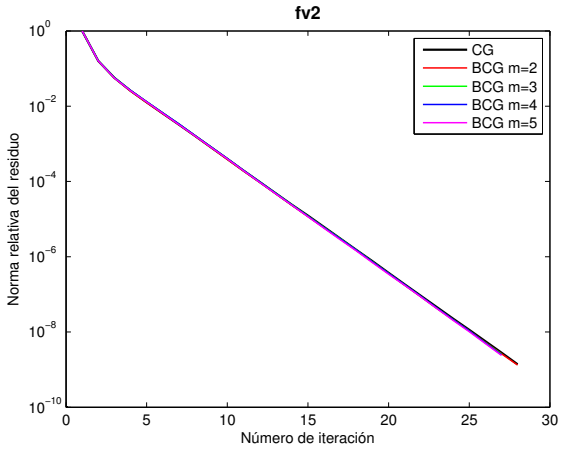
Figura 4.1: Norma relativa del residuo en función al número de iteración para cfd1, ex3 y bundle1.

Para 4.1(a) cfd1 hay un decremento un tanto uniforme en la cantidad de iteraciones necesarias para llegar a la convergencia a medida que aumenta el tamaño de bloque. Excepto para BCG con $m = 5$ que terminó en casi la misma cantidad de iteraciones que para $m = 4$. Con 4.1(b) ex3 no se observa ningún patrón con relación al tamaño de bloque y la cantidad

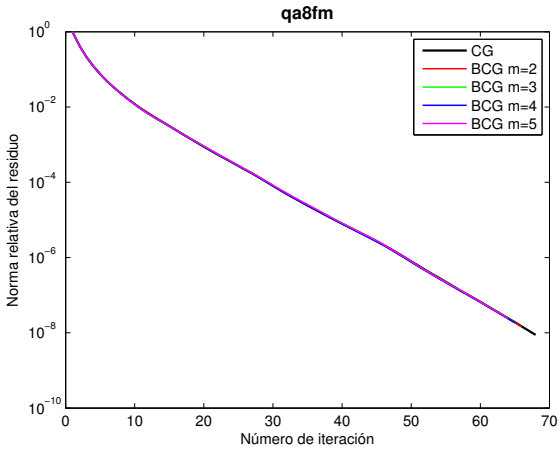
de iteraciones ejecutadas. El caso con la convergencia más acelerada es con $m = 2$, mientras que se realizaron más iteraciones para $m > 2$ en comparación a CG. La matriz 4.1(c) bundle1 muestra reducción cercana al 20 % en la cantidad de iteraciones para $m = 2$ por sobre la implementación clásica.



(a) poisson10k.



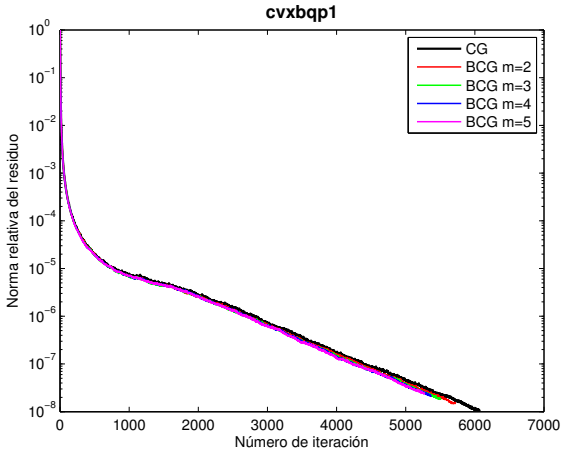
(b) fv2.



