Universidade Federal Fluminense

MARCO ANTONIO DE OLIVEIRA COELHO

Métodos Iterativos para Resolver Sistemas de Equações Algébricas Lineares em Estruturas Esparsas

VOLTA REDONDA

2014

MARCO ANTONIO DE OLIVEIRA COELHO

Métodos Iterativos para Resolver Sistemas de Equações Algébricas Lineares em Estruturas Esparsas

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional em Ciência e Tecnologia da Universidade Federal Fluminense, como requisito parcial para obtenção do título de Mestre em Modelagem Computacional em Ciência e Tecnologia. Área de Concentração: Modelagem Computacional.

Orientador:

Diomar Cesar Lobão

Coorientador:

Gustavo Benitez Alvarez
Emerson Souza Freire

Universidade Federal Fluminense

VOLTA REDONDA

2014

C672 Coelho, Marco Antonio de Oliveira.
Métodos iterativos para resolver sistemas de equações algébricas lineares em estruturas esparsas / Marco Antonio de Oliveira Coelho; orientador: Diomar Cesar Lobão; co-orientadores: Gustavo Benitez Alvarez, Emerson Souza Freire – Volta Redonda, 2015.
78 f. : il.

Dissertação (Mestrado em Modelagem Computacional) – Universidade Federal Fluminense, 2015.

1. Matriz Esparsa. 2. Métodos iterativos. 3. Algébra linear. I. Lobão, Diomar Cesar; II. Alvarez, Gustavo Benitez. III. Freire, Emerson Souza. IV. Título.

CDD 624.171

Métodos Iterativos para Resolver Sistemas de Equações Algébricas Lineares em Estruturas Esparsas

Marco Antonio de Oliveira Coelho

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional em Ciência e Tecnologia da Universidade Federal Fluminense, como requisito parcial para obtenção do título de Mestre em Modelagem Computacional em Ciência e Tecnologia. Área de Concentração: Modelagem Computacional.

Aprovada por:

Prof. Diomar Cesar Lobão. Ph.D. / MCCT-UFF

Prof. Emerson de Souza Freire. D.Sc. / MCCT-UFF

Prof. José Alberto Fernandes Ferreira, D.Sc. / ITA-UNITAU

Prof. Tiago Araújo Neves, D.Sc. / MCCT-UFF

Prof. Jorge Alberto Rodrigues Durán, D.Sc. / UFF

Prof. Panters Rodríguez Bermúdez, D.Sc. / MCCT-UFF

Volta Redonda, 23 de agosto de 2014.

Dedicatoria.Para meus pais minha esposa e todos do programa de pós graduação do MCCT

Agradecimentos

Primeiramente a Deus, pela saúde e força para vencer mais essa etapa de minha vida.
A meus pais que me deram os ensinamentos para que chegasse aqui, a minha esposa Solange pelo incentivo e compreensão dos momentos em que estive ausente.
Aos professores desse programa de pós graduação, em especial ao meu orientador Ph.D Diomar César Lobão pela extrema atenção e boa vontade em ensinar e ajudar, aos professores D.Sc Gustavo Benitez, D.Sc Emerson de Souza Freire e D.Sc Vanessa da Silva Garcia pelos ensinamentos, só tenho a agradecer a atenção e boa vontade em transmitir o conhecimento para realização desse trabalho.
A todos os alunos do programa de pós graduação, a Lidiane e Karine da Secretaria, ao Júnior, aluno do programa de pós graduação e parceiro amigo que me acompanhou na trajetória desse trabalho.
Ao professor D.Sc Tiago Neves do VCE/EEIMVR/UFF pela gentileza de doar este TEMPLATE e esclerecer eventuais dúvidas sobre o funcionamento do mesmo.

