Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística

Rafaela Correia Brum

Estudo Comparativo de Métodos de Resolução de Sistemas Lineares para Matrizes Cheias e Esparsas

Rio de Janeiro
2019

Rafaela Correia Brum

Estudo Comparativo de Métodos de Resolução de Sistemas Lineares para Matrizes Cheias e Esparsas

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Orientadores: Prof.ª Dra. Maria Clicia Stelling de Castro
Prof.ª Dra. Cristiane Oliveira de Faria

Rio de Janeiro
2019

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC-A

B893 Brum, Rafaela Correia.
Estudo comparativo de métodos de resolução de sistemas lineares para matrizes cheias e esparsas / Rafaela Correia Brum. – 2019.
105 f. : il.

Orientadoras: Maria Clicia Stelling de Castro
Cristiane Oliveira de Faria
Dissertação (Mestrado em Ciências Computacionais) - Universidade do Estado do Rio de Janeiro, Instituto de Matemática e Estatística.

1. Sistemas lineares - Teses. 2. Computação – Matemática – Teses. I. Castro, Maria Clicia Stelling de. II. Faria, Cristiane Oliveira de. III. Universidade do Estado do Rio de Janeiro. Instituto de Matemática e Estatística. IV. Título.

CDU 512.644

Patricia Bello Meijinhos – CRB-7/5217 – Responsável pela elaboração da ficha catalográfica

_____ _____

Assinatura Data

Rafaela Correia Brum

Estudo Comparativo de Métodos de Resolução de Sistemas Lineares para Matrizes Cheias e Esparsas

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Aprovada em 06 de Agosto de 2019
Banca Examinadora:

Prof.ª Dra. Maria Clicia Stelling de Castro (Orientador)
Instituto de Matemática e Estatística - UERJ

Prof.ª Dra. Cristiane Oliveira de Faria (Orientador)
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Luiz Mariano Paes De Carvalho Filho
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Diego Nunes Brandão
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca - CEFET/RJ

Rio de Janeiro
2019

AGRADECIMENTOS

A Deus em primeiro lugar, por ter me permitido chegar até aqui.

As minhas orientadoras, Cristiane Faria e Maria Clicia Castro, por todas as orientações que levaram à conclusão deste trabalho.

Aos meus amigos, por todas os momentos de distração e compreensão que me proporcionaram durante estes anos.

Aos meus pais, Antonino Brum e Mônica Brum, por todo apoio e carinho dado a mim desde meu nascimento.

A minhas irmãs, Renata Brum e Roberta Brum, por serem verdadeiras amigas para mim.

Aos meus sobrinhos, Lucas Brum, Maria Alice dos Santos e Pedro dos Santos, por me trazerem um sorriso e ânimo nos momentos mais difíceis.

Ao meu namorado, Raul de Queiroz, por me acompanhar nesse trabalho, ser meu apoio em todos os momentos e compreender minhas angústias e medos no decorrer do mestrado.

Por fim, agradeço a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) pela bolsa de mestrado concedida a mim (Código de Financiamento 001).

RESUMO

BRUM, Rafaela. *Estudo Comparativo de Métodos de Resolução de Sistemas Lineares para Matrizes Cheias e Esparsas*. 2019. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro.

Sistemas lineares são modelos recorrentes na resolução de problemas em diversas áreas de conhecimento como simulação de fluidos, *design* de componentes eletrônicos, e até em áreas recentes como, *Deep Learning* e análise de dados de redes sociais. Quanto mais precisa a descrição do problema for, mais variáveis devemos conhecer para resolver os sistemas lineares, chegando a sistemas que passam dos *terabytes* de armazenamento. Por isso, é importante entendermos os métodos de resolução para propormos melhorias em seus algoritmos, como por exemplo, otimizações de memória e pontos de paralelismo. O objetivo principal dessa dissertação é realizar estudos comparativos entre métodos de resolução de sistemas lineares, para aprofundar o entendimento de cada método, além de observar se existe alguma predisposição de paralelismo nos métodos abordados. O primeiro estudo considerou métodos diretos de resolução de sistemas lineares com matrizes cheias. Resultados comparando precisão de variáveis, tempo de execução e a influência de parâmetros de compilação são apresentados. No segundo estudo, métodos iterativos foram aplicados em matrizes cheias. Resultados comparando um método recente, o método chamado de *Delayed Over Relaxation* (DOR) (ANTUONO; COLICCHIO, 2016) e um método distribuído, o Gauss-Seidel distribuído (SHANG, 2009) também são apresentados. Este último método teve um *speedup* de 1,15. Por fim, o último estudo focou nos métodos iterativos de projeção para matrizes esparsas. Neste estudo comparamos métodos clássicos como o GMRES(m) (SAAD; SCHULTZ, 1986) com métodos recentes, como o αGMRES (BAKER; JESSUP; KOLEV, 2009) e o *Heavy Ball* GMRES (IMAKURA; LI; ZHANG, 2016). Além disso, apresentamos um estudo inicial de paralelização do método *Heavy Ball* GMRES, que apresentou um *speedup* de 2,11.

Palavras-chave: Sistemas Lineares. Métodos Diretos e Iterativos. Métodos de Projeção. Matrizes Cheias e Esparsas. Otimização de memória. Paralelização.

ABSTRACT

BRUM, Rafaela. *Comparative Study of Methods for Solving Full and Sparse Linear Systems*. 2019. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro.

Linear systems are recurrent models to solve problems in many areas as dynamic fluids, electronic computer design and even recent areas as Deep Learning and social media data analysis. As the linear system gets more precise to describe the real problem, more variables are needed to correctly solve the system, easily needing terabytes of storage. Having said that, it is important to understand the resolution methods to propose better algorithms, as memory optimizations and parallelism points. The objective of this work is do many comparative studies between resolution methods of linear systems to better understand each one and observe if there is any paralellism spot in the given methods. The first study was with direct methods to solve full matrices linear systems. In this study, we compare variable precision, execution time and compilation flags. The second study evaluated iterative methods to solve full matrices linear systems. In this study, we compare a recent method, named DOR (ANTUONO; COLICCHIO, 2016), and a distributed method, the Distributed Gauss-Seidel (SHANG, 2009). Our reproduction of this last method obtained 1,15 of speedup. Finally, the last study focused on iterative projective methods to solve sparse matrices linear systems. In this last study, we compare classical methods like GMRES(m) (SAAD; SCHULTZ, 1986) with recent methods as αGMRES (BAKER; JESSUP; KOLEV, 2009) and the Heavy Ball GMRES (IMAKURA; LI; ZHANG, 2016). Besides, we show an initial study of parallel points in the Heavy Ball GMRES method, which obtained an speedup of 2.11.

Keywords: Linear systems. Direct and Iterative Methods. Projection Methods. Full and Sparse Matrices. Memory optimization. Parallelism.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ALGORITMOS

SUMÁRIO

INTRODUÇÃO

Diversos problemas nas áreas de Engenharias e Computação, como por exemplo, *design* de componentes de computadores, problemas térmicos, dispersão acústica e escoamento de fluidos são descritos por modelos matemáticos formados por sistemas de equações lineares ou não-lineares. Estes modelos podem conter equações simples, encontradas empiricamente, relacionando as variáveis em estudo, como também podem conter equações diferenciais ordinárias ou parciais para descrever a relação entre as variáveis estudadas. Quanto mais características do problema real são consideradas, mais complexo se torna o modelo e, consequentemente, maior a quantidade de dados relacionados a ele. Tais características demandam métodos mais elaborados e eficazes. Exemplos mais recentes nas áreas de aplicações de aprendizado profundo (*Deep Learning*) (DENG; YU, 2014), aplicações de computação científica (LI et al., 2016) e em aplicações voltadas para redes sociais (*Social Media*) (PENG; PARK, 2011; MCCAUGHEY et al., 2014) comprovam a necessidade de novos métodos de resolução.

Outro ponto a ser abordado é como encontrar a solução por tais modelos matemáticos. Como eles podem se tornar grandes e complexos, é desejável que possamos representá-los em modelos computacionais e dessa maneira utilizar computadores para encontrar as soluções em tempo hábil. A área que se propõe a encontrar esse tipo de solução é denominada de Matemática Computacional. Ela engloba a proposta de modelos matemáticos que reproduzem cada vez melhor o problema real, e consideram o desenvolvimento de métodos numéricos para resolução de sistemas lineares focando em questões computacionais, como a velocidade de processamento e a capacidade de armazenamento dos computadores.

Sistemas lineares são denotados por m equações com n incógnitas x_i como em: $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i$. Esses sistemas são usualmente representados na forma matricial $A\vec{x} = \vec{b}$, sendo A a matriz de coeficientes, $\vec{x}$ o vetor solução e $\vec{b}$ o vetor de valores independentes (LIPSCHUTZ, 2000), o que facilita a estrutura computacional. A teoria apresenta duas metodologias para se resolver os sistemas lineares denominadas de métodos diretos e métodos iterativos. Os métodos diretos solucionam o sistema com um número finito de operações na matriz de coeficientes A e em sua maioria estão associados com processos de decomposição. Os métodos iterativos partem de um valor inicial e calculam sucessivas aproximações até atingir uma solução dentro de uma métrica pré-definida e que será explicada mais adiante. A convergência dos métodos, tanto os diretos e os iterativos, depende de alguns fatores como, por exemplo, o condicionamento da matriz de coeficientes (CUNHA, 2003).

Com a demanda de matrizes cada vez maiores, uma saída muito utilizada é tentar paralelizar os métodos e funções, como em (COURTECUISSE; ALLARD, 2009; SCHUBERT et al., 2011; YZELMAN; ROOSE, 2014). Em (COURTECUISSE; ALLARD, 2009), os autores apresentam uma implementação do método Gauss-Seidel, um método iterativo para matrizes cheias, para GPUs (Unidades de Processamento Gráfico, ou *Graphics Processing Units* em inglês, uma arquitetura com muitos núcleos de processamento) que é executado em paralelo. Já em (SCHUBERT et al., 2011), é apresentada uma imple-

mentação paralela do produto de matriz por um vetor que utiliza tanto o paradigma da troca de mensagens quanto o de memória compartilhada na comunicação dos processos. Em (YZELMAN; ROOSE, 2014), os autores propõem uma implementação deste mesmo produto usando o paradigma de memória compartilhada na comunicação dos processos executados em paralelo.

No artigo (WOLFSON-POU; CHOW, 2017), os autores propõem o método *Southwell* distribuído, um método criado para competir com o algoritmo Block Jacobi de pré-condicionamento. Já no artigo (EFTEKHARI; BOLLHÖFER; SCHENK, 2018), os autores criaram um algoritmo capaz de desempenhar e escalar bem em sistemas esparsos de muitas dimensões. A solução proposta é a utilização de métodos de máxima verossimilhança de segunda ordem, identificando-se os gargalos no processo para paralelização com MPI–OpenMP, duas APIs de programação paralela e distribuída, respectivamente.

Mas para propor e obter otimizações e implementações paralelas dos métodos utilizados, é preciso entender cada implementação sequencial e a teoria por trás deles. Por isso, essa dissertação tem como objetivo principal realizar estudos comparativos entre métodos de resolução de sistemas lineares, aprofundando o entendimento de cada um deles, além de observar se existe alguma predisposição de paralelismo nos métodos estudados.

Uma consequência da ideia fundamental desta dissertação é entender a teoria por trás de cada método. Por isso, foram feitos três estudos separados. O primeiro estudo comparativo foi feito com métodos diretos de resolução de sistemas lineares cheios (ou com matrizes cheias). Isso porque estes são os métodos com a teoria e implementações mais simples e ainda são bastante utilizados. Em seguida, um segundo estudo foi feito com métodos iterativos para sistemas lineares cheios, por terem uma teoria mais complexa que os métodos diretos, mas com simplicidade de implementação. Por fim, o último estudo foi realizado com métodos iterativos de projeção utilizados com matrizes esparsas. Estes métodos são construídos a partir dos espaços de Krylov e a implementação dos métodos de projeção são mais complexas. Além disso, eles exigem uma mudança na forma de armazenamento computacional da matriz.

O restante do trabalho está organizado da seguinte forma:

O Capítulo 1 aborda os conceitos matemáticos usados nesta dissertação, como as definições básicas, tipos de matrizes e subespaços de Krylov. O Capítulo 2 aborda as definições e conceitos computacionais relacionados a execução paralela como *speedup*, por exemplo. O Capítulo 3 descreve o estudo comparativo entre seis métodos diretos para matrizes cheias, entre eles o método de Decomposição de Cholesky. Este capítulo é a compilação de três artigos publicados na IV Escola Regional de Alto Desempenho do Rio de Janeiro (ERAD-RJ), e apresentado no fórum de Pós-Graduação; outro para o XXXVIII Congresso Nacional de Matemática Aplicada e Computacional (CNMAC) (BRUM; CASTRO; FARIA, 2018a), apresentado na forma de pôster; e um artigo submetido para a revista TEMA (*Trends in Applied and Computational Mathematics*), como extensão do artigo apresentado no CNMAC. Este último artigo ainda está em processo de revisão.

O Capítulo 4 apresenta um segundo estudo comparativo, agora com métodos iterativos para matrizes cheias. Também foram avaliados seis métodos, incluindo o método DOR, um método recente (ANTUONO; COLICCHIO, 2016) que até o momento do estudo foram encontradas apenas simulações da convergência do método e nenhuma implementação, e o método Gauss-Seidel distribuído (SHANG, 2009), que utiliza o paradigma de troca de mensagens para a comunicação entre os processos paralelos. Este capítulo é um compilado de um artigo enviado para o XXI Encontro Nacional de Modelagem Computacional (ENMC) (BRUM; CASTRO; FARIA, 2018b), apresentado em uma sessão oral.

O Capítulo 5 aborda o último estudo comparativo realizado, entre quatro métodos iterativos que utilizam um subespaço de Krylov para encontrar a solução aproximada. Estes quatro métodos são métodos de projeção ortogonal que utilizam o subespaço de Krylov sobre o resíduo tanto para o subespaço de procura como para o subespaço de restrição. Os métodos estudados são o GMRES reiniciado (GMRES(m)) (SAAD; SCHULTZ, 1986), o GMRES flexível (FGMRES) (SAAD, 1993), o αGMRES (BAKER; JESSUP; KOLEV, 2009) e o *Heavy Ball* GMRES (HBGMRES) (IMAKURA; LI; ZHANG, 2016).

Por fim, apresentamos as considerações finais e propostas de trabalhos futuros.

1 CONCEITOS MATEMÁTICOS BÁSICOS

Este capítulo introduz conceitos matemáticos que são utilizados ao longo deste trabalho, além de suas representações computacionais sempre que possível Também são definidos os métodos diretos e métodos iterativos, usados para resolver os sistemas lineares de equações, além de subespaços e algumas de suas características (RUGGIERO; LOPES, 1997; LIPSCHUTZ, 2000; WATKINS, 2002; CUNHA, 2003; SAAD, 2003).

1.1 Vetores

Os vetores são coleções ordenadas de objetos, que são suas componentes. Um vetor $v \in \mathbb{R}^n$ pode ser representado por $v = (v_1, v_2, \ldots, v_n)$, sendo n o tamanho do vetor. Computacionalmente, os vetores são armazenados em *arrays*, que são uma estrutura de dados que corresponde a uma sequência de posições de memória que armazenam um mesmo tipo de dado.

A seguir são apresentadas algumas definições considerando vetores.

Definição 1. *O Produto Interno entre dois vetores é dado por*

$$\langle u, v\rangle = u_1 \times v_1 + u_2 \times v_2 + \ldots + u_n \times v_n.$$

Definição 2. *Dizemos que dois vetores são ortogonais quando o produto interno entre eles é igual a zero, ou seja,* $\langle u, v\rangle = 0$.

Definição 3. *A Norma Euclidiana de um vetor é dada por*

$$\|v\| = \sqrt{\sum_{k=1}^{n} v_i^2} \equiv \sqrt{\langle v, v\rangle}.$$

Definição 4. *Dizemos que um vetor é unitário quando sua norma é igual a 1, ou seja,* $\|v\| = 1$.

Definição 5. *Dizemos que dois vetores u e v são ortonormais quando eles são ortogonais e unitários, ou seja,* $\langle u, v\rangle = 0$ *e* $\|u\| = 1$ *e* $\|v\| = 1$.

Definição 6. *Dizemos que temos uma combinação linear de um vetor por outros quando podemos escrever esse primeiro vetor em função dos outros. Ou seja, uma combinação linear de z por x e y é igual a* $z = \alpha x + \beta y$.

1.2 Matrizes

As matrizes são compostas por um conjunto de m vetores, e podem ser representadas como $A = (a_{11}, a_{12}, \ldots, a_{mn})$ ou como em (1.1), onde m é a quantidade de linhas da matriz

e n a quantidade de colunas. Computacionalmente, elas podem ser armazenadas como um *array* de *arrays* ou como um único *array* de tamanho $m \times n$.

A seguir são apresentadas algumas definições considerando matrizes.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \tag{1.1}$$

Definição 7. *A ordem de uma matriz é dada pelo produto do seu número de linhas e colunas. Por exemplo, a ordem da matriz A apresentada em (1.1) é $m \times n$. Quando a quantidade de linhas é igual à de colunas, ou seja, $m = n$, dizemos que a matriz é quadrada e sua ordem é m.*

Definição 8. *Uma matriz A é uma matriz cheia quando a quantidade de valores não-nulos ($a_{ij} \neq 0$) da matriz é maior que 50% do total de elementos. Por exemplo, a matriz a seguir é uma matriz cheia.*

$$A = \begin{bmatrix} x & 0 & x & x & x & x & x & x \\ x & x & x & x & x & x & x & 0 \\ x & x & x & x & x & x & x & x \\ x & x & x & x & 0 & x & x & x \\ 0 & x & x & x & x & x & x & x \\ x & x & x & x & x & x & 0 & x \\ x & 0 & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \end{bmatrix}$$

Definição 9. *Uma matriz A é uma matriz esparsa quando a quantidade de valores nulos ($a_{ij} = 0$) da matriz é maior que 50% do total de elementos. Por exemplo, a matriz a seguir é uma matriz esparsa.*

$$A = \begin{bmatrix} x & 0 & 0 & x & 0 & 0 & 0 & 0 \\ x & x & 0 & x & 0 & 0 & 0 & 0 \\ x & 0 & x & x & x & 0 & 0 & 0 \\ 0 & 0 & x & x & 0 & x & 0 & 0 \\ 0 & 0 & 0 & 0 & x & 0 & 0 & 0 \\ 0 & x & 0 & 0 & 0 & x & 0 & 0 \\ 0 & 0 & x & 0 & 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}$$

Definição 10. *Uma matriz L é uma matriz triangular inferior quando os elementos não-nulos estão agrupados na parte inferior da matriz, ou seja, quando $l_{i,j} \neq 0$ quando $i \geq j$ como mostrado a seguir.*

$$L = \begin{bmatrix} l_{1,1} & 0 & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & 0 & \cdots & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{m,1} & l_{m,2} & l_{m,3} & \cdots & l_{m,m} \end{bmatrix}$$

Definição 11. *Uma matriz U é uma matriz triangular superior quando os elementos não-nulos estão agrupados na parte superior da matriz, ou seja, quando $u_{i,j} \neq 0$ quando $i \leq j$ como mostrado a seguir.*

$$U = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,m} \\ 0 & u_{2,2} & u_{2,3} & \cdots & u_{2,m} \\ 0 & 0 & u_{3,3} & \cdots & u_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{m,m} \end{bmatrix}$$

Definição 12. *Uma matriz D é uma matriz diagonal quando os únicos elementos não-nulos da matriz estão na diagonal principal, ou seja, quando $u_{i,j} \neq 0$ quando $i = j$ como mostrado a seguir.*

$$D = \begin{bmatrix} d_{1,1} & 0 & 0 & \cdots & 0 \\ 0 & d_{2,2} & 0 & \cdots & 0 \\ 0 & 0 & d_{3,3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & d_{m,m} \end{bmatrix}$$

Definição 13. *Uma matriz A é uma matriz identidade, I, quando os únicos elementos não-nulos da matriz estão na diagonal principal e seus valores são iguais a 1 como mostrado a seguir.*

$$I = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

Definição 14. *Uma matriz A é uma matriz não singular, ou invertível, quando existe uma matriz A^{-1}, chamada matriz inversa de A, tal que $AA^{-1} = A^{-1}A = I$, sendo I a matriz identidade.*

Definição 15. *Uma matriz A^t é a transposta de A quando invertemos as linhas pelas colunas de A.*

Definição 16. *Uma matriz A é simétrica quando seus elementos i,j e j,i são iguais, isto é, quando $a_{i,j} = a_{j,i}$. Assim, $A^t = A$.*

Definição 17. *Uma matriz A é antissimétrica quando seus elementos i,j e j,i são elementos opostos, isto é, quando $a_{i,j} = -a_{j,i}$. Então $A^t = -A$.*

Definição 18. *Uma matriz H é uma matriz de Hessenberg superior quando tem elementos não-nulos na parte superior da matriz e na diagonal logo abaixo da diagonal principal, como mostrado a seguir*

$$H = \begin{bmatrix} x & x & x & x & x & x \\ x & x & x & x & x & x \\ 0 & x & x & x & x & x \\ 0 & 0 & x & x & x & x \\ 0 & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \end{bmatrix}$$

Definição 19. *Uma matriz simétrica A é definida positiva se, para qualquer vetor $x \neq 0$, tem-se a relação $\langle Ax, x\rangle \geq 0$.*

Definição 20. *Uma matriz Q é ortogonal se suas colunas são vetores ortonormais. Além disso, uma matriz é ortogonal quando $Q^{-1} = Q^t$.*

Definição 21. *Uma matriz A está na forma escalonada se valerem as duas condições a seguir (em que um elemento líder não nulo de uma linha de A significa o primeiro elemento não nulo dessa linha):*

1. *Todas as linhas nulas, se existirem, estão agrupadas junas nas linhas inferiores da matriz.*
2. *Cada elemento líder não nulo de uma linha está a direita do elemento líder não nulo da linha precedente.*

Ou seja, $A = [a_{ij}]$ é uma matriz escalonada se existirem entradas não nulas $a_{ij_1}, a_{ij_2}, \ldots, a_{ij_r}$ com $j_1 < j_2 < \cdots < j_r$.

Definição 22. *A norma do máximo, ou norma 1 de uma matriz A, $\|A\|_1$, é definida por*

$$\|A\|_1 = \max_{j=1,\ldots,m} \sum_{i=1}^{n} |a_{ij}|$$

Definição 23. *Uma matriz R é uma matriz de rotação quando modifica um vetor (ou uma matriz) sem alterar a sua norma do máximo.*

Definição 24. *O espectro de uma matriz A, denominado por $\sigma(A)$, é o conjunto de todos os autovalores de A, isto é, todos os valores λ que satisfazem $det(A - \lambda I) = 0$.*

1.3 Subespaço vetorial

Um espaço vetorial V é um conjunto de vetores definido pela Definição 25 e um subespaço vetorial W é definido pela Definição 26.

Definição 25. *Seja $x, y, z \in V$ e $\alpha, \alpha' \in \mathbb{R}$. Então, V é um espaço vetorial se:*

- $x + y = y + x, \forall x,y \in V$
- $(x + y) + z = x + (y + z), \forall x,y,z \in V$
- $x + 0 = 0 + x = x, \forall x \in V$
- $x + (-x) = 0, \forall x \in V$
- $\alpha'(\alpha x) = (\alpha'\alpha)x, \forall \alpha, \alpha' \in \mathbb{R}$ *e* $\forall x \in V$
- $\alpha(x + y) = \alpha x + \alpha y, \forall \alpha \in \mathbb{R}$ *e* $\forall x,y \in V$
- $\alpha(xy) = (\alpha x)y, \forall \alpha \in \mathbb{R}$ *e* $\forall x,y \in V$
- $x1 = 1x = x, \forall x \in V$

Definição 26. *Seja V um espaço vetorial e $W \subset V$. W é um subespaço vetorial se as duas condições seguintes são satisfeitas:*

- $\forall x, y \in W$ *e* $\alpha, \beta \in \mathbb{R}$ *temos que* $\alpha x + \beta y \in W$
- $0 \in W$

Definição 27. *Um conjunto* $S = u_1, u_2, \ldots, u_n$ *de vetores é uma base do subespaço* W *se:*

- *Os vetores de* S *são linearmente independentes. Isto é, nenhum dos vetores de* S *pode ser escrito como uma combinação linear dos outros vetores de* S*;*
- *O conjunto* S *gera o subespaço* W*. Assim, todos os vetores de* W *podem ser escritos como uma combinação linear dos vetores do conjunto* S*.*

Definição 28. *Um subespaço* W *é invariante quando, dada uma transformação linear* $T : \mathbb{R}^n \to \mathbb{R}^n$*, todos os vetores* $v \in W$ *são transformados por* T *em vetores que estão contidos em* W*, ou seja,*

$$v \in W \Rightarrow T(v) \in W.$$

Definição 29. *O subespaço de Krylov de* A *e* v *é definido como um subespaço gerado pelas potências da matriz quadrada* A *multiplicadas pelo vetor* $v \in \mathbb{R}^n$*, como mostrado a seguir:*

$$\mathcal{K}_k(A, v) = ger\{v, Av, A^2v, \ldots, A^{m-1}v\}.$$

1.4 Sistemas Lineares

Um sistema de equações lineares, ou simplesmente sistema linear, é descrito na forma padrão mostrada em (1.2). Os coeficientes a_{ij} e b_i são constantes, onde $i = 1, \ldots, m$ e $j = 1, \ldots, n$, e x_i são as incógnitas, com $i = 1, \ldots, n$.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad\vdots \qquad\qquad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \tag{1.2}$$

Esse tipo de sistema pode ser representado computacionalmente através de uma matriz de coeficientes A de ordem $m \times n$ e dois vetores coluna, o vetor x para as variáveis e o vetor B para os coeficientes independentes. Assim, a representação matricial de um sistema linear $Ax = B$, que facilita a visualização computacional, é dado por (1.3).

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{1.3}$$

Definição 30. *Ao resolver um sistema linear* $Ax = b$*, econtramos três situações:*

1. *A matriz* A *é não singular. Existe uma única solução* x *dada por* $x = A^{-1}b$*.*
2. *A matriz* A *é singular e* b *pertence ao conjunto imagem de* A*. Existem infinitas soluções.*
3. *A matriz* A *é singular e* b *não pertence ao conjunto imagem de* A*. Não existe solução.*

Definição 31. *A matriz aumentada M de um sistema linear é a matriz que acopla o vetor de coeficientes livres b ao final da matriz A de coeficientes, como mostrado a seguir.*

$$M = \left[\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} & b_m \end{array}\right]$$

Definição 32. *Uma matriz P é uma matriz pré condicionadora quando é utilizada para multiplicar o sistema linear $Ax = b$ e transformá-lo no sistema $P^{-1}Ax = P^{-1}b$ no intuito de facilitar a resolução do sistema em questão por um método iterativo.*

Definição 33. *O resíduo, ou vetor residual, r de uma solução x é dado por*

$$r = b - Ax$$

Existem diferentes métodos para resolver os sistemas lineares, como os métodos diretos, os iterativos e os de projeção. A seguir, estão definidos cada um desses métodos.

Definição 34. *Os métodos diretos de resolução de sistemas lineares encontram a solução do sistema, com possíveis erros de arredondamento, depois de um número finito de operações aritméticas.*

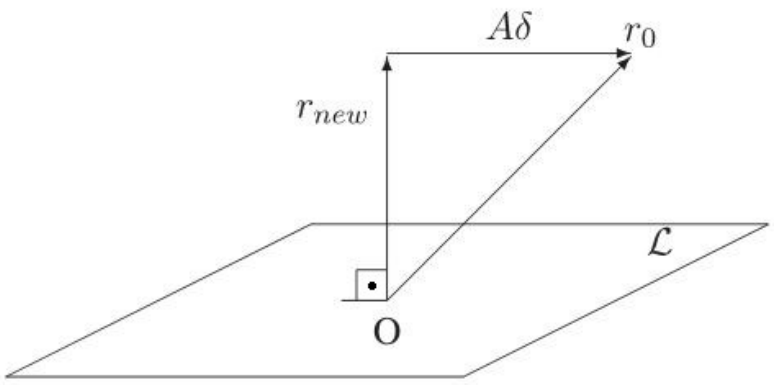
Definição 35. *Os métodos iterativos de resolução de sistemas lineares utilizam sucessivas aproximações da solução até atingir uma aproximação da solução do problema, com possíveis erros de arredondamento.*

Definição 36. *Os métodos de projeção são métodos que utilizam as propriedades dos subespaços vetoriais para encontrar a solução, através de um subespaço de procura $\mathcal{K}$ e um subespaço de restrição $\mathcal{L}$.*

Nos métodos de projeção, quando o subespaço de procura $\mathcal{K}$ tem dimensão m, em geral, são necessárias m restrições para atingir a solução aproximada do sistema linear. Assim, normalmente se impõe condições de ortogonalidade do vetor residual ($r = b - Ax$) com o subespaço de restrição $\mathcal{L}$. Esta restrição de ortogonalidade pode ser visualizada na Figura 1.

Existem dois tipos de métodos de projeção, os métodos de projeção ortogonal e os métodos de projeção oblíqua. Os métodos de projeção ortogonal usam o mesmo subespaço como base para $\mathcal{K}$ e para $\mathcal{L}$. Os métodos de projeção oblíqua utilizam subespaços diferentes para $\mathcal{K}$ e $\mathcal{L}$, que podem ou não ter relação entre eles.

Figura 1 – Visualização da ortogonalidade do resíduo com o subespaço de restrição $\mathcal{L}$.



Fonte: (SAAD, 2003)

1.5 **Outras definições**

Definição 37. *Uma sequência $S = a_n$ de números reais é monótona crescente quando a diferença de dois termos consecutivos é menor ou igual a zero, ou seja, $a_{i+1} - a_i \geq 0$.*

Definição 38. *Uma sequência $S = a_n$ de números reais é convergente quando existe um valor L no qual os valores de S tendem quando n tende ao infinito, ou seja, $L = \lim_{n\to\infty} a_n$.*

Definição 39. *Dizemos que e é o erro absoluto quando pegamos a a diferença do valor final pelo valor aproximado de uma computação.*

Definição 40. *Dizemos que ϵ é o erro relativo quando pegamos o erro absoluto e e dividimos pelo valor aproximado da computação.*

2 CONCEITOS COMPUTACIONAIS BÁSICOS

Este capítulo introduz alguns conceitos computacionais. São apresentados dois modelos de comunicação entre processos, por memória compartilhada e por troca de mensagens. Além disso, são apresentadas as definições de *overflow* e *speedup* (PATTERSON; HENNESSY, 2014; SEDGEWICK, 1990; LIPSCHUTZ, 2000; SAAD, 2003).

2.1 Base binária

Os computadores digitais trabalham internamente com dois níveis de tensão elétrica. Assim, o sistema binário de representação numérica é natural para representar as informações nestes computadores. Este sistema utiliza somente os algarismos zero e um para representar todas as informações. Por exemplo, o número 25 na base decimal é representado pela seguinte sequência de zeros e uns 11001 na base binária , enquanto o número 0,1875 na base decimal é representado por 0,0011 na base binária.

As variáveis de ponto flutuante de precisão simples tem tamanho de 32 bits e as variáveis de precisão dupla tem tamanho de 64 bits, sendo que um bit armazena 0 ou 1.

Como os valores numéricos são armazenados na base binária, pode haver erros como imprecisão na representação ou truncamento da representação. Por exemplo, o valor 0,2 na base decimal é representado por uma dízima periódica na base binária igual a $0{,}00110011\ldots$. Assim, não é possível armazenar a representação fiel do valor 0,2 da base decimal na memória do computador, tendo que truncar e armazenar uma parte dos dígitos necessários. Esse é o chamado erro de truncamento ou arredondamento por corte.

2.1.1 *Overflow* e *Underflow*

Overflow e *Underflow* são condições que ocorrem quando a representação binária de algum valor a ser armazenada precisa de uma quantidade grande de dígitos. Essa quantidade pode ser tão grande que ultrapasse a quantidade de *bits* fornecidas pelo programa ou pelo *hardware* para armazenar os dados. Essa condição é chamada de *overflow*, quando o valor é um número positivo muito grande ou número negativo muito grande e de *underflow* quando o número é muito próximo de zero.

Essa condição é mais provável de surgir a partir de operações de multiplicações ou de divisões. Estas operações aritméticas podem escalar a quantidade de dígitos necessários rapidamente.

2.2 Complexidade computacional

Para dizer a complexidade de um algoritmo, não podemos olhar para o tempo de execução em um computador, pois cada sistema computacional tem características de processamento e de memória diferentes. Por isso, de acordo com (SEDGEWICK, 1990), a complexidade de um algoritmo só pode ser dada ao observar o total de operações realizadas no processamento do pior caso possível. Essa análise é baseada na notação O.