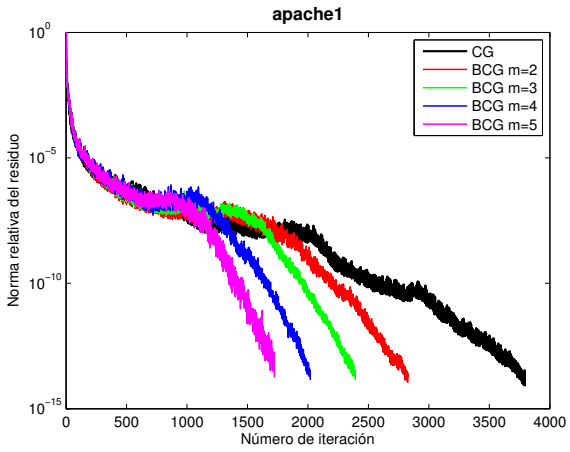
(c) qa8fm.

Figura 4.2: Norma relativa del residuo en función al número de iteración para poisson10k, fv2 y qa8fm.

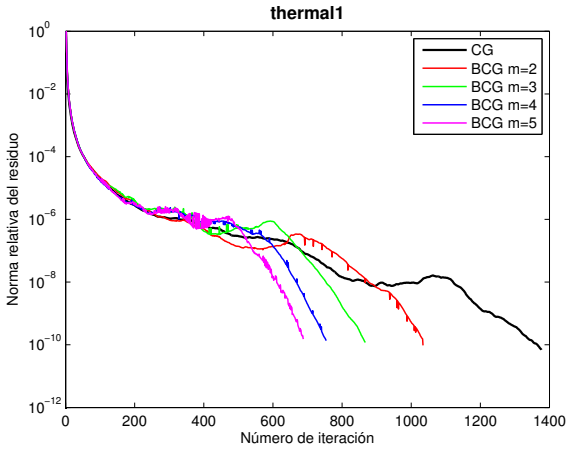
Con la matriz Laplaciana 2D, 4.2(a) poisson10k, también se obtuvieron reducciones en el número de iteraciones con respecto al tamaño de bloque. Dos casos particulares son los experimentos con 4.2(b) fv2 y 4.2(c) qa8fm, en donde tanto para CG y BCG con distintos tamaños de bloque se reduce la norma del residuo de la misma manera. Es decir, no se logra una variación en la reducción de la norma-2 del residuo al variar el valor del parámetro m.



(a) cvxbqp1.



(b) apache1.



(c) thermal1.

Figura 4.3: Norma relativa del residuo en función al número de iteración para cvxbqp1, apache1, thermal1.

La gráfica de la matriz 4.3(a) cvxbqp1 es similar a las gráficas de 4.2(b) fv2 y 4.2(c) qa8fm. La diferencia en este caso es que la cantidad de iteraciones ejecutadas varía en el orden 10^2 entre CG y BCG con $m = 5$. Para 4.3(b) apache1 hay una reducción casi uniforme en el número de iteraciones con los distintos valores para el tamaño de bloque. Aún así se observa una diferencia considerable entre BCG con $m = 2$ y CG. En la Figura 4.3(c) thermal1 se muestran variaciones similares en la reducción de la norma residual hasta aproximadamente la iteración número 200. Como en la mayoría de las pruebas, la mayor diferencia en las tasas de convergencia se observa entre BCG con tamaño de bloque igual a 2 y CG.