Resumo

Os métodos iterativos para resolver sistemas lineares são muito utilizados na matemática, física e engenharia, e possuem a vantagem de não gerar erros causados pela aritmética finita dos computadores. Problemas gerados a partir de fenômenos físicos resultam em sistemas com matrizes esparsas, onde a ultilização de estruturas para armazenamento trazem grande economia de memória e diminuição significativa do tempo de processamento com o uso do computador. Neste trabalho, serão apresentadas algumas das principais estruturas de armazenamento de matrizes esparsas e sua aplicação em dois métodos clássicos de resolução de sistemas lineares iterativos Jacobi e Gauss-Seidel, com estudos comparativos de eficiência e a construção dos gráficos de complexidade de cada método com a análise dos resultados obtidos.

Abstract

The Iterative methods for solving linear systems are widely used in mathematics, physics and engineering, and have the advantage of do not generating errors caused by the finite arithmetic of computers. Problems have been arisen from physical phenomena result in systems with sparse matrices, where the use of structures for storing bring large memory savings and a decrease in processing lifetime with computer using. In this aim, we present some of the main storage structures of sparse matrices and its application between two classical methods for solving linear systems iterative Jacobi and Gauss-Seidel, with comparative efficiency studies and the graphs of complexity of each method with analysis of results.

Palavras-chave

1. Matriz Esparsa
2. Métodos Iterativos
3. Eficiência Computacional
4. Alocação de Memória
5. Grau de Esparsidade
6. Complexidade

Glossário

MDF	:	*Método de Diferenças Finitas*
MEF	:	*Método de Elementos Finitos*
CSR	:	*Compressed Sparse Row*
CSC	:	*Compressed Sparse Column*
CDS	:	*Compressed Diagonal Storage*
GE	:	*Grau de Esparsidade*
KB	:	*Kilobyte*
MB	:	*Megabyte*
GB	:	*Gigabyte*

Sumário

Lista de Figuras

—

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Muitos dos problemas estudados pela ciência após a discretização geram sistemas lineares com matrizes esparsas de grande dimensão, onde o uso de métodos iterativos para sua solução se tornam uma boa opção. Os métodos diretos de resolução de sistemas lineares fazem com que os erros se acumulem devido aos erros de "round-off"[1], e são indicados para matrizes de pequenas dimensões. Em decorrência dos erros de arredondamento e aritmética finita "round-off", os métodos iterativos não sofrem a influência desses erros, de acordo com [2], tendo a vantagem de serem autocorretivos. Problemas aplicados a egenharia como a utilização do método de Elementos Finitos na análise de estruturas [3], método de Diferenças Finitas para resolver equações diferenciais parciais [4], Método de Elementos Finitos [5], geram grandes sistemas lineares com matrizes esparsas, onde o emprego de estruturas de armazenamento dessas matrizes proporcionam grande economia de memória e diminuição do tempo de processamento, obtendo grande eficiência computacional e rapidez nos resultados. Uma matriz para ser esparsa depende de alguns requisitos fundamentais: a matriz, o algoritmo e o computador. Qualquer matriz esparsa pode ser processada como se fosse densa, e de forma inversa, qualquer matriz densa pode ser trabalhada por um algoritmo de matrizes esparsas. Os resultados númericos obrigatóriamente devem ser os mesmos em ambos os casos, mas o fator principal será o custo computacional, sendo que uma matriz densa processada por um algoritmo projetado para matrizes esparsas vai ter menos desempenho, e da mesma forma, um algoritmo para matrizes densas vai ter baixo rendimento com uma matriz esparsa, fazendo operações matemáticas desnecessárias com elementos nulos. Atribuindo a propriedade de esparsa para uma matriz é alegar que existe um algoritmo que aproveita a esparsidade para tornar o cálculo dos