Essa notação diz que uma função $g(N)$ é $O(f(N))$ quando existe uma constante c_0 e um valor N_0 para os quais $g(N)$ é menor que $c_0 \times f(N)$ para todo $N > N_0$, sendo N e N_0 inteiros. Ou seja, a função $g(N)$ não ultrapassa a $f(N)$ para todo valor de N maior que N_0.

Para calcular a função O de cada algoritmo, é necessário analisar linha a linha do código, e determinar quanto custa computacionalmente cada uma. As operações aritméticas e lógicas tem custo constante, assim como a atribuição também tem custo constante. Um *loop* tem o custo da quantidade de iterações multiplicado pelo custo das operações internas ao *loop*, que podem ser outro *loop*, operações aritméticas, comparações ou desvios. Um desvio do tipo *if-else* tem o custo total das operações realizadas no *if* ou o custo das operações realizadas no *else*.

Assim, em um primeiro momento é calculada a função completa do algoritmo. Depois, reduz-se a função para o termo de maior grau ou maior peso. Considere a função completa $O(N^3 + N^2)$, a função final é $O(N^3)$, uma vez que ela é a função de maior grau.

2.3 **Alocação de memória**

Todo programa no computador armazena espaços de memória para as suas variáveis, que são usadas durante o processamento. Existem dois tipos de alocação de memória: a alocação estática e a alocação dinâmica.

Na alocação estática de memória o programa define suas variáveis em tempo de compilação, incluindo o tamanho dos seus *arrays*. Esse tipo de alocação é considerado fixo. Uma vez que declarado um *array* não é possível alterar seu tamanho em tempo de execução. Este fato traz uma limitação ao programador e um possível desperdício de memória. É necessário conhecer o tamanho do *array* em tempo de compilação do programa, podendo, então, ser superestimado.

Na alocação dinâmica de memória as variáveis podem ser definidas em tempo de execução. O tamanho dos *arrays* podem ser conhecidos somente quando forem usados, dando maior liberdade ao programador. Essa alocação é mais flexível que a estática, uma vez que é possível realocar os *arrays* ao longo do processamento, de acordo com a necessidade de memória. Esta alocação utiliza o conceito de ponteiros. Os ponteiros são variáveis que armazenam os endereços de memória das áreas alocadas dinamicamente. Esta alocação proporciona uma utilização da memória mais eficiente, justamente pelo fato de poder realocar os espaços de memória a medida que for necessário.

2.3.1 *Alocação de memória para uma matriz esparsa*

Quando a matriz A é esparsa, ela não precisa ser armazenada como um *array* de *arrays*, nem como um único *array* de tamanho $n \times m$. Como a maioria dos valores são nulos, esses dois esquemas desperdiçam muita memória para armazenar a matriz. Por isso, esquemas específicos para armazenamento de matrizes esparsas foram desenvolvidos como em (IMECEK; LANGR; TVRDíK, 2012; LI; SUN; VUDUC, 2018). Em (IMECEK; LANGR; TVRDíK, 2012) é apresentada uma pesquisa com os formatos de armazenamento de matrizes esparsas mais clássicos e os mais recentes. Já em (LI; SUN; VUDUC, 2018), os autores apresentam um esquema de armazenamento baseado em coordenadas hierárquicas.

Um esquema consolidado e bastante utilizado para representar matrizes esparsas é o esquema CSR (*Compressed Sparse Row*), onde os valores não-nulos da matriz A são armazenados utilizando três vetores no total (SAAD, 2003). O primeiro vetor AA, de tamanho

nz (quantidade de valores não-nulos), armazena os valores reais da matriz ordenados por linha. O segundo vetor JA, também de tamanho nz, armazena o índice das colunas de cada valor real da matriz A. O terceiro vetor IA, de tamanho $N+1$ sendo N a ordem da matriz, indica o início de cada linha da matriz A nos dois outros vetores. Como exemplo, extraído de (SAAD, 2003), temos em (2.2) os três vetores que representam a matriz A (2.1).

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix} \tag{2.1}$$

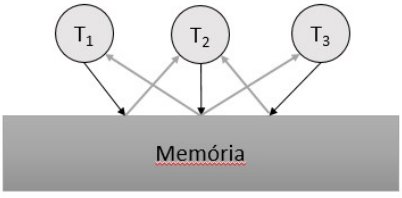
$$\begin{aligned} AA &= \{1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12\} \\ JA &= \{1; 4; 1; 2; 4; 1; 3; 4; 5; 3; 4; 5\} \\ IA &= \{1; 3; 6; 10; 12; 13\} \end{aligned} \tag{2.2}$$

2.4 **Programação paralela e distribuída**

A programação paralela e a distribuída permite que um programa seja executado em mais de um processo (ou *thread*) ao mesmo tempo. Na programação paralela, esses processos estão em um mesmo processador. Já na programação distribuída, os processos estão alocados em diferentes máquinas conectadas por uma rede. Assim, é necessário haver suporte para disparar esses processos além de prover um meio de comunicação entre eles. Existem dois principais paradigmas de interação, o de memória compartilhada e o de troca de mensagens.

O paradigma de memória compartilhada é muito usado quando os processos (ou *threads*) estão executando em um mesmo núcleo ou computador, como mostra a figura 2. Esse paradigma permite que todos os processos acessem o mesmo espaço de memória. A vantagem desse paradigma são os processos observarem, praticamente em tempo real, as mudanças dos outros processos. A desvantagem ocorre quando os processos compartilham a memória e acontecem condições de corrida ou erros de sincronização. Nesta situação, o dado compartilhado se torna inconsistente.

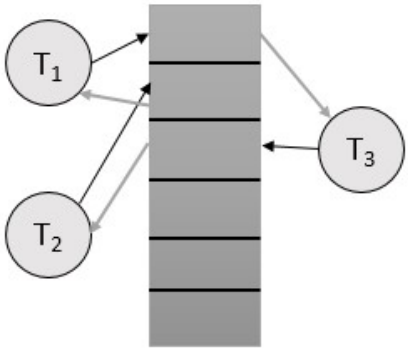
Figura 2 – Paradigma de memória compartilhada



O outro paradigma de comunicação é denominado troca de mensagens, e a comunicação ocorre através da troca explícita de mensagens entre processos paralelos e distribuídos, como mostra a figura 3. Esse paradigma permite a sincronização de processos em computadores diferentes, como em um *cluster* (conjunto de computadores interconectados através de cabos de rede). A principal vantagem desse paradigma é não haver condição de corrida para os dados compartilhados durante o processamento, pois são compartilhados através

de mensagens explícitas. Porém, a desvantagem desse paradigma é justamente controlar a comunicação explícita entre os processos, o que pode tornar o programa complexo.

Figura 3 – Paradigma da troca de mensagens



2.4.1 *Speedup*

Uma das formas de se avaliar a execução paralela de uma aplicação é utilizarmos a medida chamada *Speedup*, calculada a partir da fórmula em (2.3). Nela o $t_{sequencial}$ é o tempo de execução da versão sequencial da aplicação e $t_{paralelo}$ é o tempo de execução da versão paralela.

$$speedup = \frac{t_{sequencial}}{t_{paralelo}} \tag{2.3}$$

O *Speedup* expressa a relação de quantas vezes a aplicação paralela é mais rápida que a versão sequencial. Quando esse *Speedup* é igual a 1, isso significa que os dois programas terminaram no mesmo tempo. Desta forma, o paralelismo empregado não foi eficiente. Quando esse valor é menor que 1, significa que a aplicação paralela foi mais lenta que a versão da aplicação sequencial. Em geral, este fato ocorre quando há um grande *overhead* de comunicação, inerente ao paralelismo, independente do paradigma escolhido. O melhor cenário é o *Speedup* ser maior que 1, sendo o ideal próximo ao número de processadores utilizados na aplicação. Isso porque mostra que a aplicação paralela realmente conseguiu um ganho de tempo em relação ao programa sequencial. Além de obter uma grande eficiência.

Quando o *Speedup* da aplicação é maior que o número de processadores utilizados, é dito ser *Speedup* superlinear. Este *Speedup* ocorre em casos muito especiais. Em geral, é uma característica específica de algumas classes de aplicação.

3 MÉTODOS DIRETOS EM MATRIZES CHEIAS

Este capítulo apresenta alguns métodos diretos bastante conhecidos na literatura (GOLUB; LOAN, 1996; DEMMEL, 1997; WATKINS, 2002) aplicados para resolver sistemas lineares considerando matrizes cheias. Os métodos estudados são: Eliminação Gaussiana, Método de Gauss-Jordan, Decomposição LU, Decomposição LDU, Decomposição de Cholesky e Decomposição QR por ortogonalização Gram-Schmidt. Este capítulo mostra os resultados obtidos considerando os experimentos realizados com os dois tipos de alocações de memória existentes (estática e dinâmica). Além disso, considera a precisão de variáveis, analisando a precisão simples e a precisão dupla. Complementando esses experimentos, foram utilizadas diferentes *flags* de otimização do compilador, para se observar a sua influência na precisão do resultado e no tempo de execução de cada método.

3.1 Eliminação Gaussiana

O primeiro algoritmo de resolução de sistemas lineares abordado é a eliminação gaussiana (WATKINS, 2002). Esse algoritmo transforma a matriz na forma escalonada usando três operações elementares, que são: troca de linhas da matriz, multiplicação de uma linha por um escalar e soma de linhas da matriz. Por escalonar a matriz, a eliminação gaussiana também é conhecida como escalonamento.

Para realizar o escalonamento, é necessário percorrer cada linha da matriz e transformar o elemento da diagonal principal dessa linha i em pivô. Isto é, no último elemento não nulo da coluna. Para isso, percorre-se as linhas seguintes e usa-se a fórmula dada em (3.1), onde j é a linha que está sendo transformada, i é a linha e a coluna do pivô e k é a coluna que está sendo modificada.

$$a_{j,k} = a_{j,k} - \frac{a_{j,i}}{a_{i,i}} \times a_{i,k} \tag{3.1}$$

Supondo que o sistema linear tem a mesma quantidade de equações e variáveis, obtemos uma matriz A quadrada de ordem N. O Algoritmo 1 mostra como realizar o escalonamento. A variável M é a matriz aumentada do sistema, como apresentada na Definição 31.

Algoritmo 1 Algoritmo para escalonar uma matriz quadrada.

função FormaEscalonada(M)
1: **para** $i \leftarrow 1$ até N **faça**
2: **para** $j \leftarrow i + 1$ até N **faça**
3: **para** $k \leftarrow N + 1$ até i passo -1 **faça**
4: $m_{j,k} \leftarrow m_{j,k} - m_{i,k} \times \frac{m_{j,i}}{m_{i,i}}$

O escalonamento não é o único passo para a eliminação gaussiana. Após obter a matriz na forma triangular superior, os valores de x_i, $i = 1, \ldots, n$ precisam ser encontrados. Para

isso, é utilizado o método de retrosubstituição. Primeiro encontra-se o valor da variável x_n, depois o de x_{n-1}, até chegar no valor de x_1 usando a fórmula mostrada em (3.2). Este algoritmo é implementado diretamente pela fórmula e será apresentado somente a implementação na seção 3.7.

$$x_i = \frac{1}{a_{ii}} \times [b_i - \sum_{j=i+1}^{n_{linhas}} a_{ij} \times x_j] \tag{3.2}$$

Assim, a eliminação gaussiana, através desses dois algoritmos, pode resolver um sistema linear. Porém, nem todo sistema linear tem solução, como apresentado na Definição 30. Após a etapa de escalonamento podemos descobrir algumas situações em que a retrosubstituição não é bem sucedida. As situações que podem ser encontradas estão listadas na Tabela 1:

Tabela 1 – Situações que podem ocorrer na retrosubstituição.

Situação	Sistema
Linha de zeros (com $b = 0$)	Sistema com infinitas soluções
Linha de zeros (com $b \neq 0$)	Sistema impossível
Matriz A não quadrada ($m \neq n$)	Sistema com infinitas soluções

Antes de aplicar o método de retrosubstituição é necessário verificar se a matriz escalonada se encontra em algum desses casos.

3.2 **Método de Gauss-Jordan**

O algoritmo de Gauss-Jordan é derivado da eliminação gaussiana. Por isso, a função definida no Algoritmo 1 também é utilizada neste método. O Gauss-Jordan transforma a matriz A em uma matriz canônica, isto é, a transforma em uma matriz identidade. Como são necessários os valores de b_i, $i = 1, \ldots, m$ para resolver a equação, usa-se a matriz aumentada M para encontrar a forma canônica. Ao final do algoritmo, a matriz aumentada está na forma representada em (3.3). Com isso, a retrosubstituição no Método de Gauss-Jordan é direta, economizando tempo de processamento nesta segunda parte do algoritmo.

$$M = \left[\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & b_1 \\ 0 & 1 & \cdots & 0 & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & b_m \end{array}\right] \tag{3.3}$$

O pseudo-código para encontrar a forma canônica está descrito no Algoritmo 2.

O algoritmo de retrosubstituição, sem as verificações, é mais simples se comparado ao da eliminação gaussiana. Logo, o algoritmo de Gauss-Jordan também possui duas partes: uma para encontrar a forma canônica e outra para aplicar a retrosubstituição.

Algoritmo 2 Algoritmo para encontrar a forma canônica da matriz.

função FormaCanonica(M)
1: chamar Algoritmo 1
2: **para** $i \leftarrow N$ até 1 passo -1 **faça**
3: **para** $j \leftarrow i-1$ até 1 passo -1 **faça**
4: **para** $k \leftarrow N+1$ até i passo -1 **faça**
5: $m_{j,k} \leftarrow m_{j,k} - m_{i,k} \times \frac{m_{j,i}}{m_{i,i}}$
6: **para** $j \leftarrow N$ até i passo -1 **faça**
7: $m_{i,j} \leftarrow \frac{m_{i,j}}{m_{i,i}}$

3.3 Decomposição LU

A decomposição LU é realizada com matrizes quadradas invertíveis que podem ser reduzidas a forma triangular somente com a operação de soma de linhas, mostrada em (3.1). O nome dessa decomposição vem do inglês *Lower* e *Upper* (inferior e superior). Isso porque a matriz U é a matriz A reduzida a forma triangular superior e a matriz L é a matriz triangular inferior com os elementos $l_{l,j}$ sendo os coeficientes $\frac{a_{l,j}}{a_{i,j}}$ usados nas operações de soma de linha, com 1 na diagonal principal. Portanto, a matriz A pode ser decomposta em duas matrizes e o sistema representado por $LUx = B$, como pode ser visto em (3.4).

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 \end{bmatrix} \times \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{3.4}$$

O pseudo-código para achar as matrizes L e U está descrito no Algoritmo 3.

Algoritmo 3 Algoritmo para encontrar as matrizes L e U da Decomposição LU.

função DecomposicaoLU(A)
1: $U \leftarrow A$
2: **para** $i \leftarrow 1$ até N **faça**
3: **para** $j \leftarrow i+1$ até N **faça**
4: $l_{j,i} \leftarrow \frac{u_{j,i}}{u_{i,i}}$
5: **para** $k \leftarrow N+1$ até i passo -1 **faça**
6: $u_{j,k} \leftarrow u_{j,k} - u_{i,k} \times l_{j,i}$

Para resolver o sistema são necessárias duas etapas. Primeiro o cálculo de y em $Ly = B$ e segundo os valores de x em $Ux = y$. Para calcular x, usamos o mesmo algoritmo da retrosubstituição mostrado na eliminação gaussiana, usando a fórmula mostrada em (3.2). Porém, para calcular y, usamos o algoritmo de substituição, que calcula o y_1, depois y_2, e assim por diante até o y_n, usando a fórmula mostrada em (3.5).

$$y_i = \frac{1}{l_{ii}} \times [b_i - \sum_{j=1}^{i-1} l_{ij} \times y_j] \tag{3.5}$$

Como na eliminação gaussiana e no método de Gauss-Jordan, o pseudo-código da substituição é simples.

3.4 **Decomposição LDU**

A decomposição LDU é similar à decomposição LU. A matriz D é uma matriz diagonal com os valores da diagonal principal de U, que nesta decomposição passa a ter valor 1 (LIPSCHUTZ, 2000). Assim, a matriz A pode ser decomposta em três matrizes. O sistema, então, pode ser representado como $LDUx = B$, como mostrado em (3.6).

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 \end{bmatrix} \times \begin{bmatrix} d_{1,1} & 0 & \cdots & 0 \\ 0 & u_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_{n,n} \end{bmatrix} \times \begin{bmatrix} 1 & u_{1,2} & \cdots & u_{1,n} \\ 0 & 1 & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{3.6}$$

O pseudo-código para encontrar a decomposição LDU está descrito no Algoritmo 4.

Algoritmo 4 Algoritmo para achar a Decomposição LDU.

função DecomposicaoLDU(A)
1: $U \leftarrow A$
2: **para** $i \leftarrow 1$ até N **faça**
3: **para** $j \leftarrow i + 1$ até N **faça**
4: $l_{j,i} \leftarrow \frac{u_{j,i}}{u_{i,i}}$
5: **para** $k \leftarrow N + 1$ até i passo -1 **faça**
6: $u_{j,k} \leftarrow u_{j,k} - u_{i,k} \times l_{j,i}$
7: $d_{i,i} \leftarrow u_{i,i}$
8: **para** $j \leftarrow N - 1$ até i passo -1 **faça**
9: $u_{i,j} \leftarrow \frac{u_{i,j}}{u_{i,i}}$

Para resolver o sistema, com as três matrizes, são necessárias três etapas. Primeiro, calcular y em $Ly = B$. Depois, encontrar o vetor z em $Dz = y$. Por fim, calcular o vetor x em $Ux = z$. Para calcular y e x, utilizamos os mesmos algoritmos e fórmulas empregados na Decomposição LU. Para calcular o vetor z, a retrosubstituição (ou substituição) é direta, como no método de Gauss-Jordan.

3.5 **Decomposição de Cholesky**

A decomposição de Cholesky é a que possui mais restrições. A matriz A deve ser simétrica e definida positiva. Para calcular a decomposição de Cholesky, utilizamos as fórmulas apresentadas em (3.12). Para chegar a elas, partimos da decomposição LDU de A (3.7) e A^t (3.8). Como A é simétrica, $A = A^t$ e (3.7) é igual a (3.8). Como D é uma matriz diagonal temos $D = D^t$. Pela decomposição LDU ser única, temos que $L = U^t$ e $U = L^t$. Assim, $A = LDL^t = U^tDU$. Para continuarmos usamos $A = LDL^t$.

$$A = LDU \tag{3.7}$$

$$A^t = (LDU)^t = U^t D^t L^t \tag{3.8}$$

Como D é uma matriz diagonal, podemos separá-la em duas outras matrizes usando a raiz quadrada, como em $D = \sqrt{D}\sqrt{D}$. E por (3.9), chegamos nas matrizes G e G^t (3.10) que são a decomposição de Cholesky.

$$A = LDL^t = L\sqrt{D}\sqrt{D}L^t = L\sqrt{D}\sqrt{D}^t L^t = L\sqrt{D}(L\sqrt{D})^t = GG^t \tag{3.9}$$

$$G = \begin{bmatrix} g_{1,1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ g_{1,n} & \cdots & g_{n,n} \end{bmatrix} \text{ e } G^t = \begin{bmatrix} g_{1,1} & \cdots & g_{1,n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & g_{n,n} \end{bmatrix} \tag{3.10}$$

Expandindo a igualdade $A = GG^t$, como em (3.11), chegamos as fórmulas descritas em (3.12). O algoritmo para calcular as matrizes G e G^t é simples. Sua implementação pode ser obtida diretamente das fórmulas descritas.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} = \begin{bmatrix} g_{1,1}^2 & g_{1,1}g_{1,2} & \cdots & g_{1,1}g_{1,n} \\ g_{1,1}g_{1,2} & g_{1,2}^2 + g_{2,2}^2 & \cdots & g_{1,1}g_{1,n} + g_{2,2}g_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{1,1}g_{1,n} & g_{1,2}g_{1,n} + g_{2,2}g_{2,n} & \cdots & \sum_{k=1}^{n} g_{k,n}^2 \end{bmatrix} \tag{3.11}$$

$$\begin{aligned} g_{i,i} &= \sqrt{a_{i,i} - \sum_{k=1}^{i-1} g_{k,i}^2}, \text{ para } i = 1 \ , \ \ldots \ , \ n \\ g_{j,i} &= \frac{a_{i,j} - \sum_{k=1}^{i-1} g_{k,j}g_{k,i}}{g_{i,i}}, \text{ para } j = i+1 \ , \ \ldots \ , \ n \end{aligned} \tag{3.12}$$

Para resolver o sistema $GG^t x = B$ são necessárias duas etapas. Calcula-se y em $Gy = B$, usamos o algoritmo da substituição com a fórmula dada em (3.5). Depois encontramos x em $G^t x = y$, usamos o algoritmo da retrosubstituição com a fórmula em (3.2).

3.6 **Decomposição QR por ortogonalização de Gram-Schmidt**

Para a decomposição QR, a matriz A deve ser invertível, porque precisa ter colunas linearmente independentes. O resultado dessa decomposição é dado por $A = QR$, onde Q é uma matriz ortogonal e R é uma matriz triangular superior.

Para chegar na matriz Q ortogonal, usamos o processo de ortogonalização de Gram-Schmidt nas colunas de A. Este processo transforma as colunas de A em vetores ortogonais, para depois normalizá-los. Assim, cada coluna é dividida pela sua norma.

O processo de ortogonalização de Gram-Schmidt é descrito em (3.13), sendo $w_i, i = 1, \ldots, n$ os vetores ortogonalizados e $v_i, i = 1, \ldots, n$ os vetores originais. A fórmula para $\langle u, v \rangle$ (produto interno de dois vetores) está descrita na Definição 1.

$$
\begin{aligned}
w_1 &= v_1 \\
w_2 &= v_2 - \frac{\langle v_2, w_1 \rangle}{\langle w_1, w_1 \rangle} \times w_1 \\
w_3 &= v_3 - \frac{\langle v_3, w_1 \rangle}{\langle w_1, w_1 \rangle} \times w_1 - \frac{\langle v_3, w_2 \rangle}{\langle w_2, w_2 \rangle} \times w_2 \\
&\vdots \\
w_n &= v_n - \frac{\langle v_n, w_1 \rangle}{\langle w_1, w_1 \rangle} \times w_1 - \frac{\langle v_n, w_2 \rangle}{\langle w_2, w_2 \rangle} \times w_2 - \ldots - \frac{\langle v_n, w_{n-1} \rangle}{\langle w_{n-1}, w_{n-1} \rangle} \times w_{n-1}
\end{aligned} \tag{3.13}
$$

A matriz R armazena os coeficientes do processo de Gram-Schmidt no índice correspondente. Eles são chamados de coeficientes de Fourier. Por exemplo, o coeficiente $\frac{\langle v_2, w_1 \rangle}{\langle w_1, w_1 \rangle}$ é armazenado no índice $r_{1,2}$ da matriz R.

O pseudo-código para calcular a decomposição QR está descrito no Algoritmo 5.

Algoritmo 5 Algoritmo para achar a Decomposição QR por ortogonalização Gram-Schmidt.

função DecomposicaoQR(A)
1: $Q \leftarrow A$
2: **para** $i \leftarrow 1$ até N **faça**
3: **para** $j \leftarrow i - 1$ até 1 passo -1 **faça**
4: $r_{j,i} \leftarrow \frac{\langle Q_i, Q_j \rangle}{\langle Q_j, Q_j \rangle}$
5: **para** $k \leftarrow N + 1$ até i passo -1 **faça**
6: $q_{i,k} \leftarrow q_{i,k} - q_{j,k} \times r_{j,i}$
7: $r_{i,i} \leftarrow \sqrt{\langle Q_i, Q_i \rangle}$
8: **para** $k \leftarrow 1$ até N **faça**
9: $q_{i,k} \leftarrow \frac{q_{i,k}}{r_{i,i}}$

Neste algoritmo, as variáveis Q_i representam a coluna i da matriz Q, enquanto que as variáveis $q_{i,j}$ e $r_{i,j}$ representam o elemento da linha i e da coluna j da matriz Q e R, respectivamente. Além disso, $\langle u, v \rangle$ representa o produto interno dos vetores u e v.

Para resolver o sistema $QRx = B$ é necessário manipulá-lo para obter uma algoritmo computacional mais simples de ser resolvido. Dessa forma, multiplicamos por Q^{-1} os dois lados (3.14) da igualdade. Assim, obtemos a matriz triangular R que é diretamente solucionada computacionalmente. Como Q é ortogonal, existe a relação $Q^{-1} = Q^t$ (LIPSCHUTZ, 2000). Logo, podemos usar a transposta de Q multiplicada pelo vetor B para encontrar o vetor B'. Este vetor é usado para calcular o vetor x, em $Rx = B'$, usando a retrosubstituição com a fórmula descrita em (3.2).

$$
QRx = B \Leftrightarrow Q^{-1}QRx = Q^{-1}B \Leftrightarrow IRx = Q^{-1}B \Leftrightarrow Rx = Q^{-1}B \tag{3.14}
$$

Todos os algoritmos foram implementados em Linguagem C. A seguir, abordamos cada uma dessas implementações.

3.7 Implementação do Algoritmo de Eliminação Gaussiana

Para a eliminação gaussiana foram criadas duas funções em linguagem C. Uma para escalonar a matriz e outra para o método de retrosubstituição. O código para escalonar a matriz foi encontrado no livro "Algoritmos em C" (SEDGEWICK, 1990). Porém, ele utiliza alocação estática. Então, o modificamos para usar alocação dinâmica. Além dessa alteração, foram incluídos alguns testes para escalonar matrizes não quadradas para garantir a generalidade do código.

A implementação do escalonamento, primeira função da eliminação gaussiana, é apresentada no Algoritmo 6.

Algoritmo 6 Função para obter a matriz escalonada.

```
    função FormaEscalonada(Ponteiro: pMat)
 1: line ← 0
 2: para l ← 0 até tam faça
 3:     index ← line
 4:     para k ← line até tam passo −1 faça
 5:         se abs(pMat[k][l]) > 0 então
 6:             index ← k
 7:             break;
 8:     se pMat[index][l] == 0 então
 9:         continue;
10:     se index ≠ line então
11:         para i ← 0 até tam faça
12:             aux ← pMat[index][i]
13:             pMat[index][i] ← pMat[line][i]
14:             pMat[index][i] ← aux
15:     para i ← line + 1 até tam faça
16:         para j ← tam até l passo −1 faça
17:             pMat[i][j] ← pMat[i][j] − (pMat[i][l] ∗ pMat[line][j]/pMat[line][l])
18:     line ← line + 1
19:     se line ≥ tam então
20:         break;
```

O *loop* externo itera pelas colunas e a variável *line* indica a linha atual de execução. O primeiro *loop* aninhado, com a variável k, procura pela primeira linha, a partir da atual, que tem elemento não nulo na coluna c, para trocar de posição com a linha atual, caso seja necessária essa troca. A parte do pseudo-código apresentado na Seção 3.1 está nos dois *loop* aninhados no final do código apresentado.

A parte da retrosubstituição foi implementada como no Algoritmo 7. Os testes para as situações citadas na Seção 3.1 foram ocultados. Quando o programa chega nesse *loop* apresentado, já é garantido que a matriz A de coeficientes é quadrada e o valor do vetor de valores livres b_i, $i = 1, \ldots, m$ está na última posição da matriz *mat*.

Esses dois trechos de código são os principais para a eliminação gaussiana, e os que consomem mais tempo de execução.

A complexidade do escalonamento é $O(MN^2)$ e a complexidade da retrosubstituição é $O(N^2)$. Logo, a complexidade da eliminação gaussiana total para qualquer matriz é dada

Algoritmo 7 Função da retrosubstituição da Eliminação Gaussiana.

função acharValores(**Ponteiro:** mat; **Ponteiro de ponteiro:** var)
1: /* testar se o sistema é possível e tem solução definida */
2: **para** $i \leftarrow tamm - 1$ até 0 passo -1 **faça**
3: $(*var)[i] \leftarrow mat[i][tam]$
4: **para** $j \leftarrow i + 1$ até tam **faça**
5: $(*var)[i] \leftarrow (*var)[i] - (mat[i][j] * (*var)[j])$
6: $(*var)[i] \leftarrow (*var)[i]/mat[i][i]$

por (3.15). Quando a matriz é quadrada, a complexidade se torna $O(N^3)$.

$$O(MN^2) + O(N^2) = O(MN^2) \tag{3.15}$$

3.8 Implementação do Algoritmo de Gauss-Jordan

As funções que compõem o método de Gauss-Jordan são duas: a de encontrar a forma canônica e a retrosubstituição. A função para encontrar a forma canônica está implementada como no Algoritmo 8.

Algoritmo 8 Função para encontrar a matriz canônica.

função acharFormaCanonica(**Ponteiro:** $pMat$; **Inteiro:** tam)
1: acharFormaEscalonada(pMat)
2: **para** $l \leftarrow tam - 1$ até 0 passo -1 **faça**
3: **para** $i \leftarrow l - 1$ até 0 passo -1 **faça**
4: $pMat[i][tam] \leftarrow pMat[i][tam] - (pMat[i][l] * pMat[l][tam]/pMat[l][l])$
5: **para** $j \leftarrow l$ até 0 passo -1 **faça**
6: $pMat[i][j] \leftarrow pMat[i][j] - (pMat[i][l] * pMat[l][j]/pMat[l][l])$
7: **para** $i \leftarrow tam$ até l passo -1 **faça**
8: $pMat[l][i] \leftarrow pMat[l][i]/pMat[l][l]$

Note que primeiro é necessário chamar a função para a transformar a matriz em triangular superior, como mostrado no algoritmo de eliminação gaussiana. O *loop* externo itera pelas linhas e o primeiro *loop* aninhado, que itera na variável k, procura em qual coluna está o pivô dessa linha. Depois, os dois *loop* aninhados fazem a transformação das linhas acima e o último *loop* transforma o pivô em 1, modificando a linha inteira.

Como a matriz canônica está na forma (3.3), a retrosubstituição de Gauss-Jordan é mais direta e a implementação é apresentada no Algoritmo 9. Assim como na retrosubstituição da eliminação gaussiana, as verificações para as situações explicitadas anteriormente foram ocultadas. Ao se chegar neste *loop* a matriz é garantidamente quadrada.

Estas são as duas partes principais do método de Gauss-Jordan. A complexidade da segunda parte é $O(N)$, uma vez que o resultado é calculado direto na matriz. A complexidade da função para encontrar a forma canônica é a soma da função de escalonamento mais a complexidade do *loop* descrito anteriormente. A complexidade do *loop* é $O(M^2N)$ e a complexidade do escalonamento é $O(MN^2)$. Logo, a complexidade da primeira parte de Gauss-Jordan é dada por $O(max(M^2N, MN^2))$. Portanto, a complexidade de Gauss-Jordan é dada por (3.16). Quando a matriz é quadrada, a complexidade se torna $O(N^3)$.

Algoritmo 9 Função da retrosubstituição do Método de Gauss-Jordan.

função acharValores(**Ponteiro:** mat; **Ponteiro de ponteiro:** var; **Inteiro:** tam)
1: /* testar se o sistema é possível e tem solução definida */
2: **para** $i \leftarrow tam - 1$ até 0 passo -1 **faça**
3: $(*var)[i] \leftarrow mat[i][tam]$

$$O(max(M^2N, MN^2)) + O(N) = O(max(M^2N, MN^2)) \tag{3.16}$$

3.9 Implementação do Algoritmo de Decomposição LU

O algoritmo implementado para a decomposição LU em si está descrito no Algoritmo 10.

Algoritmo 10 Função para achar as matrizes L e U da Decomposição LU

função acharDecomposicaoLU(**Ponteiro:** $pMat$, pL; **Inteiro:** tam)
1: **para** $l \leftarrow 0$ até tam **faça**
2: **para** $i \leftarrow l + 1$ até tam **faça**
3: $coeficiente \leftarrow pMat[i][l]/pMat[l][l]$
4: $pL[i][l] \leftarrow coeficiente$
5: **para** $j \leftarrow tam - 1$ até l passo -1 **faça**
6: $pMat[i][j] \leftarrow pMat[i][j] - (pMat[l][j] * coeficiente)$

A única diferença para o pseudo-código mostrado na Seção 3.3 foi a criação de uma variável (chamada $coeficiente$) para armazenar o índice da matriz L, visando um acesso mais rápido, evitando o duplo acesso na memória da matriz.

Como a matriz L é triangular inferior, implementamos a substituição como no Algoritmo 11. Para achar os valores pela matriz U, utilizamos o mesmo algoritmo de retrosubstituição descrito no Algoritmo 7, sem os testes sobre o sistema linear.