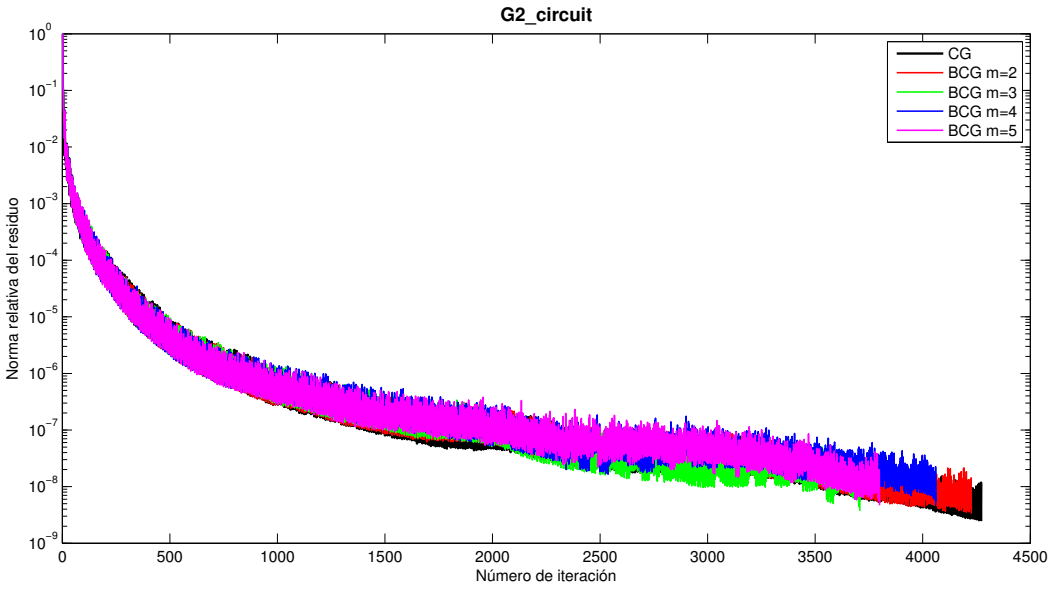


Figura 4.4: Norma relativa del residuo en función al número de iteración para G2_circuit.

Con $m = 3$ se llegó a la convergencia en menos iteraciones para G2_circuit (Figura 4.4). Para todas las pruebas sobre esta matriz, la reducción de la norma relativa del residuo se dió de forma similar. También el ritmo de convergencia se vio retardado en comparación a los resultados de las demás matrices.

Matriz	**CG**	**BCG**	$\boldsymbol{m}$	$\boldsymbol{step}$
cfd1	**20.797**	29.000	2	20
bundle1	0.76562	**0.68750**	2	11
poisson10k	0.39062	**0.25000**	2	10
fv2	~ 0	~ 0	2	10
qa8fm	**0.67188**	1.2500	2	6
cvxbqp1	**27.391**	31.321	2	19
apache1	**24.219**	37.438	2	19
thermal1	**12.938**	17.422	2	18

Cuadro 4.2: Tiempos de ejecución de CG y BCG para las matrices seleccionadas. Todos los tiempos se indican en segundos. Para el caso de BCG se agregan los parámetros m y $step$ con los cuales se lograron los tiempos más reducidos.

Los tiempos de ejecución obtenidos se sumarizan en el Cuadro 4.2. Para el caso de BCG se indicamos los tiempos transcurridos más reducidos en función a la variación en los parámetros m y $step$. Para cada matriz resaltamos la cantidad de segundos del algoritmo con menor tiempo de ejecución transcurrido.

Se puede observar que a pesar de que la cantidad de iteraciones disminuya con el tamaño de bloque, en la mayoría de los casos no se logran tiempos de ejecución inferiores. Sin embargo se lograron *speedups* considerables para las matrices bundle1 y poisson10k. Una reducción en el tiempo de ejecución de 10 % y 35 %, respectivamente. Con la matriz fv2, MATLAB registró un tiempo de 0 segundos, tanto para CG como para BCG con $m = 2$. En la Figura 4.2(b) se puede observar que se requieren menos de 30 iteraciones para llegar a la convergencia con la matriz fv2. En todos los experimentos con BCG, el tamaño de bloque que resultó en un menor tiempo de ejecución es de $m = 2$.

Capítulo 5

Conclusiones

En este trabajo estudiamos los fundamentos teóricos y realizamos experimentos sobre los algoritmos de Gradiente Conjugado clásico y su versión por bloques. Si bien la versión por bloques ya ha sido propuesta para múltiples lados derechos, en este trabajo proponemos explotar esta estrategia para el caso de un sistema de ecuaciones lineales con un único lado derecho. Teniendo en cuenta las cuestiones computacionales discutidas en el Capítulo 2 y las propiedades de ambos algoritmos, tratadas en el Capítulo 3.