resultados mais eficiente computacionalmente. Um fato importante a ser ressaltado é a crescente utilização dos métodos iterativos nas últimas décadas na computação científica. Segundo [6], até recentemente os métodos diretos eram preferidos em aplicações reais por causa de sua robustez. No entanto, foram descobertos uma série de métodos iterativos e a necessidade de se resolver grandes sistemas lineares provocou uma visível e rápida mudança na direção dos métodos iterativos em muitas aplicações. Essa tendência, conforme [6], pode ser observada até a década de 1960 e 1970, quando dois acontecimentos revolucionaram os métodos de solução para grandes sistemas lineares. Primeiro foi a intenção de se tirar proveito da esparsidade dos sistemas para projetar métodos diretos de resolução de sistemas lineares esparsos, que pode ser bem mais eficiente que os métodos diretos convencionais, iniciado por engenheiros elétricos, são os métodos de solução diretos para sistemas esparsos, que levou ao desenvolvmento de softwares de solução direta eficientes ao longo das próximas três décadas, segundo foi o surgimento do método do gradiente conjugado pré-condicionado. Verificou-se que a combinação do pré-condicionado e do método iterativo de Krylov poderiam oferecer eficiência e simplicidade, procedimentos que poderiam competir com os métodos diretos. Gradualmente, os métodos iterativos começaram a se aproximar da qualidade dos métodos diretos. Em épocas anteriores, métodos iterativos foram muitas vezes utilizados para fins especiais na natureza, que foram desenvolvidos com determinadas aplicações em mente, e sua eficiência se baseou em muitos parâmetros dependentes do problema em questão.

Agora, modelos tridimensionais são comuns e métodos iterativos são sem dúvidas, os mais utilizados para resolução desses problemas. Os métodos iterativos geram uma sequência a cada passo, que converge para a solução exata do sistema e dessa forma tais resultados são sempre aproximados, mas respeitando uma tolerância de erro estabelecida pelo usuário. Nos dias atuais, onde os computadores possuem dupla precisão com processadores de 64 bits, pode-se obter soluções com tolerância de erro na casa de 10^{-15}, soluções muito próximas das exatas do sistema e aplícãveis em vários campos da ciência e engenharia. A memória e os requisitos computacionais para solução de problemas tridimensionais de equações diferenciais parciais, ou problemas em duas dimensões que envolvem muitos graus de liberdade por ponto, podem desafiar seriamente os métodos diretos mais eficientes disponíveis hoje. Além disso, os métodos iterativos vem ganhando espaço na computação científica, pela facilidade de implementação em computadores de alto desempenho, fato que não acontece com os métodos diretos que são mais difíceis e complicados de implementar [6].

1.2 Objetivos Gerais

O Objetivo do presente trabalho é estudar técnicas de armazenamento de matrizes esparsas e implementação dessas técnicas nos métodos iterativos de resolução de sistemas lineares para uso em matrizes esparsas. Tais técnicas computacionais visam aproveitar o grande número de elementos nulos da matriz e armazenar somente os elementos não-nulos ou apenas alguns deles, dimimuindo de forma significativa o número de entradas do algoritmo e evitando operações aritméticas desnecessárias entre elementos nulos. Dessa forma, obtém-se diminuição do tempo de processamento e uso de memória para alocação da matriz dos coeficientes e vetores do sistema linear. A consequência desse processo é rapidez na resolução de grandes problemas envolvendo sistemas lineares e menor utilização da memória e diminuição do tempo de processamento.

1.3 Objetivos Específicos

Através dos objetivos gerais expostos acima, segue os objetivos específicos do presente trabalho:

1. Estudar quatro técnicas de armazenamento de matrizes esparsas e implementar tais técnicas de armazenamento em dois métodos iterativos de resolução de sistemas lineares.

2. Resolver sistemas lineares formados por matrizes esparsas geradas aletóriamente e pentadiagonais oriundas da solução da equação de Laplace em 2D, através do método de elementos finitos.

3. Construir os gráficos e os polinômios de complexidade de cada algoritmo através dos tempos de processamento dos sistemas lineares descritos anteriormente, onde é possivel comparar a eficiência de cada um nos aspectos tempo de processamento e uso de memória para armazenamento.