Algoritmo 11 Função da substituição da Decomposição LU

função acharValoresL(**Ponteiro:** mat; **Ponteiro de ponteiro:** var; **Inteiro:** tam)
1: **para** $i \leftarrow 0$ até tam **faça**
2: $(*var)[i] \leftarrow mat[i][tam]$
3: **para** $j \leftarrow i - 1$ até 0 passo -1 **faça**
4: $(*var)[i] \leftarrow (*var)[i] - (mat[i][j] * (*var)[j])$
5: $(*var)[i] \leftarrow (*var)[i]/mat[i][i]$

Para calcular a complexidade da decomposição LU, precisamos somar a complexidade dos três trechos de código citados anteriormente. A complexidade para encontar as matrizes L e U tem complexidade $O(N^3)$. A complexidade da retrosubstituição e da substituição é a mesma, e igual a $O(N^2)$. Assim, a complexidade da Decomposição LU é dada por (3.17).

$$O(N^3) + O(N^2) + O(N^2) = O(N^3) \tag{3.17}$$

3.10 Implementação do Algoritmo de Decomposição LDU

O algoritmo implementado para a decomposição LDU em si está descrito no Algoritmo 12.

Algoritmo 12 Função para encontrar a Decomposição LDU

função acharDecomposicaoLDU(**Ponteiro:** $pMat$, pL, pD; **Inteiro:** tam)
1: **para** $l \leftarrow 0$ até tam **faça**
2: **para** $i \leftarrow l + 1$ até tam **faça**
3: $coeficiente \leftarrow pMat[i][l]/pMat[l][l]$
4: $pL[i][l] \leftarrow coeficiente$
5: **para** $j \leftarrow tam - 1$ até l passo -1 **faça**
6: $pMat[i][j] \leftarrow pMat[i][j] - (pMat[l][j] * coeficiente)$
7: $pD[l][l] \leftarrow pMat[l][l]$
8: **para** $j \leftarrow tam - 1$ até l passo -1 **faça**
9: $pMat[l][j] \leftarrow pMat[l][j]/pMat[l][l]$

A única diferença para o pseudo-código mostrado na Seção 3.4 foi a criação de uma variável ($coeficiente$) para armazenar o índice da matriz L, para ter um acesso mais rápido, evitando o duplo acesso na memória da matriz.

Para resolver o sistema, utilizamos a mesma implementação da substituição descrita no Algoritmo 7 para a matriz L e a mesma implementação da retrosubstituição descrita no Algoritmo 11 para a matriz U. Para a matriz D, por ela ser diagonal, a substituição é direta e foi implementada como no Algoritmo 13.

Algoritmo 13 Função para achar os valores da matriz D da Decomposição LDU.

função acharValoresD(**Ponteiro:** mat; **Ponteiro de ponteiro:** var; **Inteiro:** tam)
1: **para** $i \leftarrow tamm - 1$ até 0 passo -1 **faça**
2: $(*var)[i] \leftarrow mat[i][tam]/mat[i][i]$

Para calcular a complexidade da decomposição LU, precisamos somar a complexidade dos quatro trechos de código citados anteriormente. A complexidade para encontrar as matrizes L, D e U tem complexidade $O(N^3)$. A complexidade da retrosubstituição em U e da substituição em L é a mesma, $O(N^2)$ e a complexidade para calcular os valores em D é $O(N)$, por ser uma matriz diagonal. Assim, a complexidade da Decomposição LDU é dada por (3.18).

$$O(N^3) + O(N^2) + O(N^2) + O(N) = O(N^3) \tag{3.18}$$

3.11 Implementação do Algoritmo de Decomposição de Cholesky

A implementação da decomposição de Cholesky (segundo as fórmulas de 3.12), está descrita no Algoritmo 14.

Os *loops* relacionados as variáveis i e j são referentes aos índices nas fórmulas (3.12). Primeiro calculamos os $g_{i,j}$ para depois calcularmos os $g_{i,i}$. Os dois *loops* relacionados a variável k são usados para calcular os somatórios das fórmulas.

Algoritmo 14 Decomposição Cholesky (versão de implementação).

função acharDecomposicaoCholesky(**Ponteiro:** $pMat$, pG, pGt; **Inteiro:** tam)
1: **para** $i \leftarrow 0$ até tam **faça**
2: **para** $j \leftarrow 0$ até i **faça**
3: $soma \leftarrow 0$
4: **para** $k \leftarrow 0$ até j **faça**
5: $soma \leftarrow soma + (pG[i][k] * pG[j][k])$
6: $pG[i][j] \leftarrow (pMat[i][j] - soma)/pG[j][j]$
7: $pGt[j][i] \leftarrow pG[i][j]$
8: $soma \leftarrow 0$
9: **para** $k \leftarrow 0$ até j **faça**
10: $soma \leftarrow pG[i][k] * pG[i][k]$
11: $pG[i][i] \leftarrow (\text{sqrt}(pMat[i][i] - soma)$
12: $pGt[i][i] \leftarrow pG[i][i]$

Para resolver o sistema linear, usamos o algoritmo de substituição mostrado no Algoritmo 11 na matriz G e o algoritmo de retrosubstituição mostrado no Algoritmo 7 na matriz G^t.

Os três trechos de código citados anteriormente são os principais para a decomposição de Cholesky. A função para implementar a decomposição de Cholesky tem complexidade de $N^3/6$ operações, que é representada por $O(N^3)$. A complexidade da retrosubstituição e da substituição é a mesma e igual a $O(N^2)$. Assim, a complexidade da Decomposição de Cholesky é dada por (3.19).

$$O(N^3) + O(N^2) + O(N^2) = O(N^3) \tag{3.19}$$

3.12 Implementação do Algoritmo de Decomposição QR por ortogonalização Gram-Schmidt

Para o método de Decomposição QR por ortogonalização Gram-Schmidt, a função do produto interno foi implementada e seu código é apresentado no Algoritmo 15. Ela considera diretamente a equação do produto interno da Definição 1.

Algoritmo 15 Produto interno entre dois vetores.

função produtoInterno(**Vetores:** $vec1$, $vec2$; **Inteiro:** tam)
1: $soma \leftarrow 0$
2: **para** $i \leftarrow 0$ até tam **faça**
3: $soma \leftarrow soma + (vec1[i] * vec2[i])$
4: **retorna** $soma$

O algoritmo principal da Decomposição QR foi implementado como no Algoritmo 16.

A única diferença para o pseudo-código apresentado na Seção 3.6 é a criação das variáveis $fourier$ e $norma$, para armazenar os índices de R em uso e agilizar o acesso a eles.

Para resolver o sistema, primeiro multiplicamos a matriz Q^t por B. Essa multiplicação está descrita no Algoritmo 17.

Algoritmo 16 Decomposição QR por ortogonalização Gram-Schmidt (versão de implementação).

função acharDecomposicaoQR(**Ponteiro:** pQ, pR; **Inteiro:** tam)
1: **para** $j \leftarrow 0$ até tam **faça**
2: **para** $i \leftarrow j - 1$ até 0 passo -1 **faça**
3: $fourier \leftarrow$ produtoInterno$(pQ[j], pQ[i], tam)$/produtoInterno$(pQ[i], pQ[i], tam)$
4: $pR[i][j] \leftarrow fourier$
5: **para** $k \leftarrow 0$ até tam **faça**
6: $pQ[j][k] \leftarrow pQ[j][k] - fourier * pQ[i][k]$
7: $norma \leftarrow$ sqrt(produtoInterno$(pQ[j], pq[j], tam)$)
8: $pR[j][j] \leftarrow norma$
9: **para** $k \leftarrow 0$ até tam **faça**
10: $pQ[[j][k] \leftarrow pQ[j][k]/norma$

Algoritmo 17 Multiplicação matriz Q pelo vetor $\vec{b}$.

função multMatrizQVetorB(**Ponteiro:** Q; **Ponteiro de ponteiro:** pB)
1: **para** $i \leftarrow 0$ até tam **faça**
2: $result[i] \leftarrow 0$
3: **para** $k \leftarrow 0$ até tam **faça**
4: $result[i] \leftarrow result[i] + (Q[i][k] * (*pB)[k])$
5: **para** $i \leftarrow 0$ até tam **faça**
6: $(*pB)[i] \leftarrow result[i]$

A variável *result* foi criada para armazenar os valores temporariamente, uma vez que é necessário o vetor B para todas as iterações da multiplicação.

Para encontrar x, usamos o algoritmo de retrosubstituição do Algoritmo 7 na matriz R.

Para calcular a complexidade da Decomposição QR, somamos a complexidade dos quatro trechos de código citados anteriormente. A função para encontrar as matrizes Q e R tem complexidade $O(N^3)$ e a complexidade da função que calcula o produto interno é $O(N)$. Porém, por ser chamada somente dentro da função principal, não é preciso contabilizar na soma final da complexidade. A complexidade da multiplicação de Q^t por B é $O(N^2)$. A complexidade da retrosubstituição também é $O(N^2)$. Assim, a complexidade da Decomposição QR é dada por (3.20).

$$O(N^3) + O(N^2) + O(N^2) = O(N^3) \tag{3.20}$$

3.13 Influência da alocação de memória

Neste estudo, verificamos a influência do tipo de alocação de memória nos métodos abordados anteriormente. Cada algoritmo foi implementado com duas versões, uma com alocação estática e outra com alocação dinâmica de memória. Na alocação estática determinamos o tamanho das matrizes em tempo de compilação. A alocação dinâmica permite maior liberdade ao usuário, de acordo com (ASCENCIO; CAMPOS, 2008). Isso porque o tamanho das matrizes ou vetores é determinado em tempo de execução. Além disso, é

possível utilizar ponteiros para manipular as matrizes ou vetores.

Todos os algoritmos percorrem uma grande parte dos elementos da matriz para resolver o sistema linear. Na versão com alocação dinâmica utilizamos sempre a aritmética de ponteiros. Ela permite calcular o endereço de memória dos elementos das matrizes, diminuindo assim a quantidade de acessos a memória em relação a alocação estática, por isso todos os métodos acabam se beneficiando da alocação dinâmica em relação a estática.

Porém, existem outros benefícios da alocação dinâmica em alguns algoritmos. Um dos algoritmos que mais se beneficia dessa aritmética de ponteiros é a Decomposição de Cholesky. Neste algoritmo, é necessário acessar, além da matriz original, os elementos anteriormente calculados da matriz resultante. Outros algoritmos que se beneficiam dessa aritmética são a Decomposição LU e a LDU. Os dois algoritmos percorrem a matriz original para calcular as matrizes triangulares superior e inferior.

A alocação dinâmica combinada a aritmética de ponteiros contribui, por mais um motivo, em dois outros métodos: Eliminação Gaussiana e Gauss-Jordan. Nesses algoritmos é preciso realizar trocas entre linhas ou colunas na matriz, devido ao escalonamento. Essa troca é mais custosa computacionalmente na alocação estática, uma vez que precisamos percorrer e trocar cada elemento das linhas (ou colunas). Com a alocação dinâmica, basta simplesmente trocar os ponteiros que representam as linhas (ou colunas).

No método de Decomposição QR por ortogonalização Gram-Schmidt, o produto interno entre as colunas da matriz é implementado como uma função a parte. Essa função é chamada em diversos momentos do algoritmo da ortogonalização. Os parâmetros do produto interno são dois ponteiros para vetor, independente do tipo da alocação. Assim, com uma pequena modificação no armazenamento da matriz original, em ambas alocações (estática e dinâmica), as colunas podem ser facilmente acessadas. A modificação foi armazenar a matriz original como a sua transposta.

3.14 **Influência da precisão das variáveis**

Neste estudo, verificamos a influência da precisão escolhida para as variáveis criadas. A precisão das variáveis define quantas casas decimais são usadas para representar computacionalmente os valores numéricos. Quanto mais casas decimais pudermos utilizar, mais preciso são os resultados e os erros numéricos vindos do processo de truncamento são reduzidos. Além disso, dados iniciais bem definidos em relação a sua precisão podem não ser bem armazenados no computador, uma vez que ele utiliza a base binária para a sua representação e suas operações.

Para representar números reais, utilizamos as chamadas variáveis de ponto flutuante. Normalmente, uma variável de ponto flutuante de precisão simples utiliza uma palavra de memória (4 *bytes* nos processadores de 32 *bits*). Uma variável de ponto flutuante de precisão dupla utiliza duas palavras de memória (8 *bytes* ou 64 *bits*, nos processadores de 32 *bits*) para armazenar os dígitos (PATTERSON; HENNESSY, 2014). Como as variáveis de precisão dupla utilizam o dobro de palavras da memória, é esperado que o tempo de execução seja maior, porque existe maior tempo de espera nas operações de leitura ou escrita da memória. Porém, a maioria dos computadores hoje utilizam sistemas operacionais com 64 bits de tamanho e com isso, as variáveis de precisão simples utilizam meia palavra da memória.

Os tipos de variáveis de ponto flutuante da linguagem C são denominados de *float*, de precisão simples, e *double*, de precisão dupla.

3.15 **Influência das *flags* de compilação**

A linguagem C foi escolhida por possuir tipagem estática, o que facilita esse estudo. Na tipagem estática a linguagem define o tipo da variável no tempo de compilação. Isto é, quando o código é analisado e transformado em código binário por um compilador (ASCENCIO; CAMPOS, 2008) já se conhece a quantidade de bytes usadas para armazenar a variável.

O artigo de (BOTOR; HABIBALLA, 2018) compara três compiladores: dois utilizados somente para a linguagem C e o terceiro utilizado para a linguagem C e a C++. Em suas conclusões afirmam que o compilador GCC (*the GNU Compiler Collection*), um compilador de licença gratuita, é um compilador melhor que o *Visual Studio* da Microsoft. Portanto, para realizar esse estudo utilizamos duas versões do compilador GCC: a versão 5.4.0 (GCC, 2016) e a versão 7.1.0 (GCC, 2017). A primeira versão foi lançada em junho de 2016 e é a versão já instalada no Ubuntu versão 16.04. A última versão foi lançada em maio de 2017 e é uma das últimas versões estáveis do GCC para o Ubuntu.

O compilador GCC oferece algumas opções para otimizar o código que é gerado. Essas opções são chamadas de *flags* do compilador. Existem oito delas usadas para otimização do código: -O0, -O, -O1, -O2, -O3, -Ofast, -Og e -Os. Cada uma dessas *flags* gera um código executável com um nível diferente de otimização. Em (HOSTE; EECKHOUT, 2008) é apresentado um algoritmo que escolhe automaticamente as opções de compilação para os códigos gerados. Porém, este algoritmo é iterativo e aumenta o tempo de compilação total. Além disso, o artigo não apresenta a qualidade dos resultados obtidos. A seguir, explicamos brevemente cada *flag* para as duas versões do compilador. Uma descrição mais detalhada pode ser encontrada em (GCC, 2016) e em (GCC, 2017).

3.15.1 *Flags* na versão 5.4.0 do GCC

A *flag* de otimização padrão é a -O0, que reduz o tempo de compilação e permite que o modo de depuração apresente os resultados esperados. O próximo nível de otimização é alcançado com as *flags* -O e -O1, que geram as mesmas otimizações no código. Neste nível, o compilador minimiza o tamanho do código e o tempo de execução, mas sem usar otimizações que demorem para serem feitas.

Continuando nos níveis de otimização, a *flag* -O2 ativa as otimizações do nível -O1 e quase todas as opções de otimização que não diminuem o tempo de compilação, aumentando o espaço ocupado pelo processo de compilação, além de poder aumentar o tamanho final do código gerado. Um exemplo de otimização deste nível é alinhar na memória o começo de uma função com uma potência de 2 maior que um determinado número, geralmente dependente da máquina na qual será executado o código. Isto é feito para se encontrar a função de forma mais rápida na memória. O próximo nível de otimização é dado pela *flag* -O3. Neste nível, todas as otimizações realizadas nos níveis -O2 e -O1 são ativadas, além das que aumentam o tamanho do código final, buscando reduzir o tempo de execução. Com a *flag* -O3, por exemplo, o compilador insere o próprio código de uma função em todas as suas chamadas, fazendo o que é chamado de *inline functions*. Estas *inline functions* são boas pois evitam um desvio na execução do código final, diminuindo o tempo de busca da instrução.

O último nível de otimização é o da *flag* -Ofast, também conhecida como *fast-math*. Além de habilitar todas as otimizações do nível -O3, a *flag* -Ofast permite algumas otimizações que não são válidas para a maioria dos programas com restrições de qualidade. Por exemplo, essa *flag* habilita algumas otimizações matemáticas que podem levar a um

resultado incorreto em programas e funções que dependem de resultados dentro do padrão IEEE para funções matemáticas.

As últimas duas *flags*, -Os e -Og, são focadas em otimizar o tamanho do código final e a depuração do código, respectivamente. A *flag* -Os habilita todas as otimizações da *flag* -O2 que não aumentam o tamanho do código, além de habilitar algumas otimizações específicas para diminuir o tamanho do código binário. Por outro lado, a *flag* -Og só habilita as otimizações que não interferem no processo de depuração do código.

3.15.2 *Flags* na versão 7.1.0 do GCC

Todas as *flags* na versão 7.1.0 tem as mesmas opções de otimização que as da versão 5.4.0 com algumas poucas novidades. No nível da *flag* -O (ou da -O1), uma nova opção de otimização é a reordenação dos blocos de instruções para evitar desvios e otimizar a localidade do código.

O nível da *flag* -O2 tem novas otimizações além das antigas. Nesta versão do GCC, o compilador procura por operações de escritas na memória que sejam menores do que uma palavra, para poder juntá-las em uma única operação de escrita. No nível da *flag* -O3, uma das novas otimizações da versão 7.1.0 do GCC, é o desenrolar de *loops* que não tenham muitas iterações, dessa forma pode-se reduzir o tempo gasto no controle do fluxo do programa.

As outras três *flags*, -Ofast, -Os e -Og, não tem otimizações novas nesta versão do GCC. Elas continuam com as mesmas otimizações e finalidades da versão 5.4.0 do GCC.

3.16 **Experimentos numéricos e análise dos resultados**

Todos os experimentos foram realizados em um computador com processador Intel Core i7 de 2,80 GHz com 4 núcleos, memória principal de 8GB e memória *cache* de 8MB. As matrizes de teste utilizadas foram matrizes diagonalmente dominante sintéticas, que pela definição também são definidas positivas. Essas matrizes foram obtidas através da fórmula mostrada em (3.21), onde C é uma matriz quadrada de números aleatórios entre 0 e 999, n é a ordem da matriz e I_n é a matriz identidade de ordem n. Estas matrizes de teste sintéticas são compostas por números reais de precisão simples.

$$A = C \times C^t + n \times I_n \tag{3.21}$$

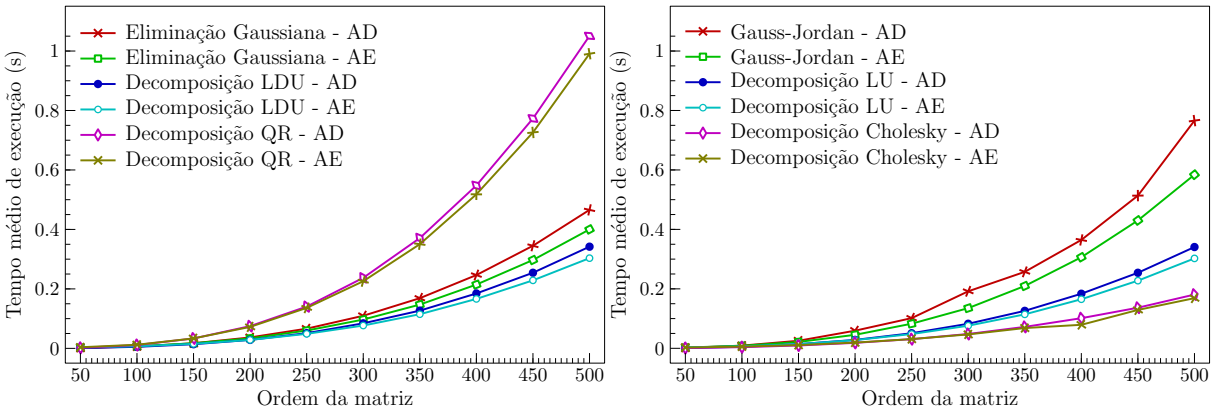
Para ter um maior controle da qualidade dos resultados, forçamos o sistema linear de teste a ter a solução trivial, o vetor unitário, como resposta. Para isso, multiplicamos essas matrizes criadas pelo vetor com valor 1 em todas as posições para encontrar o vetor dos valores independentes (vetor B).

3.16.1 Influência da alocação de memória

Neste estudo, o sistema operacional usado foi o Ubuntu versão 14.04 e por isso, a versão do GCC foi outra das apresentadas na seção anterior. Cada método foi compilado com o GCC versão 4.8.4 e executado 10 vezes. Os resultados apresentaram um desvio padrão desprezível (menor que 5% da média). Como dados de entrada foram utilizadas as matrizes de teste definidas pela fórmula (3.21) onde a ordem varia de 50 até 500, com acréscimo de 50 por teste.

Os resultados obtidos não mostraram a alocação dinâmica com menores tempos de execução. Os tempos de execução foram muito similares entre os dois tipos de alocação, como pode ser visto na Figura 4.

Figura 4 – Gráfico com os tempos de execução de todos os métodos



A medida que o tamanho da matriz aumenta, a alocação estática de memória mostra um tempo de execução ligeiramente menor para alguns métodos. Este fato ocorreu na Eliminação Gaussiana, no Método de Gauss-Jordan, na Decomposição LU e na LDU. Nas Decomposições de Cholesky e QR, o desempenho das duas alocações foi similar. Isso pode ocorrer pelo fato de estarmos contabilizando o tempo de alocação da memória da alocação dinâmica no tempo total de execução. A alocação estática é mais rápida, porque a alocação foi feita antes de contabilizar o tempo.

3.16.2 Influência da precisão de variáveis (*Float* vs. *Double*)

Este estudo foi realizado em duas etapas no mesmo sistema operacional Ubuntu, porém com versões diferentes. A primeira etapa foi realizada com a versão 14.04 e a segunda com a versão 16.04. Todos os experimentos utilizaram matrizes quadradas sintéticas, encontradas com a mesma fórmula apresentada em (3.21), com a solução trivial como resposta.

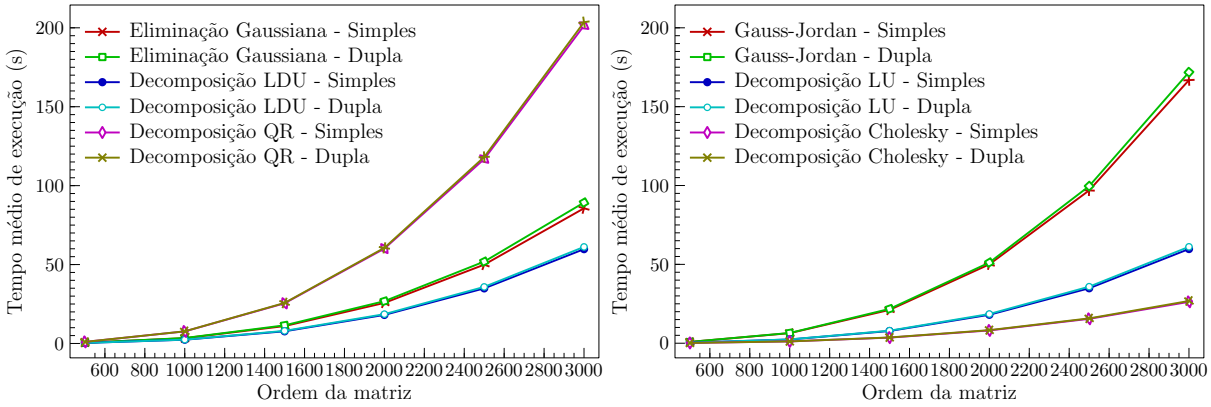
Na primeira etapa do estudo, os métodos foram compilados com GCC versão 4.8.4. E na segunda etapa os métodos foram compilados com o GCC versão 5.4.0, com a *flag* -O3. Todos os métodos foram executados pelo menos 10 vezes e os tempos apresentados são a média dessas execuções. O desvio padrão obtido foi desprezível, menor do que 1% da média.

A Figura 5 mostra a média do tempo de execução de cada método com relação as duas precisões utilizadas na declaração das variáveis, a precisão simples e a dupla. As matrizes utilizadas nesta primeira etapa variam da ordem 500 a 3000, com acréscimos de 500 entre duas matrizes.

Em todos os métodos, os tempos de execução para precisão simples ou dupla foi similar para as matrizes menores. No entanto, essa diferença cresceu um pouco a medida que o tamanho da massa de dados de entrada crescia.

Para conferir a qualidade do resultado obtido, Utilizamos a solução final de cada método, para a matriz de ordem 2000, e calculamos a norma euclidiana do erro. Para isso empregamos a fórmula (3.22), onde N é o tamanho da matriz (neste caso 2000), x_{exato} o vetor unitário e x_{aprox} o vetor da solução encontrada. Essa norma está apresentada na Tabela 2.

Figura 5 – Gráfico com os tempos de execução de todos os métodos.



$$\|x_{erro}\| = \sqrt{\sum_{i=0}^{i<N} (x_{exato} - x_{aprox})^2} \tag{3.22}$$

Tabela 2 – Avaliação da norma do erro para as precisões (simples e dupla) em uma matriz de ordem 2000.

Método direto	Precisão simples	Precisão dupla
Eliminação Gaussiana	29,32	0
Gauss-Jordan	29,32	0
Decomposição LU	469,49	0
Decomposição LDU	469,48	0
Decomposição Cholesky	644,55	0
Decomposição QR	4719,97	59,60

Considerando a qualidade dos resultados obtidos, todos os métodos tiveram valores diferentes para as incógnitas na precisão simples e somente a Decomposição QR apresentou resultados diferentes na precisão dupla. Podemos observar que a Decomposição QR é a que mais sofre com propagação de erros numéricos quando comparada aos outros métodos. Este fato ocorre porque a Decomposição QR utiliza muitas operações de multiplicação e, por isso, escala a quantidade de dígitos muito rapidamente. Na Decomposição QR pode ocorrer truncamento de dados devido ao *overflow*. Assim, comprovamos experimentalmente que a multiplicação é uma das operações com mais arredondamentos e truncamentos, como apresentado em (RUGGIERO; LOPES, 1997).

Para a segunda etapa deste estudo, foram utilizadas matrizes maiores, de ordem 5000, 10000, 15000 e 20000. Estas matrizes também foram geradas com a fórmula (3.21). Os experimentos foram executados 10 vezes, para cada tamanho de matriz e cada método. O desvio padrão foi menor do que 5%.

O gráfico na Figura 6 mostra o tempo médio de execução da Eliminação Gaussiana.

A medida que os dados de entrada aumentam de tamanho, a diferença absoluta entre as duas precisões, a simples e a dupla, aumenta linearmente. A Tabela 3 apresenta a porcentagem da diferença absoluta em relação ao tempo da precisão simples.

A Tabela 3 mostra que com 5000 incógnitas o tempo obtido com a precisão dupla é

Figura 6 – Tempos médios de execução da Eliminação Gaussiana

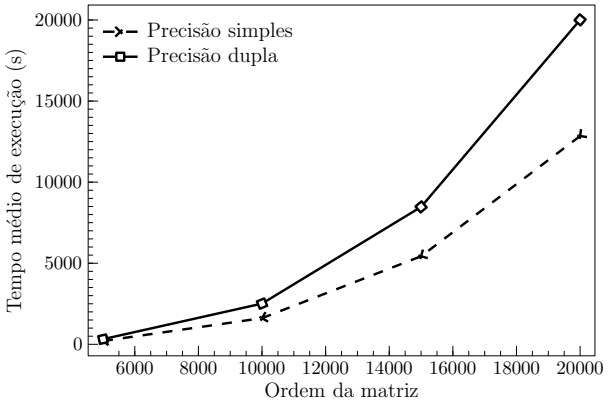


Tabela 3 – Porcentagem da diferença absoluta dos tempos da precisão simples em relação a precisão dupla da Eliminação Gaussiana.

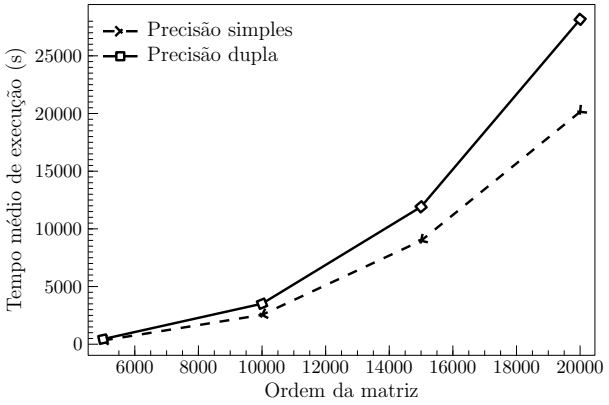
Ordem da matriz A	Diferença entre os tempos (%)
5000	54,64
10000	55,68
15000	56,07
20000	55,80

54,64% maior que o tempo da precisão simples. Da mesma forma, para todos os outros tamanhos de matrizes o tempo obtido com a precisão dupla ultrapassou 55%.

Todos os tempos com as variáveis de precisão dupla na Eliminação Gaussiana são cerca de 1,5 vezes o tempo da precisão simples.

A Figura 7 mostra o tempo médio de execução do Método de Gauss-Jordan.

Figura 7 – Tempos médios de execução do Método de Gauss-Jordan.



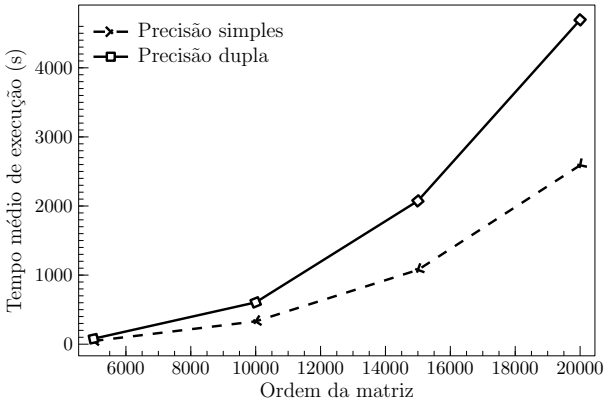
A medida que os dados de entrada aumentam de tamanho, a diferença absoluta entre as duas precisões, a simples e a dupla também aumenta. A Tabela 4, ilustra a porcentagem da diferença absoluta em relação ao tempo da precisão simples.

A Tabela 4 indica que com 5000 incógnitas o tempo da precisão dupla é 36,31% maior que o tempo da precisão simples. Observamos no Gauss-Jordan o mesmo comportamento da eliminação gaussiana, para todos os outros tamanhos de matrizes o tempo obtido com a precisão dupla ultrapassou 32%.

Tabela 4 – Porcentagem da diferença absoluta dos tempos da precisão simples em relação a precisão dupla do Método de Gauss-Jordan.

Ordem da matriz A	Diferença entre os tempos (%)
5000	36,31
10000	37,94
15000	32,77
20000	39,79

Figura 8 – Tempos médios de execução da Decomposição LU.



A Figura 8 apresenta o tempo médio de execução da Decomposição LU.

A medida que os dados de entrada aumentam de tamanho, a diferença absoluta entre as duas precisões, a simples e a dupla, aumenta, mas não linearmente. A Tabela 5 mostra a porcentagem da diferença absoluta em relação ao tempo da precisão simples.

Tabela 5 – Porcentagem da diferença absoluta dos tempos da precisão simples em relação a precisão dupla da Decomposição LU.

Ordem da matriz A	Diferença entre os tempos (%)
5000	70,63
10000	83,20
15000	92,97
20000	81,27

A Tabela 5 indica que com 5000 incógnitas o tempo da precisão dupla é 70,63% maior que o tempo da precisão simples. Com 10000 incógnitas, o tempo da precisão dupla é 83,20% maior. Para 15000 incógnitas, o tempo da precisão dupla continua crescendo (igual a 92,97% maior). Porém, para 20000 incógnitas, o tempo da precisão dupla decai para 81,27% da precisão simples, embora esteja acima de 81%.

A Figura 9 apresenta o tempo médio de execução da Decomposição LDU.

A medida que os dados de entrada aumentam de tamanho, a diferença absoluta entre as duas precisões, a simples e a dupla, aumenta. Observe que, novamente, não é um crescimento linear. A Tabela 6 mostra a porcentagem da diferença absoluta em relação ao tempo da precisão simples.

A Tabela 6 indica que com 5000 incógnitas o tempo da precisão dupla é 40,77% maior que o tempo da precisão simples. Para 10000 incógnitas a porcentagem é mais do que o

Figura 9 – Tempos médios de execução da Decomposição LDU.