Una de las características de los métodos de Krylov por bloques, es que en cada paso se logra generar un subespacio de búsqueda mayor por emplear múltiples direcciones simultáneamente. De esta manera, en principio, se puede acelerar la convergencia. A este propósito, en particular, para el caso con un único lado derecho se necesitan de direcciones de búsqueda linealmente independientes.

En cuanto a los resultados experimentales, se muestra que en general la cantidad de iteraciones necesarias para llegar a la convergencia disminuye en relación al tamaño de bloque. Y también, en la mayoría de los casos estudiados, las tasas de convergencia son similares en las primeras iteraciones. A pesar de la reducción en la cantidad de iteraciones, con BCG se lograron tiempos de ejecución transcurridos considerablemente elevados al aumentar la cantidad de soluciones iniciales en comparación al CG clásico. Esto, para las configuraciones experimentales que establecimos.

De manera a estudiar de mejor manera el potencial de la multiplicación de las matrices dispersas por múltiples vectores, la cual es aplicada a los métodos por bloques en general, como trabajo futuro se incluye la evaluación de estos algoritmos sobre implementaciones de alto desempeño. Específicamente, con librerías optimizadas, entornos de computación distribuida y sobre GPU. Adicionalmente se pueden realizar experimentos empleando técnicas de precondicionamiento para las matrices.

Apéndice A

Códigos

A.1. cg.m

```
function [x, r, iter, resvec] = cg(A, b, tol, maxit, x)

% CG   Conjugate Gradient. Implementation proposed in "Iterative Methods
%   for Linear and Nonlinear Equations" by C. T. Kelley.
%   X = CG(A,B) attempts to solve A*X=B. The matrix A must be symmetric
%   and positive definite and the right hand side vector B must have length
%   N.
%
%   X = CG(A,B,TOL) specifies the tolerance value for the residual
%   norm. The defaul value is 1e-6.
%
%   X = CG(A,B,TOL,MAXIT) specifies the maximum number of iterations to
%   be performed. The default value is the number of columns of A.
%
%   X = CG(A,B,TOL,MAXIT,X) specifies the initial guess. The default
%   value is 0 (zero vector).
%
%   [X,R] = CG(A,B,...) also returns the residual.
%
%   [X,R,ITER] = CG(A,B,...) also returns the iterations performed.
%
%   [X,R,ITER,RESVEC] = CG(A,B,...) also returns a vector of the
%   residual norms at each iteration.

    if nargin < 5
        x = zeros(numel(b),1);
        if nargin < 4
            [~,cols] = size(A);
            maxit = cols;
            if nargin < 3
                tol = 1e-6;
            end
        end
    end

    resvec = zeros(1,maxit+1);

    r = b - A * x; resvec(1) = norm(r);
    rho = r' * r;
    iter = 1;

    threshold = tol * norm(b);
```

```
    while sqrt(rho) > threshold && iter < maxit
        if iter == 1
            p = r;
        else
            beta = rho / rho_old;
            p = r + beta * p;
        end

        w = A * p;

        alpha = rho / (p' * w);
        x = x + alpha * p;
        r = r - alpha * w; resvec(iter+1) = norm(r);

        rho_old = rho;
        rho = r' * r;

        iter = iter + 1;
    end

    resvec = resvec(1:iter);

end
```