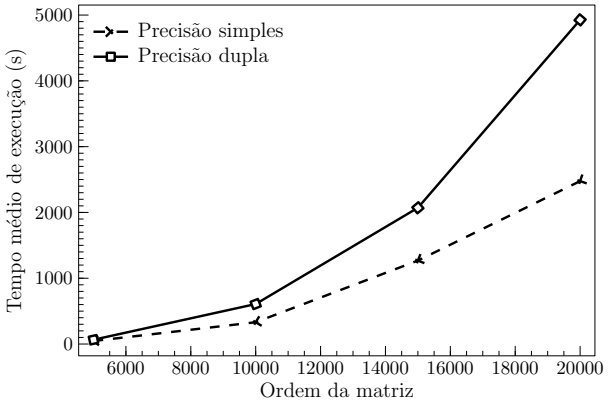


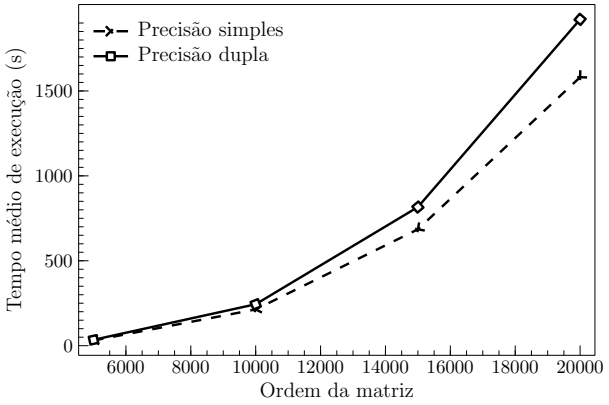
Tabela 6 – Porcentagem da diferença absoluta dos tempos da precisão simples em relação a precisão dupla da Decomposição LDU.

Ordem da matriz A	Diferença entre os tempos (%)
5000	40,77
10000	83,48
15000	63,29
20000	99,00

dobro (83,48%) do que com 5000 incógnitas. Para 15000 incógnitas, o tempo da precisão dupla passa a ser 63,29% maior e para 20000 incógnitas, o tempo da precisão dupla ultrapassa o da precisão simples em 99,00%. Ou seja, o código com precisão dupla é praticamente o dobro do tempo do código com precisão simples.

A Figura 10 apresenta o tempo médio de execução da Decomposição de Cholesky.

Figura 10 – Tempos médios de execução da Decomposição de Cholesky.



A medida que os dados de entrada aumentam de tamanho, a diferença absoluta entre as duas precisões, a simples e a dupla, aumenta, mas não linearmente. A Tabela 7 mostra a porcentagem da diferença absoluta em relação ao tempo da precisão simples.

A Tabela 7 indica que com 5000 incógnitas o tempo da precisão dupla é 12,31% maior que o tempo da precisão simples. Note que esta é a menor porcentagem observada, se considerarmos todos os outros métodos. Para 10000 incógnitas, o tempo da precisão dupla é 14,32% maior que o da precisão simples, enquanto que para 15000 incógnitas, o tempo

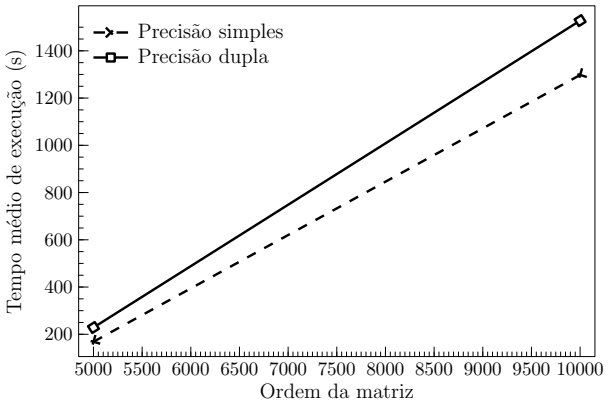
Tabela 7 – Porcentagem da diferença absoluta dos tempos da precisão simples em relação a precisão dupla da Decomposição de Cholesky.

Ordem da matriz A	Diferença entre os tempos (%)
5000	12,31
10000	14,32
15000	19,13
20000	21,55

da precisão dupla é 19,13% maior que o da precisão simples. Para 20000 incógnitas, o tempo da precisão dupla é 21,55% maior que o da precisão simples.

A Figura 11 apresenta o tempo médio de execução da Decomposição QR com ortogonalização Gram-Schmidt.

Figura 11 – Tempos médios de execução da Decomposição QR.



A Decomposição QR é a decomposição com a maior quantidade de operações por iteração, o que pode levar a erros do Sistema Operacional. O tempo com 1000 incógnitas é oito vezes maior que o tempo com 5000 incógnitas. Por essa razão os outros dois tamanhos de dados de entrada (15000 e 20000) não puderam ser testados. A Tabela 8 mostra a porcentagem da diferença absoluta em relação ao tempo da precisão simples.

Tabela 8 – Porcentagem da diferença absoluta dos tempos da precisão simples em relação a precisão dupla da Decomposição QR.

Ordem da matriz A	Diferença entre os tempos (%)
5000	35,79
10000	17,74

A Tabela 8 indica que com 5000 incógnitas o tempo da precisão dupla é 35,79% maior que o tempo da precisão simples. Com 10000 incógnitas, o tempo da precisão dupla é 17,74% maior.

O que podemos observar desse estudo é que ao declararmos as variáveis com precisão dupla o tempo total de execução é maior quando comparado ao de execução com precisão simples. Porém, os resultados obtidos são mais precisos. Dependendo do método essa diferença de tempo pode ser grande. Com 20000 incógnitas, o tempo de execução com variáveis de precisão dupla é de no máximo duas vezes o tempo de execução com variáveis de precisão simples.

3.16.3 Influência das *flags* de compilação

Este último estudo também foi dividido em duas partes, uma para comparar o tempo de execução em cada *flag* e a outra para ver o impacto que cada uma dessas *flags* tem na qualidade do resultado final. O sistema operacional utilizado foi o Ubuntu versão 16.04 nas duas partes. Os métodos foram compilados no GCC versão 5.4.0 e no GCC versão 7.1.0.

3.16.3.1 Tempo de execução considerando todas as *flags*

Para esse teste, usamos um sistema linear de ordem 5000. Cada combinação de *flag* e método foi executada pelo menos cinco vezes para calcular a média. O desvio padrão foi inferior a 1 segundo.

A Tabela 9 apresenta o tempo médio de execução para o método da Eliminação Gaussiana nas duas versões do GCC (5.4.0 e 7.1.0). Todos os tempos são mostrados em segundos.

Tabela 9 – Tempos de execução da Eliminação Gaussiana - matriz de ordem 5000.

	GCC versão 5.4.0		**GCC versão 7.1.0**	
Flag	Precisão simples segundos	Precisão dupla segundos	Precisão simples segundos	Precisão dupla segundos
-O0	410,45	413,15	417,31	421,74
-O	139,99	147,88	139,67	149,60
-O1	139,30	147,93	139,74	150,08
-O2	205,27	316,08	205,34	317,09
-O3	204,88	316,82	205,79	317,40
-Ofast	205,56	316,90	207,51	317,12
-Og	139,93	146,45	138,20	147,11
-Os	204,24	316,86	204,45	316,87

Para este método, as *flags* -O3, -Ofast e a -O2 obtiveram tempos de execução similares. A *flag* com menor tempo de execução foi a *flag* -O1 (e a -O). Como a partir do nível da *flag* -O2 o compilador pode aumentar o tamanho do código, isso pode acarretar num aumento significativo no número total de variáveis. Portanto, um aumento no acesso a memória. Outra observação é em relação a *flag* -Og, o nível de otimização focado na depuração. No método da Eliminação Gaussiana, essa *flag* é a terceira mais rápida por ter tempos similares ao primeiro nível de otimização.

Na Tabela 10 mostra os tempos médios de execução para o Método de Gauss-Jordan nas versões 5.4.0 e 7.1.0 do GCC. Todos os tempos são mostrados em segundos.

Os resultados apresentados demonstram que este algoritmo não pode ser tão otimizado quanto a Eliminação Gaussiana. O ganho de tempo na Eliminação Gaussiana é de aproximadamente 3 vezes, enquanto no Método de Gauss-Jordan o ganho máximo é de 2,5 vezes. Observamos um decréscimo no tempo de execução quando utilizado o primeiro nível de otimização no Método de Gauss-Jordan (*flag* -O). O que não ocorre nos níveis seguintes, com *flags* que permitem mais níveis de otimização. Este fato se deve a sequencialidade e dependência entre iterações do Método de Gauss-Jordan.

Comparando a Eliminação Gaussiana com o Método de Gauss-Jordan, observamos que o primeiro método pode ser considerado uma opção melhor, porque consome menos

Tabela 10 – Tempos de execução do Método de Gauss-Jordan - matriz de ordem 5000.

	GCC versão 5.4.0		**GCC versão 7.1.0**	
Flag	Precisão simples segundos	Precisão dupla segundos	Precisão simples segundos	Precisão dupla segundos
-O0	799,43	823,06	821,80	833,27
-O	342,39	467,30	338,41	463,62
-O1	342,83	466,87	340,40	463,72
-O2	323,91	441,57	323,96	441,29
-O3	323,88	441,49	324,43	442,05
-Ofast	322,22	442,22	322,51	442,19
-Og	342,02	463,40	340,54	464,37
-Os	323,45	441,58	317,42	440,68

tempo de execução e pode ser otimizado um pouco mais do que o segundo. Do nível zero de otimização (*flag* -O0) para o primeiro nível de otimização (-O), a Eliminação Gaussiana reduz o tempo de execução em 3 vezes e o Método de Gauss-Jordan em 2.5 vezes.

A Tabela 11 apresenta os tempos médios de execução da Decomposição LU nas duas versões do GCC (5.4.0 e 7.1.0). Todos os tempos são mostrados em segundos.

Tabela 11 – Tempos de execução da Decomposição LU - matriz de ordem 5000.

	GCC versão 5.4.0		**GCC versão 7.1.0**	
Flag	Precisão simples segundos	Precisão dupla segundos	Precisão simples segundos	Precisão dupla segundos
-O0	284,24	290,12	286,43	296,84
-O	108,49	118,74	108,24	117,98
-O1	108,43	118,80	107,69	118,04
-O2	59,12	80,34	59,10	76,65
-O3	46,29	78,98	45,91	70,99
-Ofast	46,85	78,96	61,40	79,57
-Og	106,61	118,12	107,63	117,78
-Os	77,27	79,53	64,14	79,33

Comparando os tempos de execução nas duas versões, observamos que a versão mais recente do GCC (2017) não tem tempos de execução menores do que a versão de 2016. A versão 5.4.0 do GCC foi disponibilizada com a versão do sistema operacional Ubuntu usado nos testes. Assim, acreditamos que essa versão do GCC pode ser otimizada especificamente para este sistema operacional. A versão 7.1.0 foi instalada manualmente. Obtivemos os tempos de execução mais rápidos com a *flag* -O3. Quando utlizamos a *flag* -Ofast o tempo de execução da versão mais recente do GCC aumentou, comparado com os tempos da *flag* -O3. Esse resultado pode ser considerado inesperado, pois com a *flag* '-Ofast' o programa final pode rearranjar operações aritméticas e comparações para evitar operações repetidas.

A Tabela 12 mostra o tempo médio de execução da Decomposição LDU nas versões 5.4.0 e 7.1.0 do compilador GCC. Todos os tempos são mostrados em segundos.

Assim como na Decomposição LU, a *flag* que obteve o melhor resultado foi a *flag* -O3. E ao comparar os tempos de execução dos dois métodos, Decomposição LU (Tabela 11)

Tabela 12 – Tempos de execução da Decomposição LDU - matriz de ordem 5000.

	GCC versão 5.4.0		**GCC versão 7.1.0**	
Flag	Precisão simples segundos	Precisão dupla segundos	Precisão simples segundos	Precisão dupla segundos
-O0	284,45	290,67	288,24	297,53
-O	107,82	113,24	108,07	119,13
-O1	107,86	121,35	108,10	119,16
-O2	59,31	71,79	59,47	80,94
-O3	46,17	64,99	46,55	80,10
-Ofast	47,45	79,48	52,65	114,12
-Og	109,71	117,34	107,40	118,95
-Os	74,91	86,43	63,69	87,22

e Decomposição LDU (Tabela 12), percebemos que os tempos são similares. Como a Decomposição LDU tem o mesmo algoritmo que a Decomposição LU apenas adicionando um pequeno passo na decomposição final, essa similaridade nos tempos é esperada.

A Tabela 13 apresenta o tempo médio de execução da Decomposição de Cholesky nas versões 5.4.0 e 7.1.0 do GCC.

Tabela 13 – Tempos de execução da Decomposição de Cholesky - matriz de ordem 5000.

	GCC versão 5.4.0		**GCC versão 7.1.0**	
Flag	Precisão simples segundos	Precisão dupla segundos	Precisão simples segundos	Precisão dupla segundos
-O0	122,38	124,96	121,85	124,12
-O	30,60	33,93	30,17	33,66
-O1	30,51	33,91	30,17	33,62
-O2	30,47	34,04	30,14	33,81
-O3	30,54	34,30	30,22	33,35
-Ofast	19,55	40,95	19,68	34,81
-Og	47,80	49,64	47,55	49,64
-Os	31,34	35,05	30,98	35,70

Este método tem o segundo maior ganho nos níveis de otimização. No primeiro nível, o tempo de execução é reduzido a um terço do tempo no nível zero de otimização. Com a *flag* -Ofast, proporcionou o menor tempo de execução para a Decomposição de Cholesky nas variáveis de precisão simples. O tempo de execução foi reduzido a 1/6 do tempo original.

A Decomposição de Cholesky é a única em que diferentes *flags* produzem tempos de execução menores, de acordo com a precisão das variáveis. Neste método, para as variáveis de precisão simples, a *flag* -Ofast é que proporcionou o menor tempo de execução. Para as variáveis de precisão dupla, a *flag* -O3 é que proporcionou o menor tempo.

A Tabela 14 mostra os tempos médios de execução para a Decomposição QR com ortogonalização de Gram-Schmidt nas duas versões do GCC, a 5.4.0 e a 7.1.0. Todos os tempos são mostrados em segundos.

A Decomposição QR é o método com o maior tempo de execução, porque possui o algoritmo mais complexo. Neste algoritmo, a matriz do sistema linear é iterada 3 vezes.

Tabela 14 – Tempos de execução da Decomposição QR - matriz de ordem 5000.

	GCC versão 5.4.0		**GCC versão 7.1.0**	
Flag	Precisão simples segundos	Precisão dupla segundos	Precisão simples segundos	Precisão dupla segundos
-O0	970,45	978,21	972,52	985,82
-O	202,20	215,15	203,25	219,55
-O1	202,25	215,15	203,21	219,49
-O2	202,23	213,81	234,20	238,09
-O3	168,37	228,64	188,12	212,06
-Ofast	81,72	149,63	78,63	152,67
-Og	224,93	238,42	224,26	245,42
-Os	224,47	234,26	223,36	240,93

A primeira para criar a matriz ortogonal Q. A segunda para multiplicar a matriz Q pelo vetor $\vec{b}$. E a última iteração encontra os valores do vetor $\vec{x}$ de incógnitas. Como este algoritmo tem bastante operações matemáticas de multiplicação e divisão, ele é o que mais obtém ganhos no processo de otimização através das *flags* do compilador. Com a *flag* -Ofast, que não faz testes do padrão IEEE para as operações matemáticas, obtivemos uma redução no tempo de execução de 12 vezes.

Essas tabelas mostram que com apenas o primeiro nível de otimização (*flags* -O e -O1), o tempo de execução de todos os métodos diminui para pelo menos para a metade do tempo de execução proporcionado pelo nível zero de otimização (*flag* -O0).

3.16.3.2 Solução numérica de todas as *flags*

O próximo experimento está relacionado com a solução numérica obtida por cada *flag* em cada método. Neste teste, utilizamos o mesmo sistema linear do experimento anterior, de ordem 5000. Com isso, o resultado esperado é a solução trivial ($\vec{x} = (1, 1, \ldots, 1)$). Para uma melhor comparação da qualidade dos resultados entre as *flags*, calculamos a norma euclidiana dos erros (segundo (3.22)) entre o resultado esperado e o obtido.

Apesar dos tempos médios de execução apresentarem diferenças entre as duas versões do GCC estudadas, os resultados numéricos obtidos foram os mesmos entre a versão 5.4.0 e a versão 7.1.0. Portanto, apresentamos somente a norma euclidiana dos erros da versão 5.4.0.

A Tabela 15 mostra a norma euclidiana dos erros numéricos dos resultados obtidos pela Eliminação Gaussiana.

Através dessa tabela podemos observar que algumas *flags* não conseguiram calcular o resultado final do sistema, devido a quantidade de operações realizadas para encontrar as incógnitas. A grande quantidade operações de multiplicações e divisões provavelmente causou *overflow* (ou *underflow*). Dessa forma, o resultado impresso na tela é igual a -NaN. Baseado neste fato, as únicas quatro *flags* que podem ser escolhidas como as melhores para a Eliminação Gaussiana são as *flags* -O2, -O3, -Ofast e -Os. Observando os tempos de execução na Tabela 9, todas essas quatro *flags* podem ser escolhidas, uma vez que proporcionam tempos de execução similares.

Além disso, ressaltamos que é necessário considerar a precisão. Na tabela 15, observamos que a precisão simples tem uma norma euclidiana de erro de 10^{-3}. Este valor pode representar uma precisão ruim para algumas aplicações. Então, as variáveis com precisão

Tabela 15 – Norma euclidiana dos erros de precisão dos resultados da Eliminação Gaussiana - matriz com ordem 5000.

Flag	**Precisão simples**	**Precisão dupla**
-O0	-NaN	-NaN
-O	-NaN	-NaN
-O1	-NaN	-NaN
-O2	0,0010	0,0000
-O3	0,0010	0,0000
-Ofast	0,0010	0,0000
-Og	-NaN	-NaN
-Os	0,0010	0,0000

dupla, que tem norma euclidiana igual a zero, são a melhor opção.

A Tabela 16 mostra a norma euclidiana dos erros numéricos dos resultados do Método de Gauss-Jordan. Nela observamos um comportamento similar a Eliminação Gaussiana (Tabela 15).

Tabela 16 – Norma euclidiana dos erros de precisão dos resultados do Método de Gauss-Jordan - matriz com ordem 5000

Flag	**Precisão simples**	**Precisão dupla**
-O0	24,3308	24,2990
-O	24,3308	24,2990
-O1	24,3308	24,2990
-O2	0,0010	0,0000
-O3	0,0010	0,0000
-Ofast	0,0010	0,0000
-Og	24,3308	24,2990
-Os	0,0010	0,0000

Como o método de Gauss-Jordan transforma a matriz M na matriz canônica, o passo de encontrar as incógnitas não tem operações de multiplicação nem de divisão. Isso explica o motivo do Método de Gauss-Jordan apresentar resultados em todas as *flags*, ao contrário da Eliminação Gaussiana. Porém, para ele as melhores *flags* em relação a precisão são as mesmas do método anterior, as *flags* -O2, -O3, -Ofast e -Os. Ao considerarmos os tempos de execução (Tabela 10) concluímos que as quatro *flags* podem ser usadas para otimizar o método, uma vez que todas proporcionam tempos de execução similares.

As variáveis de precisão simples neste método nas quatro *flags* mencionadas anteriormente tem a norma euclidiana do erro igual a 10^{-3}. Este valor pode não ser aceitável em determinadas aplicações. Então, a melhor opção para o Método de Gauss-Jordan são as variáveis de precisão dupla.

Outra observação é que nas outras *flags* (-O0, -O, -O1 e -Og), ambas as precisões tem normas com valores altos e similares devido aos dados de entrada terem números grandes e o computador realizar truncamentos e arredondamentos nas operações de multiplicação e divisão, gerando *overflow* e/ou *underflow*.

A Tabela 17 apresenta a norma euclidiana dos erros numéricos dos resultados obtidos na Decomposição LU.

Tabela 17 – Norma euclidiana dos erros de precisão dos resultados da Decomposição LU - matriz com ordem 5000.

Flag	**Precisão simples**	**Precisão dupla**
-O0	0,0010	0,0000
-O	0,0010	0,0000
-O1	0,0010	0,0000
-O2	0,0010	0,0000
-O3	0,0010	0,0000
-Ofast	0,0010	0,0000
-Og	0,0010	0,0000
-Os	0,0010	0,0000

Nenhuma das *flags* alterou a precisão dos resultados. Então, para a Decomposição LU, o melhor nível de otimização pode ser escolhido somente com base na Tabela 11. Baseado neste fato, a melhor *flag* para este método é a *flag* -Ofast para a versão 5.4.0 do GCC e a -O3 para a versão 7.1.0.

Além disso, as variáveis de precisão dupla tem uma norma euclidiana de erros com valor 0 e as variáveis de precisão simples tem valor igual a 0,00010. A precisão simples tem um erro que pode não ser aceito em algumas aplicações, mesmo sendo de apenas 10^{-3}.

Na Decomposição LDU, os resultados em ambas precisões foram os mesmos observados na Decomposição LU. Então, a decisão de melhor *flag* é baseada somente no tempo de execução na Tabela 12. Para a versão de 2016 do GCC, as melhores *flags* são a -O3 e a -Ofast. Elas proporcionam tempos de execução bem similares. Para versão 2017 do GCC a melhor *flag* é a -O3. Como a norma euclidiana é a mesma da Decomposição LU, a Decomposição LDU com variáveis de precisão simples pode ser inviável em algumas aplicações, tendo um erro na ordem de 10^{-3}.

Para a Decomposição de Cholesky, todas as *flags* apresentaram os resultados com o mesmo valor de norma euclidiana igual a 0,0002 para precisão simples e zero para precisão dupla, como apresentado na tabela 18.

Tabela 18 – Norma euclidiana dos erros de precisão dos resultados da Decomposição de Cholesky - matriz com ordem 5000.

Flag	**Precisão simples**	**Precisão dupla**
-O0	0,0002	0,0000
-O	0,0002	0,0000
-O1	0,0002	0,0000
-O2	0,0002	0,0000
-O3	0,0002	0,0000
-Ofast	0,0002	0,0000
-Og	0,0002	0,0000
-Os	0,0002	0,0000

Assim, a melhor *flag* para este método pode ser escolhida com base apenas nos tempos de execução (Tabela 13). Considerando as variáveis de precisão simples a melhor *flag* para a Decomposição de Cholesky é a -Ofast. Para as variáveis de precisão dupla a *flag* -O3 é

a melhor.

O método da Decomposição de Cholesky apresentou a menor norma euclidiana de erros para as variáveis de precisão simples, na ordem de 10^{-4},. Este método tem o algoritmo mais simples quando comparado aos outros métodos dos estudados. Desta forma, esta menor norma nos resultados é esperada. Porém, algumas aplicações podem ser consideradas inviáveis com essa norma (10^{-4}). Por isso, a precisão dupla continua a ser a melhor opção.

Quanto a Decomposição QR por ortogonalização Gram-Schmidt, as normas euclidianas de erro observadas foram iguais a 0,0017 para as variáveis com precisão simples para quase todas as *flags*, exceto para a *flag* -Ofast, onde o valor da norma foi igual a 0,0014. Para as variáveis de precisão dupla para todas as *flags* o valor da norma foi igual a zero. Estes valores estão apresentados na tabela 19.

Tabela 19 – Norma euclidiana dos erros de precisão dos resultados da Decomposição QR por ortogonalização Gram-Schmidt - matriz com ordem 5000.

Flag	**Precisão simples**	**Precisão dupla**
-O0	0,0017	0,0000
-O	0,0017	0,0000
-O1	0,0017	0,0000
-O2	0,0017	0,0000
-O3	0,0017	0,0000
-Ofast	0,0014	0,0000
-Og	0,0017	0,0000
-Os	0,0017	0,0000

Este método é o que possui a maior norma de erro, devido a quantidade de multiplicações e divisões realizadas no método. Isso pode gerar muitos erros de truncamento e arredondamento, como abordado em (RUGGIERO; LOPES, 1997), levando a *overflows* e *underflows*.

A maioria das *flags* na Decomposição QR não alteram a norma dos erros, exceto a *flag* -Ofast. Considerando este fato e observando os tempos de execução na Tabela 14, podemos concluir que a *flag* -Ofast é a melhor opção de otimização para a Decomposição QR com ortogonalização Gram-Schmidt.

As Tabelas 20 e 21 apresentam um resumo das melhores *flags* para cada método utilizando o compilador GCC versão 5.4.0 e na versão 7.1.0, respectivamente. Além da melhor *flag*, incluímos a norma do erro encontrada e a redução no tempo de execução em relação ao tempo do método sem otimização (nível -O0).

Tabela 20 – Resumo com a melhor *flag* para cada método na versão 5.4.0 do GCC - matriz de ordem 5000.

	Precisão simples			**Precisão dupla**		
Método	***Flag***	**Norma**	**Redução (%)**	***Flag***	**Norma**	**Redução (%)**
Eliminação Gaussiana	-O3	0,0010	50,08	-O3	0,0000	23,32
Método de Gauss-Jordan	-O3	0,0010	58,49	-O3	0,0000	46,36
Decomposição LU	-Ofast	0,0010	83,52	-Ofast	0,0000	72,78
Decomposição LDU	-O3	0,0010	83,77	-O3	0,0000	77,64
Decomposição Cholesky	-Ofast	0,0002	84,02	-O3	0,0000	72,55
Decomposição QR	-Ofast	0,0014	91,58	-Ofast	0,0000	84,70

Tabela 21 – Resumo com a melhor *flag* para cada método na versão 7.1.0 do GCC - matriz de ordem 5000.

	Precisão simples			**Precisão dupla**		
Método	***Flag***	**Norma**	**Redução (%)**	***Flag***	**Norma**	**Redução (%)**
Eliminação Gaussiana	-Os	0,0010	51,01	-Os	0,0000	24,87
Método de Gauss-Jordan	-Os	0,0010	61,38	-Os	0,0000	47,11
Decomposição LU	-O3	0,0010	83,80	-O3	0,0000	76,08
Decomposição LDU	-O3	0,0010	83,85	-O3	0,0000	73,08
Decomposição Cholesky	-Ofast	0,0002	83,85	-O3	0,0000	73,13
Decomposição QR	-Ofast	0,0014	91,91	-Ofast	0,0000	84,51

4 MÉTODOS ITERATIVOS EM MATRIZES CHEIAS

Neste capítulo, focamos nos métodos iterativos aplicados em sistemas lineares considerando matrizes cheias. Realizamos um estudo comparativo entre os seguintes métodos: Jacobi-Richardson (JR) (CUNHA, 2003), Gauss-Seidel (GS) (FRANCO, 2006), método de sobre-relaxação (SOR) (CUNHA, 2003) e o Jacobi com sobre-relaxação (JOR) (SAAD, 2003). Além desses, outros dois métodos mais recentes foram comparados. O método proposto por (SHANG, 2009), que é uma versão paralelizada do método Gauss-Seidel (DGS) implementada em PVM (GEIST et al., 1994) e o método JOR modificado (ANTUONO; COLICCHIO, 2016). Neste capítulo estudamos a influência da troca do parâmetro de balanceamento do método DOR e comparamos com outros métodos da literatura. Além de compararmos, também, a nossa versão do Gauss-Seidel distribuído, com outra biblioteca de troca de mensagens, com a versão sequencial. Esse estudo tem como base o artigo de 1977 (CONRAD; WALLACH, 1977) que apresenta implementações do Jacobi e do Gauss-Seidel, assim como versões paralelas e análise de convergência.

4.1 Métodos iterativos

Todos os métodos iterativos tratados neste capítulo convergem quando a matriz A tem uma norma menor que 1 em qualquer norma de matriz, ou seja, $\|A\| < 1$. Os métodos também consideram a decomposição da matriz A na forma apresentada em (4.1). As matrizes L e U são matrizes triangulares inferiores e superiores, respectivamente, e a matriz D é uma matriz diagonal.

$$A = L + D + U, \tag{4.1}$$

No livro (SAAD, 2003), o autor utiliza a decomposição mostrada em (4.2), considerando que $E = -L$ e $F = -U$ e assim temos que

$$A = D - E - F. \tag{4.2}$$

A decomposição 4.2 gera maior rapidez computacional em relação à 4.1, pois os esquemas finais dos métodos iterativos (baseados na fórmula apresentada em (4.3)) realizam apenas operações de adição em vez de subtrações. Esta rapidez decorre do modo como a operação de subtração é implementada nos processadores. Os valores são representados em complemento a dois e realizadas apenas operações de adição (PATTERSON; HENNESSY, 2014).

$$x = D^{-1}(b + Ex + Fx). \tag{4.3}$$

No método Jacobi-Richardson (JR) utilizamos o vetor $x^{(i-1)}$ da iteração anterior para calcular o vetor $x^{(i)}$ da iteração atual, como pode ser visto em (4.4).

$$x^{(i)} = D^{-1}(b + Ex^{(i-1)} + Fx^{(i-1)}) \tag{4.4}$$

O método Gauss-Seidel (GS), que utiliza a fórmula (4.5), foi proposto a partir de uma pequena modificação no método JR. Esta modificação é baseada na computação sequencial realizada pelos computadores. Cada elemento do vetor x é calculado sequencialmente, começando com x_1 até o x_n. Por isso, no método GS, os elementos anteriores da iteração atual são utilizados para calcular o elemento atual.

$$x^{(i)} = D^{-1}(b + Ex^{(i)} + Fx^{(i-1)}) \tag{4.5}$$

Em (SHANG, 2009) foi proposta uma versão distribuída do método de Gauss-Seidel (DGS). Nesta proposta os dados são divididos em blocos de linhas. Blocos com g linhas são atribuídos a cada processo e a quantidade g de linhas é calculada a partir de uma equação que considera o poder computacional de cada nó do *cluster* utilizado. Na versão DGS o processamento de cada iteração do vetor x é dividido em duas etapas. Primeiro, todos os processos calculam paralelamente $z^{(i)} = b + Fx^{(i-1)}$ sobre a sua fatia de dados. Para calcular o restante da equação ($x^{(i)} = D^{-1}(z^{(i)} + Ex^{(i)})$), cada processo deve esperar pelos valores do vetor x da iteração atual calculados nos processos que possuem os índices anteriores do vetor x. O último processo envia o vetor inteiro para o processo inicial caso a solução final seja encontrada. Caso contrário, ele envia o vetor para todos os outros processos para iniciar a próxima iteração.

Os dois próximos métodos, SOR e JOR, utilizam a metodologia de sobre-relaxação. Essa sobre-relaxação é feita através de um balanceamento entre o valor do vetor x da iteração anterior e o calculado na iteração atual. O balanceamento utiliza o parâmetro ω considerando $0 < \omega < 2$. Mais detalhes sobre a convergência do método SOR podem ser visto em (CUNHA, 2003). O método SOR (4.6) é baseado no método GS para calcular o vetor x na iteração atual. Já o método JOR (4.7) é baseado no método JR para calcular o vetor x.

$$x^{(i)}_{SOR} = (1 - \omega)x^{i-1}_{SOR} + \omega x^{i}_{GS} \tag{4.6}$$

$$x^{(i)}_{JOR} = (1 - \omega)x^{i-1}_{JOR} + \omega x^{i}_{JR} \tag{4.7}$$

Em (ANTUONO; COLICCHIO, 2016) foi proposto uma modificação do método JOR atrasando a relaxação, sendo denominado de DOR (*Delayed Over Relaxation*). Neste método o vetor solução x de duas iterações anteriores a iteração atual foi usado para encontrar a solução atual. Assim,

$$x^{(i)}_{DOR} = (1 - \omega)x^{i-2}_{DOR} + \omega x^{i}_{JR}. \tag{4.8}$$

4.2 **Algoritmos implementados**

Neste trabalho, todos os métodos (JR, GS, DGS, SOR, JOR e DOR) foram implementados na linguagem C que facilita a manipulação de memória, em particular, na alocação dinâmica (ASCENCIO; CAMPOS, 2008).

O algoritmo do método JR pode ser visto no Algoritmo 18.

No algoritmo JR, o teste para saber se o algoritmo encontrou a solução aproximada considera a diferença de cada elemento do vetor x_prox em relação ao vetor x. Quando a diferença de todos os elementos do vetor é menor do que o valor de tolerância pré-determinado, a solução é atingida e o processamento é interrompido.

A diferença do algoritmo JR para o GS está no segundo *loop* (linha 8 do Algoritmo 18). No GS, o *loop* usa o vetor x_prox em vez do vetor x, sendo então $soma \leftarrow soma + (A[i][j] * x\_prox[j])$.

Algoritmo 18 Algoritmo Jacobi-Richardson (JR).

```
1: para k ← 1 até it_max faça
2:     para i ← 0 até tam faça
3:         x_prox[i] ← B[i]; soma ← 0
4:         para j ← i + 1 até tam faça
5:             soma ← soma + (A[i][j] * x[j])
6:         x_prox[i] ← x_prox[i] + soma; soma ← 0
7:         para j ← 0 até i faça
8:             soma ← soma + (A[i][j] * x[j])
9:         x_prox[i] ← x_prox[i] + soma; x_prox[i] ← x_prox[i]/A[i][i]
10:    x ← x_prox
11:    // se convergir - break
```

O método SOR pode ser visto no Algoritmo 19.

Algoritmo 19 Algoritmo SOR.

```
1: para k ← 1 até it_max faça
2:     // calcular x_gs
3:     para i ← 0 até tam faça
4:         x_prox[i] ← (1-w) * x[i] + w *x_gs[i]
5:     x ← x_prox
6:     // se convergir - break
```

Os métodos SOR e JOR têm duas fases de processamento a cada iteração. Na primeira fase é calculado o vetor x_gs ou x_jr, respectivamente, no SOR e no JOR, com o método GS ou o método JR. A segunda fase é responsável por calcular o vetor x_prox usando o vetor x e o vetor x_gs (ou x_jr) com o parâmetro ω. Neste algoritmo a variável w armazena o valor determinado do parâmetro ω.

O método DGS (SHANG, 2009) utiliza o paradigma de troca de mensagens para comunicação entre os processos. O autor utilizou a biblioteca de comunicação paralela PVM (*Parallel Virtual Machine*) que foi criada pelo Laboratório Nacional de *Oak Ridge* (GEIST et al., 1994). Em nosso trabalho o DGS foi implementado utilizando a biblioteca de troca de mensagens MPI (*Message Passing Interface*) que é considerada padrão. Ela é portátil e largamente utilizada por pesquisadores e pela indústria (GROPP et al., 1999).

Com esse paradigma de paralelismo em mente, o cientista separou o sistema em blocos de linhas consecutivas para ser calculado por cada processador. Assim, cada processador calcula a parte que não dependia da iteração corrente em paralelo, ou seja, calcula $D^{-1}b - D^{-1}Ux^{(i-1)}$. Depois, o processador k recebe do processador $k-1$ os elementos do vetor x da iteração i já calculados (do x_0 ao último elemento antes do bloco do processador k), termina de calcular os elementos do seu bloco e manda todos para o processador $k+1$. O algoritmo pode ser visto em 20.

As chamadas para a API do MPI foram ocultadas para diminuir o tamanho do código. As chamadas usadas foram MPI_Send e MPI_Recv, a primeira para enviar assíncronamente, sem precisar esperar a mensagem chegar no destinatário para continuar sua computação, e a segunda para receber as mensagens. Como a mensagem tem que

Algoritmo 20 Algoritmo Gauss-Seidel distribuído (DGS).

```
1: se rank = 0 então
2:     g ← numTask − 1; bloco ← tam/g
3:     para i ← 1 até numTasks faça
4:         inicio ← bloco ∗ (i − 1); fim ← (i + 1 == numTasks)?tam:inicio + bloco
5:         //enviar parametros de inicio, fim e dados da matriz para a task i
6:     //receber vetor final x da ultima task
7: senão
8:     //receber parametros de inicio, fim e dados da matriz para a task i
9:     para k ← 1 até it_max faça
10:        para i ← inicio até fim faça
11:            x_prox[i] ← B[i]; soma ← 0
12:            para j ← i + 1 até tam faça
13:                soma ← soma + (A[i][j] ∗ x[j])
14:            x_prox[i] ← x_prox[i] + soma
15:        se rank > 1 então
16:            //receber os elementos de 0 a inicio da task rank − 1
17:        para i ← inicio até fim faça
18:            soma ← 0
19:            para j ← 0 até i faça
20:                soma ← soma + (A[i][j] ∗ x_prox[j])
21:            x_prox[i] ← x_prox[i] + soma; x_prox[i] ← x_prox[i]/A[i][i]
22:        se rank + 1! = numTasks então
23:            //enviar os elementos de 0 a inicio para a task rank + 1
24:            //receber o sinal de convergencia e se não convergir o vetor x completo da
    ultima task
25:        senão
26:            x ← x_prox
27:            //enviar sinal de convergencia para todas as tasks
28:            //se convergiu, enviar o vetor x para a task 0; se não convergiu, enviar o
    vetor x para todas as tasks
```

chegar antes de conseguir receber, a segunda função é síncrona, ou seja, para a execução do programa até receber a mensagem esperada. A variável g apresentada no código é um parâmetro definido pelos autores (SHANG, 2009) para poder balancear da melhor forma a carga entre os nós, caso tenha nós com poder de processamento diferentes. Como no nosso caso estamos fazendo um estudo em uma única máquina, deixamos o parâmetro g para ser igualitário entre os nós.

O Algoritmo 21 mostra a implementação do método DOR (ANTUONO; COLICCHIO, 2016).

O método DOR utiliza duas iterações anteriores do vetor x em seu esquema. Por isso, a primeira iteração deste método usa somente o método JR. A partir da segunda iteração, o vetor x_prox é calculado em duas partes, como mostrado no Algoritmo 21.

Algoritmo 21 DOR

```
1: /*calcular x_prox com o método JR*/
2: x_ant ← x; x ← x_prox
3: para k ← 2 até it_max faça
4:     para i ← 0 até tam faça
5:         x_jr[i] ← B[i]; soma ← 0
6:         para j ← i + 1 até tam faça
7:             soma ← soma + (A[i*tam + j] * x[j])
8:         x_jr[i] ← x_jr[i] + soma; soma ← 0
9:         para j ← 0 até i faça
10:            soma ← soma + (A[i*tam + j] * x[j])
11:        x_jr[i] ← x_jr[i] + soma; x_jr[i] ← x_jr[i] / A[i][i]
12:    para i ← 0 até tam faça
13:        x_prox[i] ← (1-w) * x_ant[i] + w * x_gs[i]
14:    x_ant ← x; x ← x_prox
15:    //se convergiu dar break
```

4.3 Testes numéricos e análise dos resultados

Em todos os experimentos, foram consideradas matrizes simétricas estritamente dominantes geradas aleatoriamente. Para satisfazer essa condição, as matrizes (A) foram geradas a partir da equação (4.9), onde C é uma matriz com números aleatórios de 0 a 999, n é a ordem da matriz e I é a matriz identidade. Em todos os testes, o sistema linear possui solução trivial. Então, o vetor solução b foi encontrado a partir da multiplicação da matriz A calculada e o vetor $\vec{x} = (1, 1, \ldots, 1)$. Foram geradas matrizes de dimensão 5000, 10000, 15000 e 20000 para os testes.

$$A = C \times C^T + 0{,}75 \times n \times I \tag{4.9}$$

O ambiente de execução utilizado nos experimentos possui um processador Intel Core i7 de 2,80 GHz, com 4 núcleos, 8 GB de memória principal, 8 MB de memória *cache* e executando com o sistema operacional Ubuntu versão 14.04. Cada método foi compilado com GCC versão 4.8.4.

4.3.1 Influência do parâmetro ω no método DOR

No artigo (ANTUONO; COLICCHIO, 2016), os autores apenas simulam o tempo e a convergência do método DOR. Por isso, nosso primeiro teste foi verificar experimentalmente como o método DOR se comporta. Com o intuito de avaliar a influência do parâmetro ω para o método DOR, testes computacionais variando o valor de ω e o tamanho das matrizes são mostrados na Tabela 22. Para cada ordem de matriz é mostrada a quantidade de iterações necessárias para a convergência (k).

Para o nosso sistema modelo foram encontrados resultados que convergiram considerando $\omega < 1$. Este fato não é previsto na teoria, embora necessite de uma quantidade maior de iterações (em (ANTUONO; COLICCHIO, 2016), os autores demonstram a convergência de ω para valores $1 < \omega < 2$). Porém, não converge para valores maiores que 2 (ver (ANTUONO; COLICCHIO, 2016)). Além disso, foi observado que os melhores

Tabela 22 – Número de iterações do método DOR para diferentes valores de ω.

ω	Matriz de ordem 5000 (k)	Matriz de ordem 20000 (k)
0,85	6329	6381
0,95	5167	5209
1,05	4226	4260
1,50	1527	1537
1,80	380	383
1,90	391	393
1,95	781	796
2,05	-	-

resultados foram encontrados com ω entre 1,80 e 1,90. Para se obter um valor experimental mais preciso, um segundo teste foi realizado (Tabela 23) variando ω na segunda casa decimal.

Tabela 23 – Número de iterações do método DOR com diferentes valores de ω entre 1,80 e 1,90.

ω	Matriz de ordem 5000 (k)	Matriz de ordem 20000 (k)
1,81	328	332
1,82	257	258
1,83	233	238
1,84	238	241
1,85	253	296
1,86	311	313
1,87	291	293
1,88	350	352
1,89	370	372

O melhor valor encontrado foi com $\omega = 1{,}83$. Uma terceira rodada de testes considerando mais uma casa decimal foi realizada. O melhor resultado encontrado foi com $\omega = 1{,}831$. Neste caso, a convergência foi obtida com $k = 225$ para a matriz de dimensão 5000 e $k = 238$ para a de dimensão 20000. Foi verificado que o acréscimo de mais casas decimais não altera a quantidade de iterações. Portanto, foi escolhido o valor $\omega = 1{,}831$ para ser usado nos estudos comparativos descritos a seguir.

4.3.2 Comparação dos métodos sequenciais

Neste teste comparativo foi utilizado apenas os métodos em suas versões sequenciais. Os resultados das quantidade de iterações (k) necessárias para obtenção da solução podem ser vistos na Tabela 24, para matrizes com dimensão (n) variando entre 5000 e 20000.

A Tabela 24 mostra que o método GS converge com o menor número de iterações. Os métodos com relaxação convergem com menor número de iterações se comparados ao método de JR. Além disso, percebemos que o método DOR aumenta a quantidade de iterações a medida que aumenta a dimensão da matriz, enquanto os métodos JOR e SOR continuam com a mesma quantidade de iterações independentemente da dimensão das

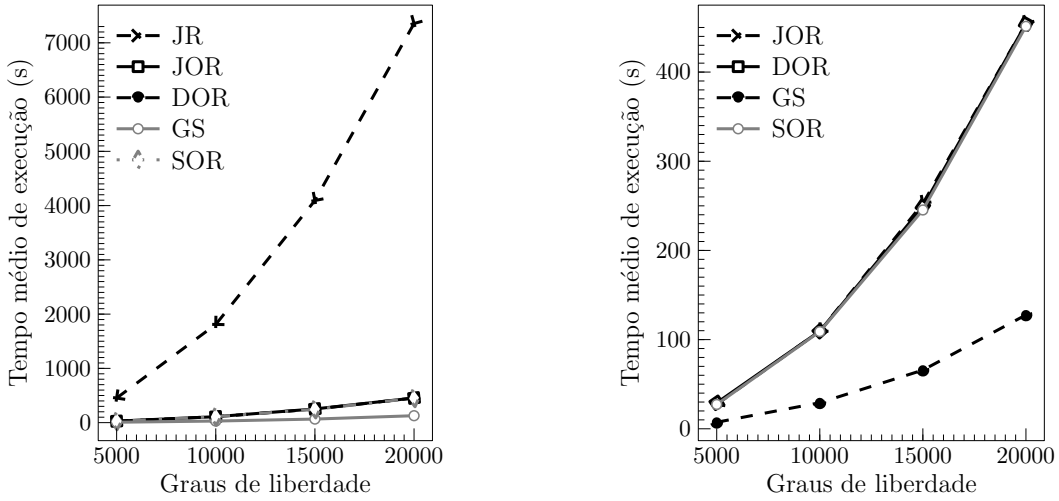
Tabela 24 – Quantidade de iterações para convergência.

ω	5000 (k)	10000 (k)	15000 (k)	20000 (k)
JR	4673	4676	4699	4711
GS	17	17	17	17
SOR com $\omega = 0{,}95$	215	215	215	215
JOR com $\omega = 0{,}95$	215	215	215	215
DOR com $\omega = 1{,}831$	225	226	229	230

matrizes. Esse aumento pode ser uma consequência de se usar duas interações anteriores para calcular a próxima.

Os próximos experimentos foram realizados para obter os tempos de execução de cada método. Todos os métodos foram executados 10 vezes, para a obtenção da média aritmética e do desvio padrão de cada um. As médias podem ser vistas na Figura 12a e o valor do desvio padrão é considerado desprezível (inferior a 0,15%).

Figura 12 – Estudo comparativo dos tempos de execução dos métodos.



(a) Todos os métodos. (b) Sem o método JR.

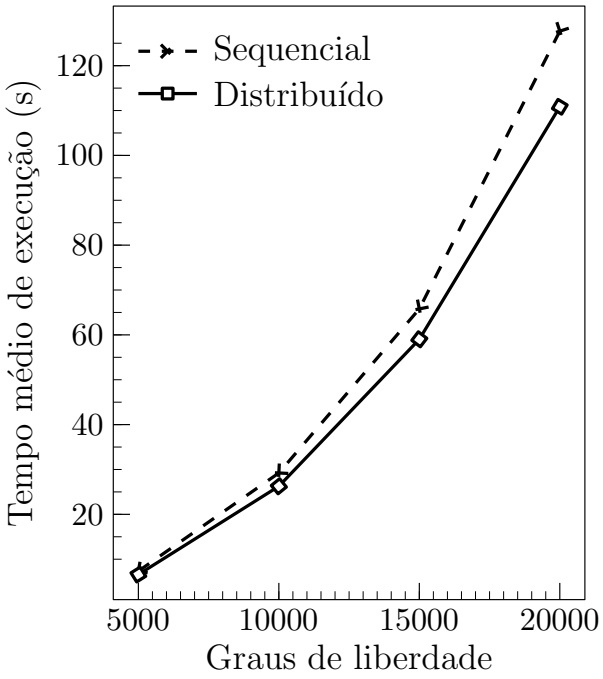
A Figura 12a mostra que o método JR tem um tempo superior aos demais métodos. Assim, geramos a Figura 12b sem o método JR e com outra escala para observar melhor os outros métodos. O método GS foi o método com o menor tempo de execução. Esse tempo aumenta proporcionalmente ao grau de liberdade do sistema. Outro fato observado é que os métodos com relaxação (JOR, SOR e o mais recente DOR) tem um tempo de execução muito similar entre eles. Assim, as iterações a mais que o método DOR possui em relação ao JOR não são suficientes para deixar o método mais lento.

4.3.3 Comparação dos métodos GS e DGS

O último experimento reproduz os resultados apresentados em (SHANG, 2009) com a biblioteca de troca de mensagens MPI e o valor do parâmetro g sendo a divisão entre o total de linhas pelos *cores* usados. A versão original é baseada na biblioteca PVM. A versão distribuída foi executada em uma única máquina com 4 *cores* por ser um estudo inicial.

A Figura 13 mostra os tempos de execução das versões sequencial e distribuída do método Gauss-Seidel. Para a comparação ser a mais precisa possível, o algoritmo da versão sequencial do GS apresentado na seção anterior (o Algoritmo 1 com a modificação na linha 9) teve a computação separada em duas partes. Primeiro o GS calcula $x^{(i)} = b + Fx^{(i-1)}$ para todas as posições de x_prox e depois calcula $x^{(i)} = D^{-1}(z^{(i)} + Ex^{(i)})$.

Figura 13 – Tempos de execução dos métodos GS (sequencial) e DGS (distribuído).



A versão distribuída do método GS obteve um tempo de execução menor em relação ao sequencial embora o *speedup*, ($s = \frac{t_{seq}}{t_{dis}}$), tenha sido pequeno. Para matrizes de dimensão 5000 a 15000 o *speedup* foi igual a 1,11 e para a matriz de 20000 o *speedup* foi igual a 1,15. Como usamos 4 processos (1 para cada núcleo), o *speedup* máximo ideal seria igual a 4. O método GS está distante deste *speedup* porque possui muita comunicação entre os processos. Esse *overhead* prejudica o possível ganho de desempenho.

No artigo de (SHANG, 2009), o *speedup* encontrado foi igual 2,08, usando 3 processadores, para 24000 incógnitas. (SHANG, 2009) divide os dados de entrada em mais do que três blocos, enviando mais de um bloco por processador ciclicamente. Como dito anteriormente, essa divisão em mais de 3 blocos é feita através do parâmetro g e considera o poder de processamento de cada nó do *cluster* usado (ou de cada processador). Como os experimentos neste trabalho foram feitos em uma máquina homogênea, o melhor cenário de divisão dos dados de entrada é a divisão igualitária, para não aumentar o *overhead* do tempo de comunicação.

5 MÉTODOS DE KRYLOV EM MATRIZES ESPARSAS

Este capítulo aborda alguns métodos iterativos aplicados para resolver sistemas lineares considerando matrizes esparsas usando subespaços de Krylov. Estes métodos são métodos de projeção ortogonal que utilizam o subespaço de Krylov como subespaço de procura $\mathcal{K}$ e restrição $\mathcal{L}$.

Todos os métodos comparados são baseados no método direto GMRES (Generalized Minimal Residual), método generalizado de mínimos residuais, criado em 1986 (SAAD; SCHULTZ, 1986). Este método procura minimizar a norma do vetor residual sobre um subespaço de Krylov a cada iteração. O GMRES utiliza o processo de ortogonalização de Gram-Schmidt modificado para encontrar os vetores ortonormais que fazem parte da base do subespaço de Krylov. O algoritmo inicia com um subespaço de dimensão 1 e a cada iteração aumenta essa dimensão em uma unidade. até chegar em um subespaço de tamanho igual a ordem da matriz do sistema linear.

Os métodos comparados são métodos encontrados na literatura (HACKBUSCH, 1994; SAAD, 2003) e métodos novos, como o αGMRES (BAKER; JESSUP; KOLEV, 2009) e o *Heavy Ball* GMRES (IMAKURA; LI; ZHANG, 2016).

O primeiro método comparado, também, foi apresentado em 1986 (SAAD; SCHULTZ, 1986) e é chamado de GMRES reinicializado (*restarted GMRES*). A modificação desse método em relação ao GMRES ocorre a cada m iterações do método GMRES original. Após essas m iterações o subespaço de Krylov é reinicializado com a dimensão igual a 1. Isso evita o estouro de memória para sistemas lineares de grandes porte e que necessitam de muitas iterações para convergir. Um ciclo do GMRES reinicializado engloba todas as m iterações do GMRES original e a reinicialização do subespaço, colocando como valor inicial do ciclo seguinte a solução aproximada encontrada no ciclo atual.

O segundo método comparado é o denominado GMRES flexível (FGMRES) (SAAD, 1993). Este método é baseado no GMRES reinicializado, porém, resolve o sistema pré-condicionado $AM^{-1}(Mx) = b$ em vez do $Ax = b$. O método cria a base do subespaço de Krylov através de vetores pré-condicionados $M^{-1}v_i$ e não dos vetores v_i. Além disso, a matriz pré-condicionadora M pode mudar a cada ciclo interno do FGMRES, para ter uma maior liberdade sobre o pré-condicionamento.

O próximo método comparado é o chamado αGMRES (BAKER; JESSUP; KOLEV, 2009). Ele é uma modificação do GMRES reiniciado que calcula o número m de iterações a cada ciclo do GMRES reiniciado para não considerar sempre a mesma quantidade de iterações. Ele utiliza o cosseno entre os vetores dos resíduos iniciais do ciclo atual e do anterior para calcular a quantidade de iterações necessárias para o ciclo seguinte.

O último método comparado, o *Heavy Ball* GMRES (IMAKURA; LI; ZHANG, 2016), utiliza um método iterativo conhecido na área de otimização matemática, o método *Heavy Ball*. Este método é utilizado para a otimização de uma função $f(x)$, que pode ou não ser contínua. Ele utiliza a diferença entre os valores iniciais das duas últimas iterações, assim como o valor final da última iteração, para atualizar o valor da iteração atual. A utilização do penúltimo valor encontrado melhora a convergência do método, reduzindo

o efeito "zigue-zague" que os métodos de otimização que usam apenas o último valor encontrado possuem (POLYAK, 1987).

Para modificar o GMRES incluindo o método *Heavy Ball*, os autores (IMAKURA; LI; ZHANG, 2016) acrescentaram o subespaço gerado pelo vetor diferença d dos vetores iniciais x_0 do último ciclo e do atual no subespaço de procura da solução aproximada do ciclo interno do GMRES reiniciado. Assim, antes de achar a solução aproximada do ciclo atual, incluí-se o vetor d na base ortogonal do subespaço de procura.

Neste capítulo, fizemos experimentos para comparar a taxa de convergência de cada método e o tempo de execução. Além disso, validamos nossas implementações comparando os resultados obtidos com os dos artigos (BAKER; JESSUP; KOLEV, 2009) e (IMAKURA; LI; ZHANG, 2016). Por fim, apresentamos um estudo com os possíveis pontos de paralelização do *Heavy Ball* GMRES, considerando uma implementação com uma API de memória compartilhada.

5.1 **Algoritmo do Método GMRES**

O primeiro algoritmo abordado é o do GMRES original (Algoritmo 22), como apresentado em (SAAD, 2003). Este algoritmo é a base do ciclo de cada um dos métodos seguintes. O problema geral do GMRES está descrito em (5.1).

$$x_m = x_0 + \arg \min_{z \in \kappa_k(A, r_0)} \|b - A \times (x_0 + z)\|_2 \tag{5.1}$$

Algoritmo 22 Algoritmo GMRES.

função GMRES(**Matriz de coeficientes:** A; **Vetor de valores independentes:** b; **Valor inicial:** x_0)

1: $r_0 \leftarrow b - A \times x_0$; beta $\leftarrow \|r_0\|_2$; $v_1 \leftarrow \frac{r_0}{\text{beta}}$
2: **para** $j \leftarrow 1$ até m **faça**
3: $\quad w_j \leftarrow A \times v_j$
4: $\quad$ **para** $i \leftarrow 1$ até j **faça**
5: $\quad\quad h_{i,j} \leftarrow \langle w_j, v_i \rangle$ ▷ A matriz H é uma matriz de Hessenberg de ordem $(m+1) \times m$
6: $\quad\quad w_j \leftarrow w_j - h_{i,j} \times v_i$
7: $\quad h_{j+1,j} \leftarrow \|w_j\|_2$
8: $\quad$ **se** $h_{j+1,j} = 0$ **então**
9: $\quad\quad$ m $\leftarrow$ j
10: $\quad\quad$ // dar break
11: $\quad v_{j+1} \leftarrow \frac{w_j}{h_{j+1,j}}$
12: /* calcular y_m que minimiza $\|\text{beta} \times e_1 - H \times y\|_2$ */
13: $x_m \leftarrow x_0 + V_m \times y_m$
14: **retorna** x_m

As variáveis r_0, x_0, x_m, v_i, w_i e e_1 são vetores. Os índices i apresentados em v_i e w_i representam a coluna i da matriz V_m e da matriz W_m, respectivamente. A variável $h_{i,j}$ representa o elemento da linha i e da coluna j da matriz H. O vetor e_1 é a primeira coluna da matriz identidade.

5.2 **Algoritmo do Método GMRES reiniciado**

O próximo algoritmo abordado é o GMRES reiniciado (SAAD; SCHULTZ, 1986). Ele cria ciclos de iterações baseado GMRES original. O intuito desses ciclos é reduzir a sobrecarga de armazenamento que os grandes sistemas lineares tem no método do GMRES original. Assim, quando essa quantidade m de iterações é computada, se o método ainda não convergiu, o ciclo de iterações se reinicia com o valor de x_m sendo o próximo x_0. O problema geral do GMRES reiniciado é descrito em (5.2).

$$x_m^{(l)} = x_0^{(l)} + \arg \min_{z \in \kappa_k(A, r_0^{(l)})} \|b - A(x_0^{(l)} + z)\|_2 \tag{5.2}$$

O Algoritmo 23 apresenta esse método como em (SAAD; SCHULTZ, 1986).

Algoritmo 23 Algoritmo GMRES reiniciado.

função GMRES(m)(**Matriz de coeficientes:** A; **Vetor de valores independentes:** b; **Valor inicial:** x_0)
1: **enquanto** resíduo r_m não convergir **faça**
2: chamar Algoritmo 22
3: $x_0 \leftarrow x_m$
4: **retorna** x_m

As diferenças desse algoritmo para o Algoritmo 22 estão no *loop* externo para reiniciar o ciclo e na atualização do x_0 com o x_m que foi calculado no ciclo do GMRES original.

5.3 **Algoritmo do Método GMRES flexível**

O algoritmo GMRES flexível (SAAD, 1993) incorpora aos vetores da base de Krylov uma matriz pré-condicionadora M. Depois, ele usa essa matriz precondicionada para transformar o vetor y_m em um vetor da base de x_0 para encontrar a solução x_m. Como o GMRES flexível é baseado no GMRES reiniciado, ele também utiliza ciclos do GMRES original e pode mudar a matriz M a cada ciclo. O algoritmo do GMRES flexível está mostrado no Algoritmo 24.

As diferenças desse método para o GMRES reiniciado (Algoritmo 23) estão nas linhas 4-5 e 15. As linhas 4-5 do Algoritmo 24 mostram a criação dos vetores z_j que são as colunas da matriz Z_m. Esta matriz Z_m é a matriz que multiplica o vetor y encontrado na solução do sistema linear auxiliar. Essa multiplicação é mostrada na linha 15. As outras partes do algoritmo são iguais ao GMRES reiniciado, tanto a ortogonalização como o problema de minimização.

Algoritmo 24 Algoritmo GMRES flexível.

função FGMRES(**Matriz de coeficientes:** A; **Vetor de valores independentes:** b; **Valor inicial:** x_0)

1: **enquanto** resíduo r_m não convergir **faça**
2: $r_0 \leftarrow b - A \times x_0$; beta $\leftarrow \|r_0\|_2$; $v_1 \leftarrow \frac{r_0}{\text{beta}}$
3: **para** $j \leftarrow 1$ até m **faça**
4: $z_j \leftarrow M^{-1} \times v_j$
5: $w \leftarrow A \times z_j$
6: **para** $i \leftarrow 1$ até j **faça**
7: $h_{i,j} \leftarrow \langle w, v_i \rangle$
8: $w \leftarrow w - h_{i,j} \times v_i$
9: $h_{j+1,j} \leftarrow \|w\|_2$
10: **se** $h_{j+1,j} = 0$ **então**
11: m $\leftarrow$ j // dar break
12: $v_{j+1} \leftarrow \frac{w}{h_{j+1,j}}$
13: /* calcular y_m que minimiza $\|\text{beta} \times e_1 - H \times y\|_2$ */
14: $x_m \leftarrow x_0 + Z_m \times y_m$; $x_0 \leftarrow x_m$
15: **retorna** x_m

5.4 Algoritmo do Método αGMRES

O αGMRES foi apresentado em 2009 (BAKER; JESSUP; KOLEV, 2009) e é uma modificação do GMRES reiniciado (Algoritmos 23 e 33). Ele flexibiliza a quantidade de iterações por ciclo do GMRES original. Os autores usaram como medida para determinar a quantidade m_l de iterações o ângulo entre o resíduo do ciclo atual e do ciclo anterior.

Este ângulo determina a relação entre a direção desses dois vetores. Se esse ângulo for pequeno, significa que os dois vetores estão em direções próximas e que o método está convergindo bem para a solução final. Dessa forma, são necessárias poucas iterações por ciclo, uma vez que provavelmente a solução final está próxima. Porém, se o ângulo entre os vetores for grande, significa que os dois vetores estão em direções divergentes ou que o método está quase estagnado, precisando assim de mais iterações no ciclo.

O resíduo da solução aproximada de um ciclo é ortogonal ao subespaço de Krylov definido pelo resíduo da solução aproximada do ciclo anterior. Assim, o cosseno do ângulo entre esses dois vetores residuais é dado por (5.3). Esse resultado é demonstrado pelo Teorema 4 em (BAKER; JESSUP; MANTEUFFEL, 2005).

$$\cos \angle(r_i, r_{i-1}) = \frac{\|r_i\|^2}{\|r_{i-1}\|^2} \tag{5.3}$$

O pseudocódigo do αGMRES pode ser visto no Algoritmo 25.

Este algoritmo possui mais parâmetros de entrada, m_{min} e m_{max}, com os possíveis valores mínimo e máximo de iterações. Além disso, a variável *cr* controla a taxa de convergência a cada ciclo do GMRES original (veja linha 5 do Algoritmo 25). Para calcular o valor da variável m_l a cada ciclo, determina-se um valor de taxa de convergência mínimo (min_cr) e máximo (max_cr). A variável d também é usada no cálculo de m_l, para determinar a quantidade de iterações que irão ser decrescidas do m_l. Os valores de max_cr, min_cr e d foram espelhados nos valores escolhidos pelos autores que apresentaram o

Algoritmo 25 Algoritmo αGMRES.

função αGMRES(**Matriz de coeficientes:** A; **Vetor de valores independentes:** b; **Valor inicial:** x_0; **Qtde mínima de iterações:** m_{min}; **Qtde máxima de iterações:** m_{max})

1: cr $\leftarrow$ 1; max_cr $\leftarrow$ cos 8; min_cr $\leftarrow$ cos 80; d $\leftarrow$ 3; $l \leftarrow 0$; $r_l \leftarrow b - A \times x_0$
2: **enquanto** resíduo r_m não convergir **faça**
3: calcular m_l (Algoritmo 26)
4: chamar Algoritmo 22
5: $x_0 \leftarrow x_m$; $l \leftarrow l + 1$; $r_l \leftarrow b - A \times x_0$; cr $\leftarrow \frac{\|r_l\|_2}{\|r_{l-1}\|_2}$
6: **retorna** x_m

αGMRES (BAKER; JESSUP; KOLEV, 2009). O algoritmo para calcular o valor de m_l é mostrado no Algoritmo 26.

Algoritmo 26 Ajuste do m_l para o método αGMRES.

1: **se** cr $>$ max_cr ou $l = 0$ **então**
2: $m_l \leftarrow m_{max}$
3: **senão**
4: **se** cr $<$ min_cr **então**
5: $m_l \leftarrow m_{l-1}$
6: **senão**
7: $m_l \leftarrow ((m_{l-1} - d) \geq m_{min})?m_{l-1} - d : m_{max}$

O algoritmo para calcular m_l é simples. Caso o cosseno entre os vetores seja maior que o máximo estabelecido, o ciclo corrente terá a quantidade máxima de iterações previstas. Caso o cosseno seja menor que o valor mínimo estabelecido, significa que o método está convergindo bem e continua com a mesma quantidade de iterações que o ciclo anterior. Por fim, quando o cosseno dos resíduos sequenciais está dentro da faixa de ajuste, tenta-se diminuir d iterações da quantidade do ciclo anterior. Se esse valor for menor que a quantidade mínima de iterações necessária, então o ciclo atual tem m_{max} iterações.

5.5 Algoritmo do Método *Heavy Ball* GMRES

O último método abordado é o *heavy ball* GMRES (IMAKURA; LI; ZHANG, 2016). A ideia da modificação desse método é baseada na área de otimização de funções de minimização ou maximização. O método do *heavy ball* é iterativo, e usa mais de uma iteração anterior para calcular o valor da solução aproximada da iteração atual. Por isso, esse método faz parte dos chamados métodos de passo múltiplo (*multistep methods*), como explicado em (POLYAK, 1987).

O método *Heavy Ball* está descrito em (5.4). Para calcular o valor da iteração $l + 1$, ele utiliza os valores da iteração atual l e a diferença dos valores da iteração atual com os da iteração anterior $l - 1$.

$$x^{(l+1)} = x^{(l)} - \alpha \bigtriangledown f(x^{(l)}) - \beta(x^{(l)} - x^{(l-1)}) \tag{5.4}$$

No GMRES, os autores (IMAKURA; LI; ZHANG, 2016) acrescentaram o subespaço gerado pela diferença entre os resíduos iniciais do ciclo anterior e o atual, na base de

procura da solução aproximada. Assim, o problema geral do GMRES descrito em (5.1) foi modificado para o problema do *Heavy Ball* GMRES descrito em (5.5).

$$x_m^{(l)} = x_0^{(l)} + \arg \min_{z \in \kappa_k(A, r_0^{(l)}) + ger\{x_0^{(l)} - x_0^{(l-1)}\}} \|b - A \times (x_0^{(l)} + z)\|_2 \tag{5.5}$$

Esse acréscimo no subespaço de procura da solução do problema de minimização pode levar a três situações distintas. O primeiro caso ocorre quando esse vetor gerador do subespaço extra está inserido no subespaço de Krylov. Neste caso, é gerado o mesmo subespaço, e o algoritmo pode continuar calculando o x_m da mesma forma.

Os dois outros casos estão relacionados a interferência que o vetor acrescentado causa na solução final. Eles podem gerar um subespaço diferente do subespaço de Krylov calculado. A diferença entre esses dois últimos casos está na soma realizada, se a soma é ou não direta. Se essa soma for direta, isso significa que a solução depende de todos os vetores do subespaço de Krylov, além do do vetor extra. Se essa soma não for direta, não podemos afirmar que o vetor extra do subespaço adicionado afeta a solução. Nesses dois casos, o problema final de minimização resolvido no algoritmo é diferente do original. Porque o vetor referente ao subespaço extra é acrescentado na base ortonormal V e reflete na matriz H. A matriz H também aumenta uma dimensão devido ao acréscimo do vetor na base. Ela pode se tornar uma matriz quadrada com dimensão $(m + 1)$ ou continuar de Hessenberg com dimensão $(m + 2) \times (m + 1)$.

O pseudocódigo do *Heavy Ball* GMRES está mostrado no Algoritmo 27.