A.2. bcg.m

```
function [x, r, iter, resvec] = bcg(A, b, tol, maxit, X, step)

% BCG   Block Conjugate Gradient with combination of solutions for a single
%   right-hand side.
%   X = BCG(A,B) attempts to solve A*X=B. The matrix A must be symmetric
%   and positive definite and the right hand side vector B must have length
%   N.
%
%   X = BCG(A,B,TOL) specifies the tolerance value for the residual
%   norm. The defaul value is 1e-6.
%
%   X = BCG(A,B,TOL,MAXIT) specifies the maximum number of iterations to
%   be performed. The default value is the number of columns of A.
%
%   X = BCG(A,B,TOL,MAXIT,X) specifies the matrix of initial guesses. The
%   default value is a randomly generated n-by-m matrix with m = 2.
%
%   X = BCG(A,B,TOL,MAXIT,X,STEP) specifies the number of iterations to be
%   performed before a convergence check.
%
%   [X,R] = BCG(A,B,...) also returns the residual.
%
%   [X,R,ITER] = BCG(A,B,...) also returns the iterations performed.
%
%   [X,R,ITER,RESVEC] = CG(A,B,...) also returns a vector of the
%   residual norms at each convergence check.

    m = 2;
    [~,n] = size(A);
    if nargin < 6
        step = 2;
        if nargin < 5
            X = randn(n,m);
            if nargin < 4
                maxit = n;
                if nargin < 3
                    tol = 1e-6;
                end
            end
        end
    end
    if nargin > 4
        [~,m] = size(X);
    end

    resvec = zeros(1,floor(maxit/step)+1);

    e = ones(m,1);

    R = b*e' - A*X;
    rho = R' * R;
    iter = 1; r_i = 1;

    threshold = tol * norm(b);
    while iter < maxit
        if mod(iter,step) == 0
            [~,W] = qr(R,0);
            xi = W' \ (W \ e);
            xi_f = 1 / (e' * xi);
            r = xi_f * R * xi; norm_r = norm(r); resvec(r_i) = norm_r; r_i = r_i +
                1;
            if norm_r <= threshold
                x = xi_f * X * xi;
                resvec = resvec(1:r_i-1); return;
```

```
            end
        end
        if iter == 1
            P = R;
        else
            Psi = rho_old \ rho;
            P = R + P * Psi;
        end

        Q = A * P;

        Lambda = (P' * Q) \ rho;
        X = X + P * Lambda;
        R = R - Q * Lambda;

        rho_old = rho;
        rho = R' * R;

        iter = iter + 1;
    end

    [~,W] = qr(R,0);
    xi = W' \ (W \ e);
    xi_f = 1 / (e' * xi);
    x = xi_f * X * xi;
    r = xi_f * R * xi; resvec(r_i) = norm(r); resvec = resvec(1:r_i);

end
```

Bibliografía

[1] Khalid Ahmad, Anand Venkat y Mary Hall. "Optimizing LOBPCG: Sparse Matrix Loop and Data Transformations in Action". En: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2016, págs. 218-232.

[2] Ariful Azad y col. "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication". En: *SIAM Journal on Scientific Computing* 38.6 (2016), págs. C624-C651.

[3] Grey Ballard y col. "Communication lower bounds and optimal algorithms for numerical linear algebra". En: *Acta Numerica* 23 (2014), págs. 1-155.

[4] Grey Ballard y col. "Minimizing communication in numerical linear algebra". En: *SIAM Journal on Matrix Analysis and Applications* 32.3 (2011), págs. 866-901.

[5] Grey Ballard y col. "Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds". En: *arXiv preprint arXiv:1202.3177* (2012).

[6] CG Broyden. "Block conjugate gradient methods". En: *Optimization methods and Software* 2.1 (1993), págs. 1-17.

[7] Erin Claire Carson. "Communication-Avoiding Krylov Subspace Methods in Theory and Practice". Tesis doct. University of California, Berkeley, 2015.

[8] Erin Carson y col. "Write-avoiding algorithms". En: *EECS Department, University of California, Berkeley, Tech. Rep.* UCB/EECS-2015-163 (2015).

[9] Carlos Carvalho. "The Gap between Processor and Memory Speeds". En: *Proc. of IEEE International Conference on Control and Automation*. 2001.

[10] Anthony T Chronopoulos y Andrey B Kucherov. "Block s-step Krylov iterative methods". En: *Numerical Linear Algebra with Applications* 17.1 (2010), págs. 3-15.

[11] AT Chronopoulos y Charles William Gear. "s-Step iterative methods for symmetric linear systems". En: *Journal of Computational and Applied Mathematics* 25.2 (1989), págs. 153-168.