Ao final do ciclo do GMRES original, nós temos a equação (5.6), onde V_k é a base do subespaço de Krylov de dimensão k e H_k é a matriz de Hessenberg do sistema linear auxiliar.

$$AV_k = V_k H_k + v_{k+1} h_{k+1} e_k^t = V_{k+1} \tilde{H}_k \tag{5.6}$$

O vetor do subespaço adicional que o *Heavy Ball* GMRES adiciona na solução (vetor d) é ortogonalizado e está definido na equação (5.7).

$$\tilde{d} = V_k^t d, g = d - V_k \tilde{d} \tag{5.7}$$

O vetor g definido pode ser ou não o vetor nulo. Caso o vetor g seja nulo, $d \in \kappa_k(A, r_0)$ e o problema de minimização final é o mesmo do GMRES reiniciado. No caso de $g \neq 0$, definimos a seguinte equação (5.8).

$$\tilde{v}_{k+1} = \frac{g}{\|g\|_2}, \tilde{V}_{k+1} = [V_k, \tilde{v}_{k+1}] \tag{5.8}$$

Depois, o método do *Heavy Ball* GMRES ortogonaliza o vetor $A\tilde{v_{k+1}}$ por V_{k+1} obtendo (5.9).

$$a = A\tilde{v}_{k+1}, \hat{a} = V_{k+1}^t a, h = a - V_{k+1}\hat{a} \tag{5.9}$$

No *Heavy Ball* GMRES qualquer $x - x_0 \in \kappa_k(A, r_0)$ pode ser expresso pela equação (5.10).

$$x - x_0 = \tilde{V}_{k+1} y, y \in \mathbb{R}^{k+1} \tag{5.10}$$

O vetor h, definido em (5.9), também pode ou não ser nulo. Caso ele não seja nulo, temos (5.11).

Algoritmo 27 Algoritmo *Heavy Ball* GMRES.

função HBGMRES(**Matriz de coeficientes:** A; **Vetor de valores independentes:** b; **Valor inicial:** x_0)

1: $l \leftarrow 1$
2: **enquanto** resíduo r_m não convergir **faça**
3: Linhas 2 a 12 do Algoritmo 23
4: **se** l igual a 1 **então**
5: $d \leftarrow x_0^l$
6: **senão**
7: $d \leftarrow x_0^l - x_0^{l-1}$
8: $\tilde{d} \leftarrow V_m^T \times d$; $g \leftarrow d - V_m \times \tilde{d}$
9: **se** $g \neq 0$ **então**
10: $\tilde{v}_{m+1} \leftarrow \frac{g}{\|g\|_2}$; $\tilde{V}_{m+1}[V_k \leftarrow \tilde{v}_{m+1}]$; $a \leftarrow A \times \tilde{v}_{m+1}$; $\hat{a} \leftarrow a$
11: **para** $i \leftarrow 1$ até $m+1$ **faça**
12: $h_{i,m+1} \leftarrow \langle \hat{a}, v_i \rangle$; $\hat{a} \leftarrow \hat{a} - h_{i,m+1} \times v_i$
13: $h_{m+2,m+1} \leftarrow \|\hat{a}\|_2$ ▷ A matriz H tem ordem $(m+2) \times (m+1)$ ou $(m+1)$
14: $he \leftarrow a - \tilde{V}_{m+1} \times \hat{a}$
15: **se** $he \neq 0$ **então**
16: /* calcular y que minimiza $\|\text{beta} \times e_1 - H \times y\|_2$ */ ▷ A Matriz H é Hessenberg com ordem $(m+2) \times (m+1)$
17: **senão**
18: /* calcular y que minimiza $\|\text{beta} \times e_1 - H \times y\|_2$ */ ▷ A Matriz H é quadrada com ordem $(m+1)$
19: $x_m \leftarrow x_0 + \tilde{V}_{m+1} \times y$
20: **senão**
21: /* calcular y que minimiza $\|\text{beta} \times e_1 - H \times y\|_2$ */
22: $x_m \leftarrow x_0 + V_m \times y$
23: $x_0^{l+1} \leftarrow x_m$
24: $l \leftarrow l + 1$
25: **retorna** x_m

$$\begin{aligned} Ax - b &= Ax_0 + A\tilde{V}_{k+1}y - b \\ &= (Ax_0 - b) + A\tilde{V}_{k+1}y \\ &= -r_0 + +\hat{V}_{k+2}\hat{H}_{k+1}y \\ &= -\|r_0\|_2\hat{V}_{k+2}e_1 + \hat{V}_{k+2}\hat{H}_{k+1}y \\ &= -\hat{V}_{k+2}[\hat{H}_{k+1}y = \|r_0\|_2 e_1] \end{aligned} \tag{5.11}$$

Assim, o problema de minimização se transforma em (5.12), que é um problema de mínimos quadrados cuja matriz de coeficientes é uma matriz de Hessenberg de ordem $(k+2) \times (k+1)$.

$$\min_y \|\hat{H}_{k+1}y - \|r_0\|_2 e_1\|_2 \tag{5.12}$$

Caso o vetor h (equação 5.9) seja nulo, temos que $a = A\tilde{v}_{k+1} = V_{k+1}\hat{a}$ em (5.9). Portanto, temos (5.13), com $\hat{H}_{k+1} \in \mathbb{R}^{(k+1)\times(k+1)}$.

$$\begin{aligned} A\tilde{V}_{k+1} &= [AV_k, A\tilde{v}_{k+1}] \\ &= [V_{k+1}\check{H}_k, V_{k+1}\hat{a}] \\ &= V_{k+1}[\check{H}_k\hat{a}] =: V_{k+1}\hat{H}_{k+1} \end{aligned} \tag{5.13}$$

Com essas relações, podemos obter a equação (5.14).

$$\begin{aligned} Ax - b &= Ax_0 + A\tilde{V}_{k+1}y - b \\ &= (Ax_0 - b) + A\tilde{V}_{k+1}y \\ &= -r_0 + +V_{k+1}\hat{H}_{k+1}y \end{aligned} \tag{5.14}$$

Portanto, o problema de minimização se transforma em (5.15), onde a matriz $\hat{H}_k + 1$ é uma matriz quadrada de ordem $(k + 1)$.

$$\min_y \|\hat{H}_{k+1}y - \|r_0\|_2 e_1\|_2 \tag{5.15}$$

As equações e definições utilizadas para descrever o método *Heavy Ball* GMRES foram retirados do artigo (IMAKURA; LI; ZHANG, 2016).

A Tabela 25 resume os métodos abordados, destacando as principais diferenças entre eles. O valor de M é o tamanho da matriz de coeficientes e o valor de m é menor ou igual a M.

Tabela 25 – Tabela sumário dos métodos estudados

	GMRES	GMRES(m)	FGMRES	αGMRES	HBGMRES
# de ciclos	1	vários	vários	vários	vários
Tamanho do ciclo	M	m fixo	m fixo	m_i variável	m fixo
Tamanho da matriz H	$(M+1) \times M$	$(m+1) \times m$	$(m+1) \times m$	$(m_i+1) \times m_i$	$(m+2) \times (m+1)$
Tamanho da matriz base (V ou Z)	$M \times M$	$M \times m$	$M \times m$	$M \times m_i$	$M \times (m+1)$
Histórico a cada reinicialização de ciclo	Perdido	Perdido	Perdido	Perdido	Acrescenta o subespaço do ciclo anterior no subespaço de procura do ciclo atual
Pré-condicionamento	Fora do método	Fora do método	Matriz M inserida na base	Fora do método	Fora do método

5.6 **Implementação de Funções Auxiliares**

Antes de mostrar as implementações dos algoritmos descritos anteriormente, algumas funções criadas para dividir o problema e organizar melhor o código são abordadas nesta seção. Todos os códigos foram implementados em linguagem C.

Norma
A primeira função implementada direto da sua fórmula é a que calcula a norma de um vetor. Sua fórmula está descrita na Definição 3 e o código está mostrado no Algoritmo 28.

Algoritmo 28 Norma vetorial.

função normaVetorial(**Vetor:** a; **Inteiro:** $size$)
1: $norma \leftarrow 0$
2: **para** $i \leftarrow 1$ até $size$ **faça**
3: $\quad norma \leftarrow norma + (a[i] * a[i])$
4: $norma \leftarrow \text{sqrt}(norma)$
5: **retorna** $norma$

Produto Interno
Outra função criada foi para calcular o produto interno entre dois vetores e seu código está mostrado no Algoritmo 29. Ele foi diretamente implementado da fórmula do produto interno, descrita na Definição 1.

Algoritmo 29 Produto interno entre dois vetores.

função produtoInternoVetorial(**Vetores:** a, b; **Inteiro:** $size$)
1: $result \leftarrow 0$
2: **para** $i \leftarrow 1$ até $size$ **faça**
3: $\quad result \leftarrow result + (a[i] * b[i])$
4: **retorna** $result$

A complexidade computacional das duas funções abordadas é $O(size)$, porque exite único *loop*, cujo tamanho é $size$ para cada uma das funções.

Multiplicação de matriz esparsa por um vetor
A próxima função multiplica a matriz esparsa A, no esquema CSR (2.2), por um vetor v. O esquema CSR foi explicado em detalhes na Seção 2.3.1 e o código implementado está mostrado no Algoritmo 30.

Algoritmo 30 Produto da matriz A armazenada em CSR por um vetor v.

função produtoMatrizEsparsaPorVetor(**Vetores:** AA, JA, IA, x)
1: **para** $i \leftarrow 1$ até N **faça**
2: $\quad k1 \leftarrow IA[i]$
3: $\quad k2 \leftarrow IA[i+1]$
4: $\quad$ **para** $j \leftarrow k1$ até $k2$ **faça**
5: $\quad\quad auxAA[j-k1+1] \leftarrow AA[j]$
6: $\quad\quad auxX[j-k1+1] \leftarrow x[JA[j]]$
7: $\quad y[i] \leftarrow$ produtoInternoVetorial($auxAA$, $auxX$, $k2-k1$)
8: **retorna** y

Essa função foi implementada a partir do código em FORTRAN encontrado em (SAAD, 2003). Ela foi adaptada para chamar a função que calcula o produto interno (Algoritmo 29). Sua complexidade computacional é $O(N \times nz)$, sendo nz a quantidade de

valores não-nulos da matriz. O *loop* mais interno percorre todos os valores não-nulos dentro das N iterações do *loop* externo.

Retrosubstituição

A última função criada para organizar o código resolve um sistema linear através da retro substituição. Esta função é usada para calcular o valor do vetor y do problema de minimização que aparece em todos os métodos. Seu código está apresentado no Algoritmo 31.

Algoritmo 31 Resolução de sistema linear $H\vec{x} = v\vec{e}c$ com a matriz H triangular superior.

função resolveSistemaTriangular(**Matriz:** H; **Vetor:** vec; **Inteiro:** $size$)
1: **para** $i \leftarrow size$ até 0 passo -1 **faça**
2: $y[i] \leftarrow vec[i]$
3: **para** $j \leftarrow i + 1$ até $size$ **faça**
4: $y[i] \leftarrow y[i] - (H[i][j] * y[j])$
5: $y[i] \leftarrow y[i]/H[i][i]$
6: **retorna** y

Esta função tem complexidade computacional de $O(size^2)$, uma vez que temos dois *loops* aninhados com até $size$ iterações.

Para implementar o problema de minimização da linha 14 do algoritmo do GMRES (Algoritmo 22), temos que achar o menor valor y_m que uma norma euclidiana pode ter. Como essa norma é sempre um número não-negativo, o menor valor possível é zero. Pela Definição 3, o único vetor que pode ter norma zero é o vetor nulo. Assim, o y_m que minimiza a norma apresentada é a solução do sistema linear $H \times y = \text{beta} \times e_1$.

Como a matriz H é uma matriz de Hessenberg, podemos utilizar matrizes de rotação para eliminar a diagonal abaixo da diagonal principal e transformar a matriz H em uma matriz triangular superior. As matrizes utilizadas são as rotações de Givens, definidas em (5.16).

$$G_i = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & c_i & s_i & \cdots & 0 \\ 0 & \cdots & -si & c_i & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 1 \end{bmatrix}, \text{ com } c_i = \frac{h_{i,i}}{\sqrt{h_{i,i}^2 + h_{i+1,i}^2}} \text{ e } s_i = \frac{hi+1,i}{\sqrt{h_{i,i}^2 + h_{i+1,i}^2}} \quad (5.16)$$

Cada matriz G_j elimina o valor $h_{j+1,j}$ da matriz de Hessenberg. Com isso, o sistema auxiliar final é dado por $\overline{H}_k \times y = \overline{e}$, com $\overline{H}_k = G_k \times G_{k-1} \times \cdots \times G_1 \times H$ e $\overline{e} = G_k \times G_{k-1} \times \cdots \times G_1 \times \|r_0\| \times e_1$.

Para poupar espaço e operações na implementação, não é necessário armazenar todas as matrizes G_i e nem realizar todos os cálculos ao final do *loop* principal do GMRES. A cada iteração do *loop* principal calculamos o c_i e o s_i da iteração, que são armazenados em dois vetores de tamanho m, além de atualizar os valores da coluna atual da matriz H. Esta resolução do sistema linear auxiliar foi apresentada em (SAAD; SCHULTZ, 1986).

5.7 Implementação do Método GMRES

O código implementado do GMRES está apresentado no Algoritmo 32. Nesta versão, assumimos como valor inicial o vetor nulo.

Algoritmo 32 Método GMRES (versão de implementação)

```
    função GMRES(Vetores: AA, JA, IA, b, x0)
 1: m ← MAX_M
 2: r0 ← produtoMatrizEsparsaVetor(AA, IA, JA, x0)
 3: r0 ← b − r0
 4: e[0] ← normaVetorial(r0, N)
 5: Vt[0] ← r0/e[1]
 6: tol ← e[1]
 7: para j ← 0 até m faça
 8:     w ← produtoMatrizEsparsaVetor(AA, IA, JA, Vt[j])
 9:     para i ← 0 até j faça
10:         H[i][j] ← produtoInternoVetorial(w, Vt[i], N)
11:         w ← w − (H[i][j] ∗ Vt[i])
12:     H[j + 1][j] ← normaVetorial(w, N)
13:     Vt[j + 1] ← w/H[j + 1][j]
14:     para k ← 0 até j − 1 faça
15:         aux ← H[k][j]
16:         H[k][j] ← c[k] ∗ aux + s[k] ∗ H[k + 1][j]
17:         H[k + 1][j] ← −s[k] ∗ aux + c[k] ∗ H[k + 1][j]
18:     ra ← sqrt((H[j][j] ∗ H[j][j]) + (H[j + 1][j] ∗ H[j + 1][j]))
19:     c[j] ← H[j][j]/ra
20:     s[j] ← H[j + 1][j]/ra
21:     H[j][j] ← ra
22:     H[j + 1][j] ← 0
23:     e[j + 1] ← −s[j] ∗ e[j]
24:     e[j] ← c[j] ∗ e[j]
25:     tol ← abs(e[j + 1])
26:     se tol < TOL então
27:         m ← j
28:         break
29: y ← resolverSistemaTriangular(H, e, m)
30: para i_Aux ← 0 até N faça
31:     xSol[i_Aux] ← x0[i_Aux]produtoInternoVetorial(V[i_Aux], y, m + 1)
32: retorna xSol
```

Neste algoritmo, as variáveis do Algoritmo 22 continuam com as mesmas definições, exceto a variável e, que agora é definida como um vetor. O vetor e armazena os valores independentes do sistema auxiliar, e $e[i]$ representa o valor na posição i deste vetor. As variáveis $c[i]$ e $s[i]$ são as posições i dos vetores c e s, respectivamente, que armazenam os valores de cada matriz G_i. A função chamada na linha 18 é uma função interna da linguagem C que calcula a raiz quadrada do valor que recebe como parâmetro. Como os computadores representam os números com uma quantidade finita de dígitos (PATTER-

SON; HENNESSY, 2014), eles podem não conseguir chegar no resultado final preciso e por isso foi acrescentado um valor de tolerância TOL para aceitarmos a solução.

Como podemos observar que, nesta versão, a cada iteração do GMRES, os valores da coluna j da matriz H são atualizados com os coeficientes das matrizes de Givens. Este procedimento economiza tempo de processamento. Para refletir as multiplicações no vetor $\vec{e}$ (valores independentes do sistema linear auxiliar), a cada iteração do GMRES, são atualizados os valores das linhas j e $j+1$.

As linhas 3, 5, 11 e 13 do algoritmo são traduzidas para a linguagem C como *loops* percorrendo todos os índices dos vetores em questão. Da mesma forma, o vetor $xSol$ é percorrido nas linhas 30-31. Sua complexidade computacional é $O(MAX\_M^2)$ ou $O(MAX\_M \times N))$. Isso se deve ao *loop* mais externo e a cada um dos *loops* internos. Como no GMRES a quantidade máxima de iterações é igual ao tamanho da matriz, sua complexidade então é $O(N^2)$.

5.7.1 Análise de convergência

A análise realizada foi também apresentada em (SAAD; SCHULTZ, 1986). Ela mostra que o método GMRES trata tanto matrizes positivas quanto negativas.

Assumir que os k primeiros vetores ortogonais podem ser construídos significa que os valores de $h_{j+1,j} \neq 0, j = 1, \ldots k$. De fato, se $h_{j+2,j+1} \neq 0$ quando considerada a norma do vetor w_{j+1}, o valor da diagonal $h_{j+1,j+1}$ depois de calcular todas as rotações de Givens satisfaz a equação (5.17).

$$h_{j+1,j+1} = (c_{j+1} \times h_{j+1,j+1} - s_{j+1} \times h_{j+2,j+1}) = \sqrt{h_{j+1,j+1}^2 + h_{j+2,j+1}^2} > 0 \qquad (5.17)$$

Portanto, os elementos diagonais da matriz H não se anulam. Assim, o sistema linear auxiliar baseado do problema de minimização $\|\|r_0\| \times e_1 - H \times y\|_2$ sempre tem solução. Dessa forma, o método GMRES tem solução se $h_{j+1,j} \neq 0, j = 1, \ldots, k$.

Considere que encontramos um valor $h_{j+1,j}$ igual a zero na iteração j. Com essa hipótese, o vetor v_{j+1} não pode ser construído. Porém, pelo algoritmo da ortogonalização, temos que $AV_j = V_j H$ mostrando que o subespaço de Krylov κ_j gerado por V_j é invariante. Se a matriz A é não-singular, a matriz H, cujo espectro é uma parte do espectro de A, também é não-singular.

Como a matriz H é não-singular, o mínimo da norma $\|\|r_0\| \times e_1 - H \times y\|_2$ é atingido quando $y = H^{-1} \times \text{beta} \times e_1$ tem norma zero. Assim, podemos calcular a única solução x_j aproximada.

Para provar que a solução x_j é exata e as soluções $x_i, i = 1, \ldots, j-1$ não são exatas, é necessário que resíduo r de x_j seja $r_j = 0$ e o das outras soluções sejam $r_i \neq 0, i = 1, \ldots, j-1$. Então, pelo Teorema 2 de (BAKER; JESSUP; MANTEUFFEL, 2005), temos que a norma de r_j é igual a $s_j \times e_{j-1}$, e que e_{j-1} é a norma de r_{j-1}. Como $r_{j-1} \neq 0$, s_j deve ser igual a zero, isso leva a $h_{j+1,j} = 0$ e o método pára.

5.8 **Implementação do Método GMRES reiniciado**

A implementação do GMRES reiniciado é apresentada no Algoritmo 33.

Para testar se o vetor $xSol$ convergiu, calculamos o vetor $bAux$, sendo o produto entre a matriz A e o vetor $xSol$. Depois, obtemos a norma da diferença entre $bAux$ e o vetor b

Algoritmo 33 Método GMRES reiniciado (versão de implementação).

função GMRESm(**Vetores:** AA, JA, IA, b, $x0$)
para $ciclo \leftarrow 1$ até MAX_CICLOS **faça**
    chamar Algoritmo 32
    para $i\_Aux \leftarrow 0$ até N **faça**
        $x0[i\_Aux] \leftarrow xSol[i\_Aux]$
    $count \leftarrow 0$
    $bAux \leftarrow$ produtoMatrizEsparsaVetor(AA, JA, IA, $xSol$)
    para $i\_Aux \leftarrow 0$ até N **faça**
        $bAux[i\_Aux] \leftarrow bAux[i\_Aux] - b[i\_Aux]$
    se normaVetorial($bAux$, N) $< TOL$ **então**
        break
retorna $xSol$

de valores independentes. A solução final convergiu quando essa norma é nula, dentro da tolerância computacional.

A complexidade computacional do GMRES reiniciado é dada por $O(MAX\_CICLOS \times MAX\_M^2)$ ou $O(MAX\_CICLOS \times MAX\_M \times N)$, adicionando a quantidade de passos do *loop* externo. A ideia principal do GMRES reiniciado é possuir uma quantidade de iterações por ciclo menor do que o tamanho da matriz. Por isso, a complexidade final deste método é $O(MAX\_CICLOS \times MAX\_M \times N)$.

5.8.1 Análise de convergência

É garantido que este método não irá parar porque quantidade MAX_M é pequena. Porém, sua convergência não é garantida em todos os casos. Existem instâncias que podem produzir normas residuais das soluções aproximadas x_m que não convergem para zero. Em (ELMAN, 1982), a convergência do Método do Resíduo Conjugado (GCR) é provada para matrizes reais positivas, e o método que o GMRES reiniciado foi baseado neste método. Assim, o GMRES reiniciado, também, converge para matrizes reais positivas.

Porém, para matrizes indefinidas, essa convergência não se estende. Considere como exemplo o problema $A\vec{x} = \vec{b}$ descrito em (5.18). Se executarmos o GMRES reiniciado, também conhecido como GMRES(m), com uma iteração por ciclo e assumirmos $x_0 = 0$, no segundo ciclo do GMRES(m) o algoritmo fica em estado de estagnação, porque x_1 também é igual a 0.

$$A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}, \vec{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{5.18}$$

Como a norma residual é minimizada a cada ciclo, ela não é crescente. Quando o valor de iterações por ciclo é suficientemente grande, o método GMRES(m) normalmente converge. Porém, como a intenção é economizar memória, não é vantajoso o GMRES(m) convergir somente com valores de m altos. No artigo (SAAD; SCHULTZ, 1986), é apresentado um limite inferior aproximado para o valor de m.

5.9 Implementação do Método GMRES flexível

O método GMRES flexível possui uma função a mais para calcular a matriz M pré-condicionadora. O código dessa função está mostrado no Algoritmo 34.

Algoritmo 34 Cálculo da matriz M pré-condicionadora para o método GMRES flexível.

função calcularMatrizM(**Vetores:** AA, JA, IA; **Ponteiros de vetor:** $*M\_AA$, $*M\_JA$, $*M\_IA$)
1: **para** $i \leftarrow 0$ até N **faça**
2: $k1 \leftarrow IA[i]$; $k2 \leftarrow IA[i+1]$
3: $tem\_valor \leftarrow 0$
4: **para** $j \leftarrow k1$ até $k2$ **faça**
5: **se** $JA[j] == i$ **então**
6: $tem\_valor \leftarrow 1$; $valor \leftarrow AA[j]$
7: **break**
8: **senão**
9: **se** $JA[j] > i$ **então**
10: **break**
11: $valor \leftarrow (tem\_valor)?\ 1/valor : 1$
12: $(*M\_AA)[i] \leftarrow valor$
13: $(*M\_JA)[i] \leftarrow i$
14: $(*M\_IA)[i] \leftarrow i$
15: $(*M\_IA)[i] \leftarrow i$

Esta função calcula a matriz M como sendo a diagonal inversa da matriz A. Sua complexidade computacional é igual a complexidade da função de multiplicação entre a matriz esparsa A e um vetor (Algoritmo 30). Assim, a complexidade é $O(N \times nz)$, sendo nz a quantidade de valores não nulos. Em um passo futuro, pretendemos modificar essa função para que a matriz M resultante seja diferente a cada ciclo do GMRES flexível. No Algoritmo 35 está o código do ciclo de cada iteração do GMRES flexível.

Algoritmo 35 Ciclo interno do GMRES flexível.

1: Linhas 1 a 6 do Algoritmo 32
2: **para** $j \leftarrow 0$ até m **faça**
3: calcularMatrizM(AA, IA, JA, $\&M\_AA$, $\&M\_JA$, $\&M\_IA$)
4: $w \leftarrow$ produtoMatrizEsparsaVetor(M_AA, M_IA, M_JA, $Vt[j]$)
5: $Zt[j] \leftarrow w$
6: $w \leftarrow$ produtoMatrizEsparsaVetor(AA, IA, JA, $Zt[j]$)
7: **para** $i \leftarrow 0$ até j **faça**
8: $H[i][j] \leftarrow$ produtoInternoVetorial(w, $Vt[i]$, N)
9: $w \leftarrow w - (H[i][j] * Vt[i])$
10: Linhas 12 a 28 do Algoritmo 32
11: $y \leftarrow$ resolverSistemaTriangular(H, e, m)
12: **para** $i\_Aux \leftarrow 0$ até N **faça**
13: $xSol[i_Aux] \leftarrow x0[i_Aux]$produtoInternoVetorial($Z[i\_Aux]$, y, $m+1$)

A complexidade computacional do ciclo interno do GMRES flexível é a maior função entre $O(MAX\_M \times N \times nz)$, $O(MAX\_M^2)$ e $O(MAX\_N \times N)$. Como o número máximo de iterações no GMRES flexível é menor do que o tamanho da matriz, então a complexidade do ciclo interno do GMRES flexível é $O(MAX\_M \times N \times nz)$.

5.9.1 Análise de convergência

A análise de convergência do método GMRES flexível é similar a análise de convergência do método GMRES (Seção 5.7).

A única diferença está na relação do algoritmo de ortogonalização que é $AZ_m = V_m H$ e não mais $AV_m = V_m H$. Assim, o subespaço de Krylov κ_m gerado por V_m não é mais invariante.

Essa diferença só acrescenta uma hipótese a mais para a convergência do GMRES flexível, a de que a matriz H também é não-singular.

O código do método GMRES flexível completo está apresentado no Algoritmo 36.

Algoritmo 36 Método GMRES flexível (versão de implementação)

```
    função FGMRES(Vetores: AA, JA, IA, b, x0)
 1: para ciclo ← 1 até MAX_CICLOS faça
 2:     chamar Algoritmo 35
 3:     para i_Aux ← 0 até N faça
 4:         x0[i_Aux] ← xSol[i_Aux]
 5:     count ← 0
 6:     bAux ← produtoMatrizEsparsaVetor(AA, JA, IA, xSol)
 7:     para i_Aux ← 0 até N faça
 8:         bAux[i_Aux] ← bAux[i_Aux] − b[i_Aux]
 9:     se normaVetorial(bAux, N) < TOL então
10:         break
11: retorna xSol
```

A complexidade computacional do método GMRES flexível é $O(MAX\_CICLOS \times MAX\_M \times N \times nz)$.

5.10 Método αGMRES

O Algoritmo 37 apresenta o código implementado do αGMRES. Ele modifica o início e o final do ciclo do GMRES inicial, para calcular a quantidade de iterações de cada ciclo.

O cálculo da quantidade de iterações por ciclo (Algoritmo 26) tem complexidade constante ($O(1)$) e não interfere na complexidade computacional do método αGMRES. A complexidade final do método é dada pela maior função entre $O(MAX\_CICLOS \times MAX\_M^2)$, $O(MAX\_CICLOS \times MAX\_M \times N)$ e $O(MAX\_CICLOS \times N)$. Como a quantidade máxima de interações é menor que o tamanho da matriz, então a complexidade do método αGMRES é dada por $O(MAX\_CICLOS \times MAX\_M \times N)$.

5.10.1 Análise de convergência

No artigo (BAKER; JESSUP; KOLEV, 2009), a análise de convergência do αGMRES é apresentada formalmente apenas para matrizes simétricas. Porém, através de experi-

Algoritmo 37 Algoritmo αGMRES (versão de implementação)

função αGMRES(**Vetores:** AA**,** JA**,** IA**,** b**,** $x0$; **Inteiros:** m_{min}**,** m_{max})
1: $cr \leftarrow 1$; $max\_cr \leftarrow \cos 8$; $min\_cr \leftarrow \cos 80$; $d \leftarrow 3$; $l \leftarrow 0$
2: $r0\_atual \leftarrow b - A \times x_0$
3: **para** $ciclo \leftarrow 1$ até MAX_CICLOS **faça**
4: calcular m (Algoritmo 26)
5: Linhas 4-31 do Algoritmo 32
6: $x0 \leftarrow xSol$
7: $r0\_anterior \leftarrow r0\_atual$
8: $r0\_atual \leftarrow b - A \times x_0$
9: $cr \leftarrow$ normaVetorial$(r0_atual, N)$/normaVetorial$(r0_anterior, N)$
10: $count \leftarrow 0$
11: $bAux \leftarrow$ produtoMatrizEsparsaVetor$(AA, JA, IA, xSol)$
12: $bAux \leftarrow bAux - b$
13: **se** normaVetorial$(bAux, N) < TOL$ **então**
14: **break**
15: **retorna** x_m

mentos numéricos é constatada a mesma convergência para matrizes não-simétricas.

A ideia deste método é diminuir o valor de m sempre que possível e evitar o comportamento repetitivo do GMRES reiniciado (GMRES(m)). Em (BAKER; JESSUP; MANTEUFFEL, 2005), foi observado que os vetores residuais do GMRES(m) de dois ciclos não sequenciais frequentemente apontam para a mesma direção. Por exemplo, os ciclos $i-1$, $i+1$ podem apresentar os resíduos r_{i-1}, r_{i+1} sendo $r_{i+1} \approx \sigma r_{i-1}$, onde $\sigma \leq 1$. Este é o chamado comportamento repetitivo.

Para provar que esse comportamento ocorre com qualquer valor de m não crescente para qualquer matriz A simétrica ou antissimétrica, supomos ϕ_i como o ângulo entre r_{i+1} e r_{i-1} e temos que (5.19) é o valor do cosseno desse ângulo.

$$\cos \phi_i = \frac{\langle r_{i+1}, r_{i-1} \rangle}{\|r_{i+1}\|_2 \|r_{i-1}\|_2} \tag{5.19}$$

A fórmula (5.19) não tem valor definido para quando o resíduo de r_{i+1} ou de r_{i-1} é igual a zero. Porém, o resíduo nulo nos diz que chegamos a solução aproximada final, sem necessidade de continuar as iterações. Assim, existe a garantia de que a fórmula (5.19) sempre funciona.

Como a solução aproximada do GMRES(m) é $x_m \in x_0 + \kappa_k(A, r_0)$, podemos escrever o resíduo r_i em termos de um polinomial de $A \times r_{i-1}$, como em (5.20), onde α_{ik} é um valor escolhido de forma que a norma de r_i é minimizada, que $r_i \perp A\kappa_m(A, r_{i-1})$ (SAAD, 2003).

$$r_i = r_{i-1} - \sum_{k=1}^{m} \alpha_{ik} A^k r_{i-1} \tag{5.20}$$

Como o valor de m_i pode variar a cada ciclo, definimos r_{i+1} (5.21). Calculando o produto interno de r_{i+1} a esquerda nessa equação e com a relação $r_{i+1} \perp A\kappa_m(A, r_i)$, temos (5.22).

$$r_{i+1} = r_i - \sum_{k=1}^{m_{i+1}} \alpha_{(i+1)k} A^k r_i \tag{5.21}$$

$$\langle r_{i+1}, r_{i+1} \rangle = \langle r_{i+1}, r_i - \sum_{k=1}^{m_{i+1}} \alpha_{(i+1)k} A^k r_i \rangle \Rightarrow \langle r_{i+1}, r_{i+1} \rangle = \langle r_{i+1}, r_i \rangle \tag{5.22}$$

Similarmente, calculamos o produto interno de r_{i-1} à direita na equação (5.21) e obtemos (5.23).

$$\langle r_{i+1}, r_{i-1} \rangle = \langle r_i - \sum_{k=1}^{m_{i+1}} \alpha_{(i+1)k} A^k r_i, r_{i-1} \rangle \tag{5.23}$$

Como a matriz A é simétrica (ou antissimétrica) por hipótese e o valor de m_i é não crescente, isso significa que $m_{i+1} \leq m_i$, e a equação (5.23) é reduzida para (5.24).

$$\langle r_{i+1}, r_{i-1} \rangle = \langle r_i, r_{i-1} \rangle - \sum_{k=m_i+1}^{m_{i+1}} \langle \alpha_{(i+1)k} A^k r_i, r_{i-1} \rangle \Rightarrow \langle r_{i+1}, r_{i-1} \rangle = \langle r_i, r_{i-1} \rangle \tag{5.24}$$

Assim, das equações (5.22) e (5.24), temos que $\langle r_{i+1}, r_{i-1} \rangle = \langle r_i, r_i \rangle$ e usando a (5.19), podemos escrever (5.25).

$$\cos \phi_i = \frac{\|r_i\|_2^2}{\|r_{i+1}\|_2 \|r_{i-1}\|_2} \tag{5.25}$$

Sendo a_i a sequência $a_i = \|r_{i+1}\|_2 \ / \ \|r_i\|_2$, podemos reescrever (5.25) como (5.26). Como $\cos \phi_i \leq 1$, temos que $a_{i-1} \leq a_i$. Portanto, a sequência a_i é crescente monótona e, por construção a sequência tem um limite inferior, pois $\|r_{i+1}\|_2 < \|r_i\|_2$.

$$\cos \phi_i = \frac{a_{i-1}}{a_i} \tag{5.26}$$

Como resultado, existe um valor β que a_i tende quando pegamos um i grande ($a_i \to \beta$ quando $i \to \infty$), o que implica em (5.27), mostrando que os resíduos alternados se aproximam de zero.

Essa demonstração mostra que essa alternância é um processo limitador do GMRES(m) no caso de um m fixo ou não crescente.

$$\cos \phi_i = \frac{\beta}{\beta} = 1 \tag{5.27}$$

Apesar da prova anterior englobar apenas as matrizes simétricas e antissimétricas, essa alternância também é observada para matrizes não simétricas (BAKER; JESSUP; MANTEUFFEL, 2005). O αGMRES tenta interromper essa alternância através da taxa de convergência dos resíduos sequenciais. Assim, esse monitoramento do αGMRES previne o método de ter um desempenho pior que o GMRES(m) para os problemas que o GMRES(m) converge.

5.11 **Método *Heavy Ball* GMRES**

O último código implementado é o *Heavy Ball* GMRES. Este código lida com as três possibilidades do problema de otimização que aparecem no Algoritmo 27. Para

isso, a primeira parte do algoritmo continua igual ao GMRES reiniciado (Algoritmo 33), utilizando as rotações de Givens para eliminar os valores da matriz H das posições i,j onde $i = j + 1$.

Caso o vetor g resulte no vetor nulo, o problema de minimização é o mesmo do GMRES reiniciado (Algoritmo 33), e basta calcular o valor de y.

Caso o vetor g não seja o vetor nulo, adicionamos o vetor g normalizado na base V do novo subespaço de procura e calculamos a nova coluna $m + 1$ da matriz H ao ortogonalizar o vetor $a = A \times Vt[m+1]$. Se a base V não é ortogonal (quando $he \neq 0$), a matriz H continua de Hessenberg e usamos mais uma matriz de rotação de Givens para eliminar o valor de $h_{m+2,m+1}$. Se a base V é ortogonal (quando $he = 0$), a matriz H é quadrada. Como já eliminamos os valores das posições i,j onde $i = j + 1$, temos uma matriz triangular superior. Portanto, não é necessário fazer nenhuma operação a mais no sistema linear auxiliar quando $he = 0$. Para achar o valor de y nestes dois casos extras do método *Heavy Ball* GMRES, basta fazer a retrosubstituição começando da posição $m+1$ em vez da m.

A versão completa de implementação do *Heavy Ball* GMRES está mostrada no Algoritmo 39.

Algoritmo 38 Parte do código do *Heavy Ball* GMRES.

```
1: Vt[m + 1] ← g/norma
2: a ← produtoMatrizEsparsaVetor(AA, JA, IA, xSol)
3: aHat ← a
4: para i ← 0 até m + 1 faça
5:     H[i][m + 1] ← produtoInternoVetorial(aHat, Vt[i], N)
6:     aHat ← aHat − (H[i][m + 1] ∗ Vt[i])
7: H[m + 2][m + 1] ← normaVetorial(aHat, N)
8: para i_Aux ← 0 até N faça
9:     he[i_Aux] ← a[i_Aux]− produtoInternoVetorial(V[iAux], aHat, m + 2)
10: se fabs(normaVetorial(he, N)) > TOL então
11:     para k ← 0 até m faça
12:         aux ← H[k][m + 1]
13:         H[k][m + 1] ← c[k] ∗ aux + s[k] ∗ H[k + 1][m + 1]
14:         H[k + 1][m + 1] ← −s[k] ∗ aux + c[k] ∗ H[k + 1][m + 1]
15:     ra ← sqrt((H[m+1][m+1]∗H[m+1][m+1])+(H[m+2][m+1]∗H[m+2][m+1]))
16:     c[m + 1] ← H[m + 1][m + 1]/ra
17:     s[m + 1] ← H[m + 2][m + 1]/ra
18:     H[m + 1][m + 1] ← ra
19:     H[m + 2][m + 1] ← 0
20:     e[m + 2] ← −s[m + 1] ∗ e[m + 1]
21:     e[m + 1] ← c[m + 1] ∗ e[m + 1]
22: y ← resolverSistemaTriangular(H, e, m + 1)
23: para i_Aux ← 0 até N faça
24:     xSol[i_Aux] ← x0_atual[i_Aux]+ produtoInternoVetorial(V[iAux], y, m + 2)
```

Algoritmo 39 Algoritmo *Heavy Ball* GMRES (versão de implementação).

função HBGMRES(**Vetores:** AA, JA, IA, b, $x0\_atual$)
1: $l \leftarrow 1$
2: **para** $ciclo \leftarrow 1$ até MAX_CICLOS **faça**
3: Linhas 1 a 28 do Algoritmo 32, tendo $x0$ como $x0\_atual$
4: **se** l **então**
5: $d \leftarrow x_0^l$; $l \leftarrow 0$
6: **senão**
7: $d \leftarrow x0\_atual - x0\_anterior$
8: **para** $i\_Aux \leftarrow 0$ até N **faça**
9: $dTil[i\_Aux] \leftarrow$ produtoInternoVetorial($Vt[i_Aux]$, d, N)
10: **para** $i\_Aux \leftarrow 0$ até N **faça**
11: $g[i\_Aux] \leftarrow d[i\_Aux]-$ produtoInternoVetorial($V[i_Aux]$, $dTil$, $m+1$)
12: $norma \leftarrow$ normaVetorial(g, N)
13: **se** fabs($norma$) $> TOL$ **então**
14: Algoritmo 38
15: **senão**
16: $y \leftarrow$ resolverSistemaTriangular(H, e, m)
17: **para** $i\_Aux \leftarrow 0$ até N **faça**
18: $xSol[i\_Aux] \leftarrow x0\_atual[i\_Aux]+$ produtoInternoVetorial($V[i_Aux]$, y, $m+1$)
19: $x0\_anterior \leftarrow x0\_atual$
20: $x0\_atual \leftarrow xSol$
21: Linha 5-10 do Algoritmo 33
22: **retorna** $xSol$

5.11.1 Análise de convergência

De acordo com (IMAKURA; LI; ZHANG, 2016), a convergência do *Heavy Ball* GMRES (HBGMRES) é melhor ou igual a convergência do GMRES reiniciado. O HBGMRES procura sua solução aproximada dentro de subespaços que contêm o subespaço de Krylov $\kappa_k(A, r_0^{(l)})$. Por isso, a minimização deste método é sobre todos os polinômios p_k de grau k ou menor tal que $p_k(0) = 1$, como mostra (5.28).

$$\|r_k^{(l)}\|_2 \leq \min_{p_k(0)=1} \|p_k(A) r_0^{(l)}\|_2 \tag{5.28}$$

O lado direito de (5.28) é a norma residual do ciclo do GMRES de tamanho k, com valor inicial $x_0^{(l)}$. Essa relação mostra que cada ciclo do HBGMRES tem convergência melhor ou igual a um ciclo do GMRES reiniciado de tamanho k.

5.12 Experimentos e análise dos resultados

Todos os códigos foram desenvolvidos em Linguagem C, e compilados com o compilador GCC versão 5.4.0 no Sistema Operacional Ubuntu versão 16.04. As matrizes usadas para teste foram encontradas no *Matrix Market* (BOISVERT et al., 1999), um repositório com diversas matrizes esparsas disponíveis para testes, e no antigo repositório da Universidade da Flórida (DAVIS; HU, 2011), hoje chamado de *SuiteSparse Matrix Collection*

(<https://sparse.tamu.edu>).

Foram escolhidas 15 matrizes desses dois repositórios, as quais todas tinham valores para o vetor b para achar a solução do sistema linear. Todas essas matrizes possuem números reais, não são positivas definidas e não podem ser decompostas na fatoração Cholesky. O nome e as características das matrizes escolhidas estão mostrados na Tabela 26. A coluna Fonte indica o *site* do qual a matriz foi retirada. Nessa coluna utilizamos a sigla MM para indicar o *site Matrix Market* e a sigla SS para o *SuiteSparse*.

Tabela 26 – Matrizes esparsas escolhidas para os testes e suas características.

Id	Nome	Fonte	N	nz	Aplicação
1	$add20$	MM	2395	17319	*Design* de componente de computador
2	$cavity05$	SS	1182	32747	Dinâmica dos fluidos computacional
3	$cavity10$	SS	2597	76367	Dinâmica dos fluidos computacional
4	$chipcool0$	SS	20082	28150	Problema de modelo de redução
5	$circuit\_2$	SS	4510	21199	Simulação de circuito
6	$comsol$	SS	1500	97645	Problema estrutural
7	$flowmeter5$	SS	9669	67391	Problema de modelo de redução
8	$fpga\_trans\_01$	SS	1220	7382	Simulação de circuito
9	$fpga\_trans\_02$	SS	1220	7382	Simulação de circuito
10	$matrix - new\_3$	SS	125329	893894	Dispositivo semicondutor
11	$memplus$	SS	17758	126150	Simulação de circuito
12	$raefsky1$	SS	3242	294276	Dinâmica dos fluidos computacional
13	$raefsky2$	SS	3242	194276	Dinâmica dos fluidos computacional
14	$sherman4$	MM	1104	3786	Simulação de reservatório de petróleo
15	$wang3$	SS	26064	177168	Dispositivo semicondutor

A Tabela 27 mostra o número de condicionamento, a norma e o mínimo valor singular de cada matriz. Estes dados foram coletados no *SuiteSparse Matrix Collection*, na página que podemos obter cada matriz escolhida.

A matriz 10, $matrix - new\_3$, é a única que não possui essas estatísticas da matriz no *site* do repositório. Por isso, ela não foi mostrada na Tabela 27.

Como o GMRES é um método direto, todos os resultados obtidos através deste método são usados como referência nas comparações. As comparações são realizadas considerando os outros quatro métodos: o GMRES reiniciado, o GMRES flexível, o αGMRES e o *Heavy Ball* GMRES.

Para os testes desses últimos quatro métodos, o valor escolhido para MAX_CICLOS foi de 3000, valor também escolhido em (IMAKURA; LI; ZHANG, 2016), e a cada ciclo o valor de MAX_M é 30, pois é o tamanho comum do ciclo encontrado na literatura (SAAD, 1993; BAKER; JESSUP; KOLEV, 2009). Como o *Heavy Ball* GMRES adiciona um vetor a cada ciclo interno, o valor de MAX_M escolhido para esse método é 29. Assim, todos os métodos procuram sua solução em um subespaço de tamanho 30.

5.12.1 Precisão dos resultados

O primeiro experimento considerou as 15 matrizes executando os cinco métodos para observar a qualidade dos resultados obtidos. Para isso, contabilizamos quantos valores do vetor b_Aux final são próximos ao b original. Além dessa quantidade absoluta, con-

Tabela 27 – As principais características das matrizes de teste.

Id	Esparsidade (%)	Número de condicionamento	Norma da matriz	Mínimo valor singular
1	0,3019	$1{,}204710 \times 10^{4}$	$7{,}221442 \times 10^{-1}$	$5{,}994340 \times 10^{-5}$
2	2,3439	$5{,}770648 \times 10^{5}$	$1{,}300724 \times 10^{1}$	$2{,}254035 \times 10^{-5}$
3	1,1323	$2{,}955066 \times 10^{6}$	$1{,}305274 \times 10^{1}$	$4{,}417072 \times 10^{-6}$
4	0,0697	$8{,}330854 \times 10^{6}$	$3{,}144574 \times 10^{-1}$	$3{,}774612 \times 10^{-8}$
5	0,1042	$1{,}319253 \times 10^{5}$	$2{,}638701 \times 10^{1}$	$2{,}000148 \times 10^{-4}$
6	4,3398	$1{,}415217 \times 10^{6}$	$6{,}328481 \times 10^{-1}$	$4{,}471738 \times 10^{-7}$
7	0,0721	$7{,}105826 \times 10^{6}$	$2{,}088514 \times 10^{3}$	$2{,}939158 \times 10^{-4}$
8	0,4960	$1{,}221426 \times 10^{4}$	$2{,}887701 \times 10^{0}$	$2{,}364205 \times 10^{-4}$
9	0,4960	$1{,}224937 \times 10^{4}$	$2{,}887701 \times 10^{0}$	$2{,}357427 \times 10^{-4}$
10	0,0057	-	-	-
11	0,0400	$1{,}294355 \times 10^{5}$	$1{,}494846 \times 10^{0}$	$1{,}154896 \times 10^{-5}$
12	2,7998	$1{,}288509 \times 10^{4}$	$3{,}709513 \times 10^{0}$	$2{,}878919 \times 10^{-4}$
13	2,7998	$4{,}251937 \times 10^{3}$	$3{,}724811 \times 10^{0}$	$8{,}760269 \times 10^{-4}$
14	0,3106	$2{,}178631 \times 10^{3}$	$6{,}650889 \times 10^{1}$	$3{,}052784 \times 10^{-2}$
15	0,0261	$6{,}188121 \times 10^{3}$	$2{,}680075 \times 10^{-1}$	$4{,}331000 \times 10^{-5}$

sideramos, também, o valor da norma da diferença entre esses dois vetores. As próximas tabelas apresentam os resultados referentes a qualidade da solução de cada método.

A Tabela 28 mostra o tamanho N da matriz, a quantidade de valores corretos da solução e a norma da diferença para o método GMRES.

Tabela 28 – Precisão dos resultados do método GMRES com tolerância de 10^{-15}.

Id	N	# de valores corretos	Norma vetorial do erro
1	2395	2395	$9{,}603484 \times 10^{-16}$
2	1182	1182	$9{,}374152 \times 10^{-16}$
3	2597	2597	$9{,}559325 \times 10^{-16}$
4	20082	20082	$2{,}700546 \times 10^{-15}$
5	4510	178	$1{,}876379 \times 10^{-12}$
6	1500	0	$1{,}628296 \times 10^{-09}$
7	9669	262	$7{,}498319 \times 10^{-09}$
8	1220	1220	$7{,}011582 \times 10^{-16}$
9	1220	1220	$9{,}945542 \times 10^{-16}$
10	125329	Erro de memória	Erro de memória
11	17758	17758	$9{,}596166 \times 10^{-16}$
12	3242	3242	$8{,}080599 \times 10^{-16}$
13	3242	3242	$9{,}286792 \times 10^{-16}$
14	1104	558	$2{,}161496 \times 10^{-10}$
15	26064	34	$5{,}147919 \times 10^{-08}$

A Tabela 28 mostra que para a matriz $matrix - new\_3$ (Id 1) não foi possível alocar todos os vetores e matrizes necessárias para calcular sua solução. Esse problema de memória é o que os métodos iterativos baseados no GMRES procuram resolver.

Note que aproximadamente metade das matrizes não convergiram com N iterações.

Esse é o número máximo de iterações que o GMRES faz para cada matriz. Pela Tabela 27 observamos que as matrizes que não convergem no GMRES *circuit_2*, *comsol*, $flowmeter5$, $matrix-new\_3$, $sherman4$ e $wang3$ (Ids 5, 6, 7, 10, 14 e 15) são as aquelas com um condicionamento ruim, além do valor da norma distante do mínimo valor singular. Essa distância entre os valores das duas últimas características pode gerar diversos erros de aproximação e truncamentos nas operações de divisão e multiplicação do método.

A Tabela 29 mostra a qualidade da solução obtida no método GMRES reiniciado. Esta tabela possui três colunas. Uma com a quantidade de ciclos, outra com a quantidade de valores corretos e a terceira com a norma da diferença de $\vec{b\_aux}$ por $\vec{b}$.

Tabela 29 – Precisão dos resultados do método GMRES reiniciado com tolerância de 10^{-15}.

Id	# ciclos	# valores corretos	Norma vetorial do erro
1	12	2395	$1{,}422667 \times 10^{-15}$
2	13	1182	$7{,}755597 \times 10^{-15}$
3	204	2597	$1{,}398342 \times 10^{-14}$
4	3000	1	$7{,}304903 \times 10^{-08}$
5	51	4510	$4{,}375845 \times 10^{-15}$
6	3000	0	$5{,}349684 \times 10^{-05}$
7	3000	0	$7{,}468860 \times 10^{-01}$
8	166	1220	$2{,}982472 \times 10^{-15}$
9	176	1220	$2{,}076719 \times 10^{-15}$
10	3000	116344	$9{,}752802 \times 10^{-12}$
11	19	17758	$1{,}724587 \times 10^{-15}$
12	208	3242	$1{,}278717 \times 10^{-14}$
13	147	3242	$1{,}408298 \times 10^{-14}$
14	3000	566	$2{,}100190 \times 10^{-12}$
15	3000	3195	$3{,}844191 \times 10^{-10}$

Como a quantidade de iterações por ciclo é igual a 30 iterações, todas as matrizes de teste escolhidas obtiveram algum resultado com o GMRES reiniciado.

A quantidade de matrizes que convergiram e as que não convergiram no GMRES reiniciado é a mesma que as observadas no método direto GMRES, nove matrizes convergem e seis não convergem. Porém, duas matrizes tiveram resultados diferentes entre os dois métodos e são apresentadas a seguir.

A matriz *chipcool*0 (Id 4) que converge no método GMRES, não convergiu no método GMRES reiniciado. O motivo pode ser justamente essa perda do histórico ao reinicializar o ciclo do GMRES neste método. Já a matriz *circuit_2* (Id 5) não converge com o método GMRES, mas converge com o método GMRES reiniciado. Um dos motivos para isso é o menor número de iterações por ciclo do GMRES reiniciado. Os possíveis erros de arredondamento e truncamento não são propagados da mesma forma que no GMRES.

Das matrizes que não convergiram nos dois métodos, as matrizes *comsol* (Id 6) e $flowmeter5$ (Id 7) apresentaram uma solução pior ao final dos 3000 ciclos. As matrizes $sherman4$ e $wang3$ (Ids 14 e 15) apresentaram soluções melhores ao final do limite de ciclos definido. As matrizes 6 e 7 tem um condicionamento ruim ($1{,}41521 \times 10^6$ e $7{,}105826 \times 10^6$, respectivamente) enquanto que as matrizes 14 e 15 tem um condicionamento melhor ($2{,}178631 \times 10^3$ e $6{,}188121 \times 10^3$, respectivamente).

Quando a quantidade de iterações por ciclo diminui, as matrizes com mal condicionamento precisam de mais ciclos para convergir, visto que ao final de cada ciclo o histórico é perdido. Por isso as matrizes *comsol* e $flowmeter5$ (Ids 6 e 7) apresentaram uma solução pior no GMRES reiniciado do que no GMRES. As matrizes *sherman*4 e *wang*3 (Ids 14 e 15) tem um condicionamento melhor e a distância entre os valores da norma e do mínimo valor singular de cada matriz não é grande. Com isso, a perda do histórico a cada poucas iterações afetou positivamente a solução final apresentada.

A Tabela 30 mostra os resultados obtidos para o método GMRES flexível com a tolerância computacional de 10^{-15}.

Tabela 30 – Precisão dos resultados do método GMRES flexível com tolerância de 10^{-15}.

Id	# ciclos	# valores corretos	Norma vetorial do erro
1	3	2395	$1{,}299340 \times 10^{-15}$
2	7	1182	$7{,}602842 \times 10^{-15}$
3	78	2597	$1{,}443427 \times 10^{-14}$
4	50	20082	$1{,}039664 \times 10^{-14}$
5	14	4510	$3{,}008970 \times 10^{-15}$
6	3000	0	$1{,}740398 \times 10^{-03}$
7	3000	8714	$6{,}214747 \times 10^{-13}$
8	220	1220	$3{,}046258 \times 10^{-15}$
9	170	1220	$1{,}453658 \times 10^{-15}$
10	3000	103974	$7{,}488915 \times 10^{-05}$
11	2	17758	$2{,}344306 \times 10^{-15}$
12	262	3242	$1{,}669797 \times 10^{-14}$
13	147	3242	$1{,}423482 \times 10^{-14}$
14	3000	563	$2{,}169959 \times 10^{-12}$
15	3000	3273	$3{,}883184 \times 10^{-10}$

No método GMRES flexível obtivemos uma matriz a mais convergindo do que no método GMRES reiniciado. A matriz *chipcool*0 (Id 4) é aquela com o maior número de condicionamento de todas ($8{,}330854 \times 10^6$). Quando usamos o pré-condicionador da diagonal inversa, diminuímos esse valor. Por isso esta matriz converge com o método do GMRES flexível e não converge com o GMRES reiniciado.

Dentre as nove matrizes que convergem no GMRES flexível, seis matrizes diminuem a quantidade de ciclos, duas matrizes aumentam a quantidade de ciclos e uma possui a mesma quantidade de ciclos para chegarem na solução final. Essa última é a matriz $raefsky2$ (Id 13). As duas que aumentam a quantidade de ciclos são as matrizes $fpga\_trans\_01$ e $raefsky1$ (Ids 8 e 12). As outras seis matrizes que convergem no GMRES flexível com menos ciclos que no GMRES reiniciado têm o comportamento esperado quando usamos uma matriz precondicionadora no sistema linear, justamente o de diminuir a quantidade de ciclos necessários para a convergência.

As cinco matrizes de teste restantes não convergem no método GMRES flexível. Duas tiveram uma solução final melhor que no método GMRES reiniciado e as outras três apresentaram uma solução pior. As duas matrizes que melhoraram a quantidade de ciclos foram as $flowmeter5$ e a *wang*3 (Ids 7 e 15). As três matrizes que aumentaram a quantidade de ciclos no GMRES flexível foram as matrizes *comsol*, *sherman*4 e *wang*3 (Ids 6, 14 e 15).

Como na nossa implementação do GMRES flexível a matriz pré condicionadora M é sempre a mesma, as matrizes nas quais os maiores valores não estão na diagonal principal não obtém vantagem no pré condicionamento.

A Tabela 31 mostra os resultados obtidos para o método αGMRES com a tolerância computacional de 10^{-15}.

Tabela 31 – Precisão dos resultados do método αGMRES com tolerância de 10^{-15}.

Id	# ciclos	# valores corretos	Norma vetorial do erro
1	14	2395	$7{,}708828 \times 10^{-16}$
2	18	1182	$7{,}470137 \times 10^{-15}$
3	204	2597	$1{,}485172 \times 10^{-14}$
4	3000	0	$6{,}959102 \times 10^{-08}$
5	61	4510	$3{,}028923 \times 10^{-15}$
6	3000	0	$4{,}662471 \times 10^{-06}$
7	3000	0	$7{,}430300 \times 10^{-01}$
8	111	1220	$2{,}333213 \times 10^{-15}$
9	88	1220	$1{,}314144 \times 10^{-15}$
10	3000	114547	$2{,}226391 \times 10^{-11}$
11	23	17758	$1{,}149071 \times 10^{-14}$
12	110	3242	$2{,}032995 \times 10^{-14}$
13	192	3242	$1{,}467992 \times 10^{-14}$
14	3000	565	$1{,}983679 \times 10^{-12}$
15	3000	3347	$3{,}800995 \times 10^{-10}$

Podemos observar na Tabela 31 que as matrizes que convergem e as que não convergem são as mesmas do método GMRES reiniciado. Assim, nove matrizes convergem e seis não convergem.

Das nove matrizes que convergem, somente uma permanece com a mesma quantidade de ciclos que no GMRES reiniciado, a matriz *cavity*10 (Id 3). Apenas três matrizes de teste diminuíram a quantidade de ciclos para chegar a solução convergente. Essas três foram as matrizes a $fpga\_trans\_01$, a $fpga\_trans\_02$ e a $raefsky1$ (Ids 8, 9 e 12). As outras cinco matrizes que convergem no αGMRES aumentaram a quantidade de ciclos necessária para chegar na solução aproximada. Estas matrizes foram as matrizes *add*20, *cavity*05, $circuit\_2$, *memplus* e $raefsky2$ (Ids 1, 2, 5, 11 e 13).

As outras seis matrizes não convergem no αGMRES. As matrizes *comsol*, $flowmeter5$ e *wang*3 (Ids 6, 7 e 15) tem uma melhora na solução apresentada ao final dos 3000 ciclos. Enquanto as matrizes *chipcool*0, $matrix-new\_3$ e *sherman*4 (Ids 4, 10 e 14) obtiveram uma solução pior ao final se comparado ao método GMRES reiniciado.

Esta última tabela (Tabela 32) mostra os resultados obtidos no método *Heavy Ball* GMRES com a mesma tolerância de 10^{-15}.

As mesmas matrizes que convergem no GMRES reiniciado convergem no *Heavy Ball* GMRES. Assim como as matrizes que não convergem no primeiro método também não convergem no *Heavy Ball* GMRES.

A matriz *add*20 (Id 1) é a única que converge no *Heavy Ball* GMRES com o mesmo número de ciclos que no GMRES reiniciado. Outra matriz a $fpga\_trans\_01$ (Id 8) converge com mais iterações no HBGMRES do que no GMRES reiniciado. As outras sete matrizes que convergem nos dois métodos reduzem a quantidade de ciclos no *Heavy Ball*

Tabela 32 – Precisão dos resultados do método *Heavy Ball* GMRES com tolerância de 10^{-15}.

Id	# ciclos	# valores corretos	Norma vetorial do erro
1	12	2395	$1{,}413690 \times 10^{-15}$
2	12	1182	$7{,}746546 \times 10^{-15}$
3	119	2597	$1{,}319151 \times 10^{-14}$
4	3000	0	$6{,}600168 \times 10^{-07}$
5	40	4510	$2{,}049329 \times 10^{-15}$
6	3000	0	$3{,}364343 \times 10^{-05}$
7	3000	0	$7{,}689120 \times 10^{-01}$
8	241	1220	$9{,}950931 \times 10^{-16}$
9	88	1220	$2{,}219824 \times 10^{-15}$
10	3000	115336	$1{,}699175 \times 10^{-11}$
11	18	17758	$2{,}404009 \times 10^{-14}$
12	151	3242	$1{,}658147 \times 10^{-14}$
13	146	3242	$1{,}314313 \times 10^{-14}$
14	3000	564	$2{,}212972 \times 10^{-12}$
15	3000	3243	$3{,}833462 \times 10^{-10}$

GMRES. Estas matrizes são beneficiadas pela adição do subespaço extra para inclusão do histórico dos ciclos anteriores.

Das seis matrizes de teste que não convergem, somente as matrizes *comsol* e *wang*3 (Ids 6 e 15) obtiveram uma melhora solução final em relação a solução final apresentada no GMRES reiniciado. As outras quatro matrizes apresentaram uma solução final pior.

5.12.2 Tempo médio de execução

Esta próxima avaliação considera o tempo de execução médio de cada método para todas as 15 matrizes de testes. Para isso, cada matriz foi executada 10 vezes em cada método. Além disso, o tempo de execução contabilizado desconsidera as alocações iniciais de memória e sua posterior liberação. Nas próximas tabelas são apresentados os tempos médios de execução e o desvio padrão.

A Tabela 33 mostra os tempos médios de execução, em segundos, do método GMRES, o método direto usado como base para os outros métodos iterativos.

A matriz $matrix - new\_3$ (Id 10) é a única matriz que não foi executada. Não foi possível alocar todos os vetores necessários para calcular a solução. Logo, não há tempo de execução do método GMRES para esta matriz.

Nessa Tabela 33, os tempos de execução das matrizes $circuit\_2$, *comsol*, $flowmeter5$, *sherman*4 e *wang*3 (Ids 5, 6, 7, 14 e 15) que não convergem são proporcionais ao tamanho da matriz. Isso porque a quantidade de iterações feitas para cada uma dessas matrizes é exatamente esse tamanho.

A Tabela 34 ilustra os tempos médios de execução (em segundos) e desvio padrão das matrizes no método GMRES reiniciado e no método GMRES flexível.

Sete matrizes de teste obtiveram tempos de execução maiores no GMRES reiniciado do que no GMRES. São elas as matrizes *cavity*10, *comsol*, $fpga\_trans\_01$, $fpga\_trans\_02$, $raefsky1$, $raefsky2$ e *sherman*4 (Ids 3, 6, 8, 9, 12, 13 e 14). Isso porque essas matrizes executaram mais iterações (multiplicando a quantidade de ciclos por MAX_M, que é

Tabela 33 – Tempos médios de execução do método GMRES com tolerância 10^{-15}.

Id	Tempo médio de execução segundos	Desvio padrão
1	0,0967	0,002766
2	0,1497	0,002078
3	1,0007	0,023182
4	4902,9156	8,118090
5	103,0601	0,224333
6	4,0207	0,018977
7	1004,1200	0,861779
8	0,2003	0,002782
9	0,1894	0,001752
10	-	-
11	1,2657	0,007631
12	0,5029	0,002777
13	0,7635	0,005857
14	1,4539	0,009758
15	32597,5632	1831,01280

Tabela 34 – Tempos médios de execução do método GMRES reiniciado e do GMRES flexível com tolerância 10^{-15}.

	GMRES reiniciado		GMRES flexível	
Id	Tempo médio de execução segundos	Desvio padrão	Tempo médio de execução segundos	Desvio padrão
1	0,064059	0,000876	0,018476	0,001047
2	0,058551	0,002615	0,039552	0,000669
3	2,147949	0,031130	1,083391	0,021335
4	207,104146	0,557788	4,948423	0,024514
5	0,453622	0,003812	0,165113	0,001743
6	28,251005	0,154795	37,079045	0,166875
7	61,615598	0,432146	92,688475	0,529369
8	0,421644	0,002971	0,778223	0,005931
9	0,443990	0,003774	0,600246	0,004002
10	1426,955728	0,282931	2164,92870	1,611242
11	0,807732	0,004787	0,076857	0,000648
12	5,489981	0,033382	9,006654	0,056323
13	3,911233	0,020930	5,108301	0,026817
14	6,025116	0,050716	8,495109	0,053352
15	197,097241	0,191270	291,225854	0,612846

igual a 30) do que no GMRES.

Da mesma forma, sete matrizes de teste apresentaram tempos menores no GMRES reiniciado do que no GMRES. São elas: as matrizes com Ids 1, 2, 4, 5, 7, 11 e 15. Dessas matrizes, a matriz *add*20, *cavity*05, $circuit\_2$ e *memplus* (Ids 1, 2, 5 e 11) executaram com menos iterações no total se comparado ao método GMRES. As outras matrizes *chipcool*0, $flowmeter5$ e *wang*3 (Ids 4, 7 e 15), apesar do total de iterações realizadas ser maior no

GMRES reiniciado, executam com uma quantidade pequena de iterações por ciclo. Este fato faz com que as matrizes V e H geradas sejam menores. Dessa forma, o tempo de execução de cada ciclo é reduzido.

Como o GMRES flexível realiza mais processamento por ciclo do que o GMRES reiniciado, as matrizes que não convergem no limite de ciclos estabelecido possuem um tempo de execução maior. São elas as matrizes *comsol*, $flowmeter5$, $matrix-new\_3$, $sherman4$ e $wang3$ (Ids 6, 7, 10, 14 e 15).

As matrizes Ids 8, 9, 12 e 13 também obtiveram o tempo médio de execução maior no GMRES flexível se comparado ao GMRES reiniciado. A matriz $raefsky2$ (Id 13) executou com o mesmo número de ciclos nos dois métodos. Como cada ciclo do GMRES flexível possui mais processamento do que o ciclo do GMRES reiniciado, o tempo de execução total foi maior. As matrizes $fpga\_trans\_01$ e $raefsky1$ (Ids 8 e 12) executaram com mais ciclos no GMRES flexível do que no GMRES reiniciado. Já na matriz $fpga\_trans\_02$ (Id 9) observamos uma redução no número de ciclos. Porém, esta redução não foi o suficiente para compensar o aumento de processamento por ciclo.

As outras seis matrizes obtiveram tempos de execução menores no GMRES flexível se comparadas ao método GMRES reiniciado. As matrizes em questão convergiram com menos ciclos no primeiro método do que no segundo, e por isso, o tempo de execução foi menor. As matrizes são a $add20$, $cavity05$, $cavity10$, $chipcool0$, $circiut\_2$ e *memplus* (Ids 1, 2, 3, 4, 5 e 11).

A Tabela 35 mostra os tempos médios de execução (em segundos) e desvio padrão das matrizes nos métodos αGMRES e *Heavy Ball* GMRES.

Tabela 35 – Tempos médios de execução do método αGMRES e do *Heavy Ball* GMRES com tolerância 10^{-15}.

	αGMRES		*Heavy Ball* GMRES	
Id	Tempo médio de execução segundos	Desvio padrão	Tempo médio de execução segundos	Desvio padrão
1	0,048942	0,001400	0,065369	0,001254
2	0,054498	0,001581	0,050392	0,000740
3	1,890231	0,013089	1,199724	0,001389
4	207,436188	4,764457	206,788875	0,483030
5	0,347686	0,003294	0,367036	0,000956
6	28,001572	0,185834	28,179758	0,017044
7	63,002206	0,340617	64,190934	0,724241
8	0,172907	0,002441	0,631950	0,005598
9	0,141085	0,001979	0,229159	0,000737
10	1410,353946	0,924836	1433,292330	1,758677
11	0,645189	0,001602	0,766207	0,003319
12	1,776893	0,011270	3,962495	0,010275
13	2,923058	0,018424	3,835903	0,035344
14	5,635719	0,036115	6,209164	0,032767
15	189,916926	0,553060	201,106835	0,705190

Praticamente todas as matrizes de teste obtiveram tempos de execução no αGMRES menores se comparadas ao método GMRES reiniciado. Como a modificação do método αGMRES altera a quantidade de iterações por ciclo, podendo ser de 3 a 30 iterações, até

as matrizes que precisariam de mais ciclos, podem ser executadas com menos iterações no total de todos os ciclos.