[12] T. A. Davis e Y. Hu. "The University of Florida Sparse Matrix Collection". En: *ACM Transactions on Mathematical Software* 38.1 (2011), 1:1-1:25. URL: `http://www.cise.ufl.edu/research/sparse/matrices`.

[13] James Demmel y col. "Avoiding communication in sparse matrix computations". En: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE. 2008, págs. 1-12.

[14] Jack Dongarra. "High-Performance Computing, Performance Evaluation, and Trends". En: *University of Tennessee and Oak Ridge National Laboratory* (). URL: `http://www.netlib.org/utk/people/JackDongarra/SLIDES/mit198/tsld005.htm`.

[15] Jack Dongarra y col. *Applied mathematics research for exascale computing*. Inf. téc. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.

[16] Victor Eijkhout, E. Chow y R. van de Geijn. *Introduction to High Performance Scientific Computing*. 2nd. lulu.com, 2014.

[17] YT Feng, DRJ Owen y D Perić. "A block conjugate gradient method applied to linear systems with multiple right-hand sides". En: *Computer methods in applied mechanics and engineering* 127.1-4 (1995), págs. 203-215.

[18] Magnus Rudolph Hestenes y Eduard Stiefel. *Methods of conjugate gradients for solving linear systems*. Vol. 49. NBS, 1952.

[19] Eun-Jin Im y Katherine A Yelick. *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.

[20] Eun-Jin Im, Katherine Yelick y Richard Vuduc. "Sparsity: Optimization framework for sparse matrix kernels". En: *International Journal of High Performance Computing Applications* 18.1 (2004), págs. 135-158.

[21] Alan Jennings. "Influence of the eigenvalue spectrum on the convergence rate of the conjugate gradient method". En: *IMA Journal of Applied Mathematics* 20.1 (1977), págs. 61-72.

[22] C T Kelley. "Iterative Methods for Linear and Nonlinear Equations". En: *Raleigh, N. C.: North Carolina State University* (1995).

[23] Xing Liu y col. "Improving the performance of dynamical simulations via multiple right-hand sides". En: *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE. 2012, págs. 36-47.

[24] Sally A McKee. "Reflections on the memory wall". En: *Proceedings of the 1st conference on Computing frontiers*. ACM. 2004, pág. 162.

[25] William McLean. *Poisson Solvers*. 2004.

[26] Marghoob Mohiyuddin y col. "Minimizing communication in sparse matrix solvers". En: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM. 2009, pág. 36.

[27] Dianne P O'Leary. "The block conjugate gradient algorithm and related methods". En: *Linear algebra and its applications* 29 (1980), págs. 293-322.

[28] *Roofline Performance Model*. `https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/`.

[29] Carlos Sauer. *Circuitos Electrónicos 2D: Simulación, Control y Refrigeración*. Trabajo Final de Grado. Universidad Nacional de Asución, 2009.

[30] Jonathan Richard Shewchuk y col. *An introduction to the conjugate gradient method without the agonizing pain*. 1994.

[31] Edgar Solomonik y col. "A communication-avoiding parallel algorithm for the symmetric eigenvalue problem". En: *arXiv preprint arXiv:1604.03703* (2016).

[32] Gilbert Strang. *Linear Algebra and Its Applications*. 4th. Brooks/Cole, 2006.

[33] Rich Vuduc y col. "Performance optimizations and bounds for sparse matrix-vector multiply". En: *Supercomputing, ACM/IEEE 2002 Conference*. IEEE. 2002, págs. 26-26.

[34] Samuel Williams, Andrew Waterman y David Patterson. "Roofline: an insightful visual performance model for multicore architectures". En: *Communications of the ACM* 52.4 (2009), págs. 65-76.

[35] Wm A Wulf y Sally A McKee. "Hitting the memory wall: implications of the obvious". En: *ACM SIGARCH computer architecture news* 23.1 (1995), págs. 20-24.