As matrizes *chipcool*0 e $flowmeter5$ (Ids 4 e 7) foram as únicas duas matrizes de teste que obtiveram tempos de execução maiores no αGMRES se comparadas ao GMRES reiniciado. Essas matrizes não convergem em ambos os métodos. Provavelmente, no método αGMRES todos os ciclos são executados com o máximo de iterações possível. Assim,o tempo total de execução é maior no αGMRES, porque é necessário calcular a taxa de convergência a cada ciclo e o valor das iterações.

No método *Heavy Ball* GMRES, as sete matrizes que convergem com uma menor quantidade de ciclos, se comparadas ao método GMRES reiniciado, obtiveram, também, redução no tempo de execução. As matrizes em questão são *cavity*05, *cavity*10, $circuit\_2$, $fpga\_trans\_02$, $memplus$, $raefsky1$ e $raefsky2$ (Ids 2, 3, 5, 9, 11, 12 e 13).

A matriz *add*20 (Id 1) que convergiu com a mesma quantidade de ciclos nos dois métodos teve um tempo de execução maior no *Heavy Ball* GMRES. O motivo disto é justamente por conta do processamento a mais de inclusão do subespaço extra no subespaço de procura a cada ciclo. A matriz $fpga\_trans\_01$ também obteve um tempo de execução maior no *Heavy Ball* GMRES. Porém, como precisou de mais ciclos para convergir, já era esperado esse aumento no tempo de execução.

Das outras seis matrizes de teste, duas obtiveram um tempo de execução menor, provavelmente por não incluir em nenhum dos ciclos o subespaço extra e, assim, não foi necessário calcular o vetor extra. As outras quatro matrizes obtiveram tempos de execução maiores no *Heavy Ball* GMRES do que no GMRES reiniciado, devido a esse processamento a mais de cada ciclo para incluir o subespaço extra no subespaço de procura.

5.12.3 Validação do método αGMRES

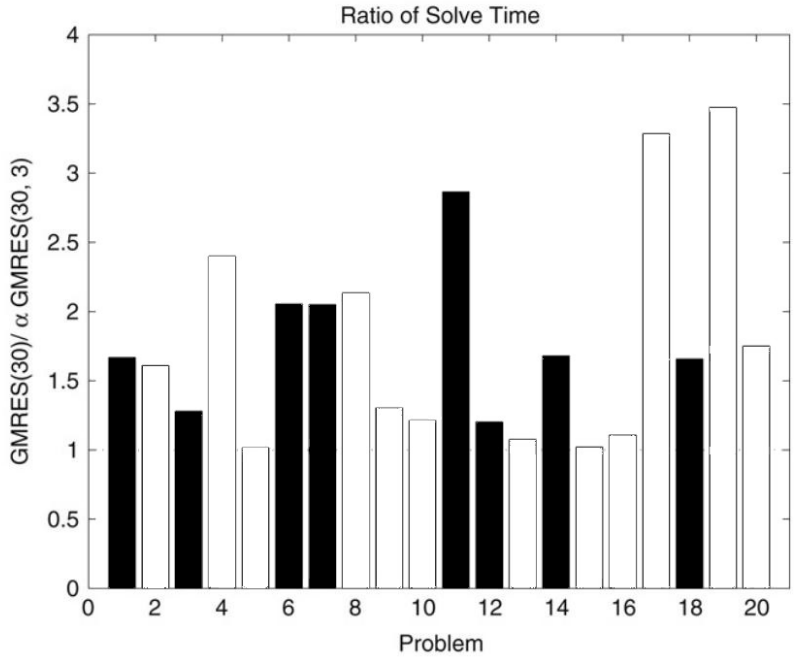
Das 15 matrizes de teste, oito delas foram testadas também no artigo (BAKER; JESSUP; KOLEV, 2009). Foram as matrizes *add*20, $circuit\_2$, $fpga\_trans\_01$, $matrix - new\_3$, $raefsky1$, $raefsky2$, *sherman*4 e 15 *wang*3 (Ids 1, 5, 8, 10, 12, 13, 14 e 15). A Figura 14, retirada de (BAKER; JESSUP; KOLEV, 2009), apresenta a relação R de quantas vezes o GMRES reiniciado é mais lento que o αGMRES, utilizando a equação (5.29). As colunas preenchidas são as que representam as matrizes citadas e estão na mesma ordem. As outras colunas foram outras matrizes que os autores utilizaram para os testes.

$$R = \frac{t_{GMRES(m)}}{t_{\alpha GMRES}} \tag{5.29}$$

A Figura 15 apresenta a mesma relação anterior dos tempos (tempo do GMRES(m) dividido pelo tempo do αGMRES), porém com os valores encontrados nos nossos experimentos. Nesta figura as colunas preenchidas são as que representam as mesmas matrizes citadas anteriormente, na ordem alfabética.

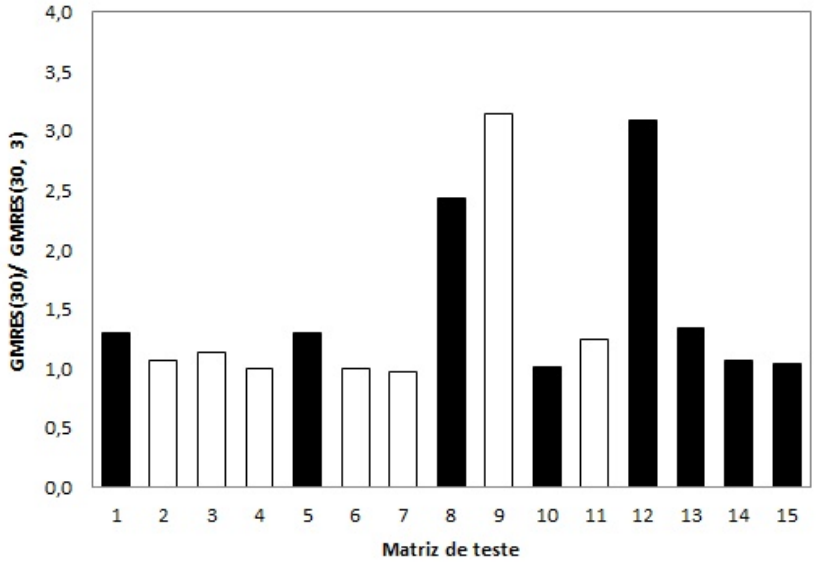
Comparando os valores das oito matrizes de teste em comum, observa-se que quatro delas tem valores bem diferentes. São as matrizes *add*20, $matrix - new\_3$, *sherman*4 e *wang*3. Dessas quatro, as últimas três não convergiram com 3000 iterações. No artigo (BAKER; JESSUP; KOLEV, 2009), os autores utilizam outra forma de testar a convergência das matrizes. Eles consideram a norma do resíduo do ciclo atual dividido pela norma do resíduo do ciclo zero. Este fato pode fazer as matrizes convergirem mais rápido no αGMRES e aumentar essa diferença entre ele e o GMRES(m).

Figura 14 – Comparação do tempo de convergência das matrizes de teste. Adaptado de (BAKER; JESSUP; KOLEV, 2009)



Fonte: (BAKER; JESSUP; KOLEV, 2009)

Figura 15 – Comparação do tempo de convergência das matrizes de teste



5.12.4 Validação do método *Heavy Ball* GMRES

No artigo (IMAKURA; LI; ZHANG, 2016), os autores compararam o *Heavy Ball* GMRES com o GMRES reiniciado, além de dois outros métodos. Eles utilizaram dez das nossas 15 matrizes de teste, as matrizes *cavity*05, *cavity*10, *chipcool*0, *comsol*, $flowmeter5$, $fpga_trans_02$, *memplus*, $raefsky1$, $raefsky2$, *wang*3 (ids 2, 3, 4, 6, 7, 9, 11, 12, 13 e 15). Como no artigo do αGMRES, os autores que apresentaram o *Heavy Ball* GMRES

utilizaram uma forma diferente de calcular a convergência dos métodos, eles utilizaram a equação (5.30), chamada de Resíduo Normalizado (NRes) sendo $\|A\|_1$ encontrada pela fórmula apresentada na Definição 22.

$$\text{NRes} = \frac{\|A\vec{x} - \vec{b}\|_2}{\|A\|_1\|\vec{x}\|_2 + \|\vec{b}\|_2} \tag{5.30}$$

Essa equação representa o resíduo normalizado. Com ela pode ser possível chegar ao valor da tolerância mais rápido se comparado ao cálculo do resíduo puro ($\|A\vec{x} - \vec{b}\|_2$), utilizado nos nossos experimentos. Além dessa diferença, os autores de (IMAKURA; LI; ZHANG, 2016) utilizaram técnicas de reortogonalização no processo de Gram-Schmidt modificado. Eles duplicaram o *loop* que preenche a linha de H, tanto no ciclo do GMRES original quanto na adição do vetor a mais no *Heavy Ball* GMRES. Assim, eles evitam que valores pequenos sejam anulados.

A Tabela 36 apresenta a quantidade de iterações que os autores (IMAKURA; LI; ZHANG, 2016) encontraram para o GMRES reiniciado e para o *Heavy Ball* GMRES, para as dez matrizes citadas anteriormente. A coluna Reortogonalização forçada representa os valores encontrados sempre que o *loop* duplicado é executado. A coluna Reortogonalização seletiva representa os valores encontrados, somente quando a norma do vetor w após o *loop* é menor que uma tolerância.

Tabela 36 – Número de ciclos do GMRES original e do *Heavy Ball* GMRES encontrados no artigo (IMAKURA; LI; ZHANG, 2016)

	Reortogonalização forçada		Reortogonalização seletiva	
Id	GMRES(m)	HBGMRES	GMRES(m)	HBGMRES
2	741	76	477	76
3	1961	171	1623	172
4	-	533	-	491
6	2931	261	-	287
7	-	1409	-	1400
9	155	38	155	38
11	83	38	83	38
12	204	38	218	38
13	188	140	188	135
15	25	23	26	23

As células com '-' significam que a matriz não convergiu no limite 3000 ciclos.

Esse resíduo normalizado que os autores do *Heavy Ball* GMRES implementaram aumentou a quantidade de iterações em quatro matrizes de teste no GMRES reiniciado, enquanto que em outras quatro diminuiu. As matrizes *chipcool*0 e $flowmeter5$ (Ids 4 e 7) também não convergiram com este teste de convergência, mesmo com a reortogonalização forçada.

A Tabela 37 apresenta a quantidade de ciclos que os métodos GMRES reiniciado e o *Heavy Ball* GMRES convergiram para a solução nos nossos experimentos. Estes são os mesmos valores das Tabelas 29 e 32.

Apesar da quantidade de ciclos ser diferente entre os resultados do artigo que apresenta o *Heavy Ball* (IMAKURA; LI; ZHANG, 2016) e os resultados do presente trabalho, em ambos os resultados podemos perceber que o HBGMRES consegue convergir com menos

Tabela 37 – Número de ciclos do GMRES original e do *Heavy Ball* GMRES encontrados

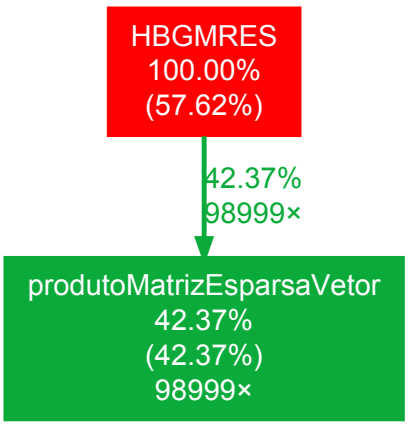
Id	GMRES(m)	HBGMRES
2	13	12
3	204	119
4	-	-
6	-	-
7	-	-
9	176	88
11	19	18
12	208	151
13	147	146
15	-	-

iterações que o GMRES reiniciado, ou GMRES(m).

5.12.5 Paralelização do *Heavy Ball* GMRES

E um último experimento estudamos pontos de paralelização do HBGMRES (Algoritmo 39). Iniciamos com a execução um analisador de desempenho do Linux, chamado **gprof** para verificar em que pontos do código são gastos as maiores porcentagens do tempo de execução. Obtivemos os seguintes resultados: 57,69% do tempo de execução é gasto na função principal (Algoritmo 39) e 42,42% do tempo de execução em muitas chamadas para a função *produtoMatrizEsparsaVetor* (Algoritmo 30). A Figura 16 mostra esse resultado obtido.

Figura 16 – Resultado da análise de desempenho do *Heavy Ball* GMRES com a matriz *chipcool*0



Considerando esse resultado, realizamos um simples teste de paralelizar o *loop* da função *produtoMatrizEsparsaVetor* (Algoritmo 30), utilizando o OpenMP (CHAPMAN; JOST; PAS, 2008), uma API com modelo de paralelismo de memória compartilhada. Uma primeira abordagem foi dividir todas as linhas da matriz igualmente pelas N *threads*, como mostra o Algoritmo 40.

Como a matriz A é esparsa e a posição dos seus elementos não é previsível, essa primeira abordagem pode levar a um grande desbalanceamento de carga. Para evitar isso, usamos

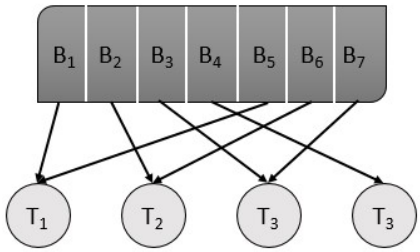
Algoritmo 40 Primeira versão paralela do produto da matriz A armazenada em CSR por um vetor v.

função produtoMatrizEsparsaPorVetor(**Vetores:** AA, JA, IA, x)
1: **#pragma omp** parallel for
2: **para** $i \leftarrow 1$ até N **faça**
3: $k1 \leftarrow IA[i]$
4: $k2 \leftarrow IA[i+1]$
5: **para** $j \leftarrow k1$ até $k2$ **faça**
6: $auxAA[j-k1+1] \leftarrow AA[j]$
7: $auxX[j-k1+1] \leftarrow x[JA[j]]$
8: $y[i] \leftarrow$ produtoInternoVetorial($auxAA$, $auxX$, $k2-k1$)
9: **retorna** y

o parâmetro **scheduler** do *parallel for* do OpenMP para modificar o tamanho de cada bloco e como eles são alocados a cada *thread*.

Ao modificarmos o parâmetro, chegamos a duas abordagens diferentes, uma com um escalonador estático (*static scheduler*) e outra com um escalonador dinâmico (*dynamic scheduler*). O escalonador estático aloca os blocos a cada *thread* de maneira rotativa antes do início da computação, como mostra a figura 17.

Figura 17 – Escalonador estático



Já o escalonador dinâmico aloca os primeiros N blocos antes do início da computação e só aloca os outros blocos a medida que as *threads* acabam e pedem. Esse tipo de escalonador tende a ter uma carga de trabalho mais balanceada entre as *threads*, uma vez que as que terminam mais rápido tem mais chances de pegar mais blocos. Porém, existe o *overhead* de sincronização com o escalonador.

Assim, cada uma das abordagens foi executada 10 vezes para cada matriz e a média dos tempos da primeira abordagem é apresentada na Tabela 38, junto com o desvio padrão de cada matriz. Além disso, calculamos o *speedup* em relação ao tempo de execução sequencial, que está inserido na mesma tabela. A fórmula para se encontrar o *speedup* é apresentada em (2.3).

Como podemos observar, essa primeira inserção de paralelismo não proporcionou bons resultados. Isso porque a maioria das matrizes obtiveram *speedups* próximos a 1. Apesar do *loop* ter sido dividido para quatro núcleos de processamento, o *overhead* de cópia de memória e da comunicação entre as *threads* a cada chamada da função *produtoMatrizEsparsaVetor* faz com que o tempo ganho com o processamento paralelo seja encoberto, na maioria das matrizes.

Para as matrizes *add*20, *circuit_*2 e *sherman*4 (Ids 1, 5 e 14) o tempo de processamento da versão paralela foi pior que o tempo da versão sequencial. Isso pode ser

Tabela 38 – Tempos médios de execução do método *Heavy Ball* GMRES paralelo com tolerância 10^{-15} e o *speedup* obtido

Id	Tempo médio de execução segundos	Desvio padrão	*Speedup*
1	0,068489	0,002686	0,95
2	0,035779	0,005251	1,41
3	0,780592	0,007801	1,54
4	164,461376	0,498391	1,26
5	0,380447	0,003685	0,96
6	14,612236	0,143842	1,93
7	57,937051	0,111559	1,11
8	0,592008	0,004616	1,07
9	0,219295	0,003758	1,04
10	1183,001086	0,645202	1,21
11	0,749781	0,004988	1,02
12	1,873837	0,015251	2,11
13	1,849561	0,017136	2,07
14	6,354894	0,096503	0,98
15	200,028231	0,382332	1,01

explicado pela quantidade de iterações e pelo tamanho das matrizes. Como essa função só percorre os elementos não nulos, basta verificar o valor de nz na Tabela 26, que é igual a 17319 para a matriz 1, 21199 para a matriz 5 e 3786 para a matriz 14.

Além desses motivos, o desbalanceamento da carga entre os 4 blocos iguais pode ter afetado o tempo final da primeira abordagem paralela. Com isso, apresentamos nas Tabelas 39 e 40 os tempos das duas outras abordagens mencionadas, as que mexem no tamanho do bloco e como cada bloco é escalonado para as *threads*.

Com esses resultados, percebemos que a maioria das matrizes tiveram resultados similares tanto com o escalonador estático quando com a divisão de blocos igual, mas com o escalonador dinâmico o *speedup* foi pior. Isso pode ser explicado pelo *overhead* de comunicação das *threads* com o escalonador para requisitarem novos blocos durante a execução.

Além disso, o escalonador estático não obteve *speedups* muito melhores que a divisão em 4 blocos iguais, chegando a diminuir muitos *speedups*. Assim, a melhor abordagem das três estudadas ainda seria a divisão igualitária dos blocos.

Tabela 39 – Tempos médios de execução do método *Heavy Ball* GMRES paralelo com escalonador estático e o *speedup* obtido

Id	Tempo médio de execução segundos	Desvio padrão	*Speedup*
1	0,061480	0,002920	1,06
2	0,037528	0,003061	1,34
3	0,827422	0,007022	1,45
4	183,747461	0,699584	1,13
5	0,371530	0,016252	0,99
6	15,701355	0,137489	1,79
7	62,109510	0,374294	1,03
8	0,604768	0,003978	1,04
9	0,222982	0,003222	1,03
10	1399,201802	2,351325	1,02
11	0,755473	0,012058	1,01
12	1,971801	0,013628	2,01
13	1,939819	0,019811	1,98
14	6,675390	0,115190	0,93
15	213,821890	0,502127	0,94

Tabela 40 – Tempos médios de execução do método *Heavy Ball* GMRES paralelo com escalonador dinâmico e o *speedup* obtido

Id	Tempo médio de execução segundos	Desvio padrão	*Speedup*
1	0,097174	0,001929	0,67
2	0,052132	0,006305	0,97
3	1,160977	0,007829	1,03
4	226,715823	0,698377	0,91
5	0,620098	0,005381	0,59
6	17,985670	0,303759	1,57
7	93,053455	0,257158	0,69
8	1,044480	0,005063	0,61
9	0,384133	0,004220	0,60
10	1632,108570	0,452912	0,88
11	1,043245	0,004080	0,73
12	2,194444	0,013922	1,81
13	2,170768	0,021662	1,77
14	10,441532	0,192912	0,59
15	288,005439	0,406287	0,70

CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho teve como objetivo aprofundar o entendimento de alguns métodos de resolução de sistemas lineares e observar possíveis pontos de paralelismo. Para isso realizamos três estudos comparativos.

O primeiro estudo abordou os métodos diretos para matrizes cheias, onde comparamos seis métodos diferentes. Os métodos comparados foram: Eliminação Gaussiana, Método de Gauss-Jordan, Decomposição LU, Decomposição LDU, Decomposição de Cholesky e Decomposição QR por Gram-Schmidt. Este estudo é a compilação de três artigos publicados: um para a IV Escola Regional de Alto Desempenho do Rio de Janeiro (ERAD-RJ), apresentado no fórum de Pós-Graduação; outro para o XXXVIII Congresso Nacional de Matemática Aplicada e Computacional (CNMAC) (BRUM; CASTRO; FARIA, 2018a); e um artigo submetido para a revista TEMA (*Trends in Applied and Computational Mathematics*). Este último artigo ainda está em processo de revisão.

Neste primeiro estudo, fizemos experimentos com ambas as alocações de memória (estática e dinâmica), e com variáveis de precisão simples e precisão dupla. Comparando as duas alocações, não obtivemos um ganho de de tempo esperado com a alocação dinâmica. Entre as duas precisões, a precisão dupla tem os resultados mais preciso. Porém, observamos que a medida que o tamanho da matriz cresce, a diferença do tempo de execução da precisão dupla e da simples aumenta. O último experimento realizado com os métodos diretos considerou às *flags* de compilação do compilador GCC, com as versões 5.4.0 (2016) (GCC, 2016) e 7.1.0 (2017) (GCC, 2017). Como primeira observação, consideramos a flag -Os, um dos primeiros níveis de otimização, como a melhor *flag* para a Eliminação Gaussiana e para o Método de Gauss-Jordan na versão 7.1.0 do GCC. Além disso, a versão de 2016 do GCC obteve tempos de execução menores se comparados a versão de 2017, provavelmente devido a instalação forçada dessa última versão.

No segundo estudo comparativo consideramos seis métodos iterativos para matrizes cheias. Os métodos estudados foram o Jacobi-Richardson, o Gauss-Seidel, o SOR, o JOR, o DOR (ANTUONO; COLICCHIO, 2016) e o Gauss-Seidel distribuído (SHANG, 2009). Este capítulo é baseado no artigo enviado para o XXI Encontro Nacional de Modelagem Computacional (ENMC) (BRUM; CASTRO; FARIA, 2018b).

Avaliamos a convergência do método DOR, através de experimentos computacionais, para valores do parâmetro ω entre 0 e 2. Além disso, o tempo de execução do método DOR é similar ao dos métodos JOR e SOR para as matrizes testadas, que foram geradas sinteticamente. Mas os melhores resultados foram obtidos com o método Gauss-Seidel. Este segundo estudo também reproduziu os resultados do artigo (SHANG, 2009), do método DGS, com uma biblioteca de troca de mensagens diferente (MPI). Observamos um ganho de desempenho (*speedup*) igual a 1,11 com esta versão. Esse ganho é pequeno comparado com o máximo de ganho possível. Porém, este é um método com muita comunicação entre os processos, o que acarreta num aumento do *overhead* de tempo da paralelização. Com isso, comprovamos que o método de GS é um método paralelizável, apesar da comunicação excessiva.

O último estudo abordou os métodos de projeção iterativos para matrizes esparsas. Os métodos estudados foram o GMRES reiniciado (GMRES(m)), o GMRES flexível (FGMRES), o αGMRES e o *Heavy Ball* GMRES (HBGMRES). Neste capítulo, validamos as implementações dos αGMRES e do HBGMRES com os resultados obtidos em (BAKER; JESSUP; KOLEV, 2009) e (IMAKURA; LI; ZHANG, 2016), além de compararmos a taxa de convergência e o tempo de execução dos métodos. Mostramos que essas modificações do GMRES reiniciado reduzem a taxa de convergência e o tempo de execução das matrizes de testes usadas. Por fim, apresentamos três versões paralela do HBGMRES, baseada numa API com o paradigma de memória compartilhada (OpenMP) e obtivemos um *speedup* de, no máximo, 2,11.

Para trabalhos futuros, pretende-se estudar outras formas de otimização da memória utilizada e fazer testes com matrizes maiores. Além disso, pretendemos desenvolver novas versões desses algoritmos considerando as arquiteturas NUMA. Em uma arquitetura NUMA todos os processadores têm acesso a memória toda. Normalmente, eles usam operações de leitura e escrita para acessar a memória. Porém, diferentes processadores têm acesso às regiões da memória em diferentes velocidades. Neste caso é necessário um cuidado especial na implementação dos algoritmos para extrair melhor desempenho destas arquiteturas.

REFERÊNCIAS

ANTUONO, M.; COLICCHIO, G. Delayed Over-Relaxation for iterative methods. In: . [s.n.], 2016. v. 321, p. 892–907. ISSN 00219991. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0021999116302418>.

ASCENCIO, A. F. G.; CAMPOS, E. A. V. d. *Fundamentos da programação de computadores: algoritmos, Pascal, C/C++ e Java*. São Paulo: Pearson Prentice Hall, 2008. OCLC: 319216929. ISBN 978-85-7605-148-0.

BAKER, A. H.; JESSUP, E. R.; KOLEV, T. V. A simple strategy for varying the restart parameter in GMRES(m). *Journal of computational and applied mathematics*, Elsevier, v. 230, n. 2, p. 751–761, 2009.

BAKER, A. H.; JESSUP, E. R.; MANTEUFFEL, T. A technique for accelerating the convergence of restarted gmres. *SIAM Journal on Matrix Analysis and Applications*, SIAM, v. 26, n. 4, p. 962–984, 2005.

BOISVERT, R. F. et al. *Matrix Market*. 1999. Disponível em: <https://math.nist.gov/MatrixMarket/index.html>.

BOTOR, T.; HABIBALLA, H. Compiler optimization for scientific computation in c/c++. In: *International Conference of Computational Methods in Sciences and Engineering 2018 (ICCMSE 2018)*. AIP Conference Proceedings, 2018. v. 2040, n. 1, p. 030004. Disponível em: <https://aip.scitation.org/doi/abs/10.1063/1.5079067>.

BRUM, R. C.; CASTRO, M. C. S.; FARIA, C. O. A influência da alocação de memória e precisão na resolução de sistemas lineares. In: *Proceeding Series of the Brazilian Society of Computational and Applied Mathematics*. Campinas: Sociedade de Matemática Aplicada e Computacional, 2018.

BRUM, R. C.; CASTRO, M. C. S.; FARIA, C. O. Um estudo comparativo entre métodos iterativos para resolução de sistemas lineares. In: *XXI Encontro Nacional de Modelagem Computacional e IX Encontro de Ciência e Tecnologia de Materiais*. Essentia Editora, 2018. ISBN 978-85-54077-00-6. Disponível em: <http://www.essentiaeditora.iff.edu.br/index.php/enmc-ectm/inproceedings/view/12243>.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. *Using OpenMP: portable shared memory parallel programming*. Cambridge, Mass: MIT Press, 2008. (Scientific and engineering computation). OCLC: ocn145944336. ISBN 978-0-262-53302-7 978-0-262-03377-0.

CONRAD, V.; WALLACH, Y. Iterative solution of linear equations on a parallel processor system. *IEEE Transactions on Computers*, C-26, n. 9, p. 838–847, Sep. 1977.

COURTECUISSE, H.; ALLARD, J. Parallel dense gauss-seidel algorithm on many-core processors. In: IEEE. *2009 11th IEEE International Conference on High Performance Computing and Communications*. Seoul, South Korea, 2009. p. 139–147.

CUNHA, M. C. C. *Métodos numéricos*. Campinas: Editora da UNICAMP, 2003.

DAVIS, T. A.; HU, Y. The university of florida sparse matrix collection. In: . New York, NY, USA: ACM, 2011. v. 38, n. 1, p. 1:1–1:25. ISSN 0098-3500. Disponível em: <http://doi.acm.org/10.1145/2049662.2049663>.

DEMMEL, J. W. *Applied numerical linear algebra*. Philadelphia: Siam, 1997. v. 56.

DENG, L.; YU, D. Deep learning: Methods and applications. *Foundations and Trends® in Signal Processing*, v. 7, n. 3–4, p. 197–387, 2014. ISSN 1932-8346. Disponível em: <http://dx.doi.org/10.1561/2000000039>.

EFTEKHARI, A.; BOLLHÖFER, M.; SCHENK, O. Distributed memory sparse inverse covariance matrix estimation on high-performance computing architectures. In: IEEE PRESS. *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. Dallas, Texas, 2018. p. 20.

ELMAN, H. C. *Iterative methods for large, sparse, nonsymmetric systems of linear equations*. Tese (Doutorado) — Yale University New Haven, Conn, 1982.

FRANCO, N. B. *Cálculo numérico*. São Paulo: Pearson, 2006.

GCC. *GCC 5.4.0 Online Documentation*. 2016. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/>.

GCC. *GCC 7.1.0 Online Documentation*. 2017. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc/>.

GEIST, A. et al. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. Cambridge, Massachusetts: MIT press, 1994.

GOLUB, G. H.; LOAN, C. F. V. *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996. ISBN 0-8018-5414-8.

GROPP, W. D. et al. *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, Massachusetts: MIT press, 1999. v. 1.

HACKBUSCH, W. *Iterative solution of large sparse systems of equations*. New York: Springer, 1994. v. 95.

HOSTE, K.; EECKHOUT, L. Cole: compiler optimization level exploration. In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. [S.l.]: ACM, 2008. p. 165–174.

IMAKURA, A.; LI, R.-C.; ZHANG, S.-L. Locally optimal and heavy ball gmres methods. *Japan Journal of Industrial and Applied Mathematics*, Springer, v. 33, n. 2, p. 471–499, 2016.

IMECEK, I.; LANGR, D.; TVRDíK, P. Space-efficient sparse matrix storage formats for massively parallel systems. In: *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*. Liverpool, UK: IEEE, 2012. p. 54–60.

LI, B. et al. A one-step matrix application method for maldi mass spectrometry imaging of bacterial colony biofilms. *Journal of Mass Spectrometry*, v. 51, n. 11, p. 1030–1035, 2016. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/jms.3827>.

LI, J.; SUN, J.; VUDUC, R. Hicoo: Hierarchical storage of sparse tensors. In: IEEE. *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. Dallas,Texas, 2018. p. 238–252.

LIPSCHUTZ, S. *Álgebra linear (4a. ed.)*. Grupo A - Bookman, 2000. ISBN 978-85-407-0041-3. Disponível em: <http://public.eblib.com/choice/publicfullrecord.aspx?p=3236279>.

MCCAUGHEY, D. et al. Best practices in social media: Utilizing a value matrix to assess social media's impact on health care. *Social Science Computer Review*, v. 32, n. 5, p. 575–589, 2014. Disponível em: <https://doi.org/10.1177/0894439314525332>.

PATTERSON, D. A.; HENNESSY, J. L. *Organização e projeto de computadores: interface hardware/software*. Rio de Janeiro: Elsevier Brasil, 2014. OCLC: 902739007. ISBN 978-85-352-6410-4. Disponível em: <http://www.sciencedirect.com/science/book/9788535235852>.

PENG, W.; PARK, D. H. Generate adjective sentiment dictionary for social media sentiment analysis using constrained nonnegative matrix factorization. In: *Fifth International AAAI Conference on Weblogs and Social Media*. Barcelona, Espanha: AAAI Publications, 2011.

POLYAK, B. T. Introduction to optimization. optimization software. *Inc., Publications Division*, New York, v. 1, 1987.

RUGGIERO, M. A. G.; LOPES, V. L. d. R. *Cálculo numérico: aspectos teóricos e computacionais*. São Paulo: Makron Books do Brasil, 1997.

SAAD, Y. A flexible inner-outer preconditioned gmres algorithm. In: . [S.l.]: SIAM, 1993. v. 14, n. 2, p. 461–469.

SAAD, Y. *Iterative methods for sparse linear systems*. New York: siam, 2003. v. 82.

SAAD, Y.; SCHULTZ, M. H. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, SIAM, v. 7, n. 3, p. 856–869, 1986.

SCHUBERT, G. et al. Parallel sparse matrix-vector multiplication as a test case for hybrid mpi+openmp programming. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. Shanghai, China: IEEE, 2011. p. 1751–1758. ISSN 1530-2075.

SEDGEWICK, R. *Algorithms in C*. Reading, Mass: Addison-Wesley Pub. Co, 1990. (Addison-Wesley series in computer science). ISBN 978-0-201-51425-4.

SHANG, Y. A distributed memory parallel Gauss–Seidel algorithm for linear algebraic systems. *Computers & Mathematics with Applications*, v. 57, n. 8, p. 1369–1376, abr. 2009. ISSN 08981221. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S089812210900042X>.

WATKINS, D. S. *Fundamentals of matrix computations*. 2nd ed. ed. New York: Wiley-Interscience, 2002. (Pure and applied mathematics). ISBN 978-0-471-21394-9.

WOLFSON-POU, J.; CHOW, E. Distributed southwell: an iterative method with low communication costs. In: ACM. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver, Colorado, 2017. p. 48.

YZELMAN, A. N.; ROOSE, D. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, v. 25, n. 1, p. 116–125, Jan 2014. ISSN 1045-9219.