1.4 Estrutura da Dissertação

O capítulo 2 apresenta algumas formas de representar a esparsidade de uma matriz e estruturas de armazenamento para matrizes esparsas, estruturas computacionais que reduzem consideravelmente a quantidade de memória para o armazenamento dos elementos

das matrizes . São apresentadas as estruturas CSR (Compressed Sparse Row), compressão em linha, CSC (Compressed Sparse Colunm), compressão em coluna, CDS (Compressed Diagonal Storage), compressão em diagonal e finalmente a estrutura Skyline ou matriz de bloco. No capítulo 3, são apresentados os métodos numéricos de Jacobi e Gauss-Seidel, sua formulação e condições de convergência, e posteriormente a utilização das estruturas esparsas aos métodos iterativos. No capítulo 4, é apresentado um código que gerou uma matriz de dimensão 200x200 de forma aleatória com objetivo de testar o funcionamento dos métodos Jacobi e Gauss-Seidel, para matrizes densas e com as estuturas de armazenamento de matrizes esparsas. Logo após, os resultados obtidos através das simulações feitas com seis sistemas lineares de dimensões 100x100, 200x200, 400x400, 600x600, 800x800 e 1000x1000, obtidos através de um código em ambiente Matlab®de solução da equação de Laplace em 2D, conforme [7]. De posse dos tempos de processamento de cada sistema e algoritmo, a construção dos gráficos e polinômios de complexidade, onde é possível avaliar cada Método numérico e técnica de armazenamento de matriz esparsa.

Capítulo 2

Estruturas para matrizes esparsas

2.1 Definição de Esparsidade

Do ponto de vista prático, uma matriz é dita esparsa quando contém uma quantidade considerável de elementos são nulos, sendo possível tirar proveito dessa esparsidade para economia de memória e tempo de processamento. Entretanto, para medir essa quantidade de elementos nulos, utiliza-se alguns parâmetros, entre os quais o parâmetro γ e o grau de esparsidade, como definidas adiante.

Definição 2.1 ([8]). *Uma matriz $A \in R^{m \times n}$ é dita esparsa se existe um número de elementos por linha $rmax << m$ e um número de elementos por coluna $cmax << n$.*

Definição 2.2 ([8]). *Seja uma matriz $A \in R^{m \times n}$ é dita esparsa se o número de elementos não nulos é expresso por $O(n^{1+\gamma})$ para $\gamma < 1$.*

Através do valor de γ é possível obter uma relação com os elementos não nulos da matriz.
$O(n^{1+\gamma}) = nz$, onde nnz são os elementos não-nulos e n a ordem da matriz.
Aplicando $\log_n$ em ambos os lados da igualdade, vem:
$\log_n n^{1+\gamma} = \log_n nz$, aplicando a propriedade dos logaritmos
$(1+\gamma)\log_n n = \log_n nz$, como $\log_n n = 1$, tem-se
$1 + \gamma = \log_n nz$
$\gamma = \log_n nz - 1$
O valor de $\gamma < 1$ se justifica pelo fato do número total de elementos de uma matriz ser n^2.

Por exemplo:

Seja $n = 10^4$, o valor de γ em alguns casos:

γ	$n^{1+\gamma}$
0.1	25.119
0.2	63096
0.3	158489
0.4	398107
0.5	1000000

Tabela 2.1: Tabela com alguns valores para γ

Na tabela acima, com a ordem da matriz $n = 10000$, tem-se alguns valores para γ e a respectiva quantidade de elementos não nulos em cada caso. É fácil perceber a medida que γ aumenta, o número de elementos não nulos aumenta muito rapidamente, por se tratar de uma escala exponencial e quando o valor de γ é igual a 1, tem-se uma matriz totalmente preenchida com elementos não-nulos. Por exemplo, uma matriz de dimensão $n = 36$, com diagrama de esparsidade a seguir:

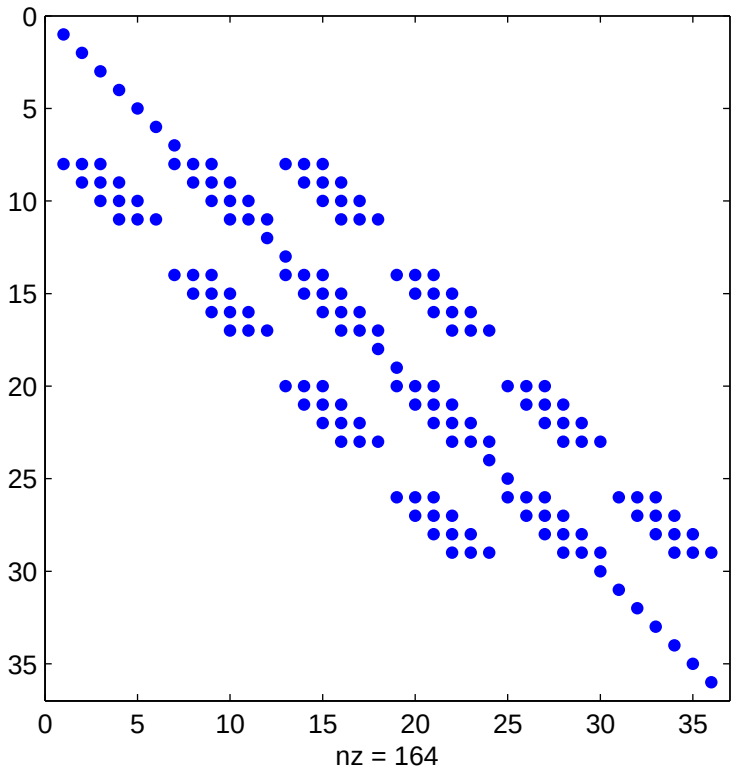


Figura 2.1: Diagrama de esparsidade matriz 36x36

A matriz possui 164 elementos diferentes de zero e ordem $n = 36$, dessa forma pode ser representada como $O(36^{1+0.4231})$, onde $\gamma = 0.4231$.

Definição 2.3 ([8]). *O grau de esparsidade representa a porcentagem de elementos nulos de uma matriz. Dada uma matriz $n \times n$, pode-se calcular o grau de esparsidade (GE) da seguinte forma:*

$$\text{GE}=\frac{\textit{número de elementos nulos}}{\textit{número total de elementos}}.100\%$$

Por exemplo, uma matriz de dimensão $n \times n$, com $n = 100$ que possui 600 elementos não nulos, o grau de esparsidade pode ser calculado. Sendo $n = 100$, tem-se $100 \times 100 = 10000$ elementos na matriz, de modo que $10000 - 600 = 9400$ elementos são nulos

$$\text{GE}=\frac{9400}{10000}.100\% = 94\% \text{ de elementos nulos.}$$

2.2 Estrutura CSR (Compressed Sparse Row)

A idéia principal do formato CSR (Compressed Sparse row) [9], [10] e [11], é transformar uma matriz esparsa em três vetores distintos, denominados AA, IA e JA de modo a não considerar elementos nulos da matriz. A quantidade de elementos não nulos é indicada por nz, de modo que os vetores AA e JA possuem nnz elementos, e o vetor IA, com $n + 1$ elementos, onde n é a ordem da matriz. Finalmente, o vetor JA, armazena o índice relativo a coluna de cada elemento do vetor AA. Considere a matriz abaixo:

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

Utilizando a estrutura CSR aplicada a (2.1), tem-se:

$$\begin{aligned} AA &= \begin{bmatrix} 1 & 4 & 5 & 3 & 2 & 3 & 6 & 8 & 5 & 9 & 4 & 3 & 7 \end{bmatrix} \\ JA &= \begin{bmatrix} 1 & 2 & 1 & 2 & 3 & 2 & 3 & 4 & 3 & 4 & 5 & 4 & 5 \end{bmatrix} \\ IA &= \begin{bmatrix} 1 & 3 & 6 & 9 & 12 & 14 \end{bmatrix} \end{aligned} \tag{2.2}$$

O vetor AA contém os elementos não nulos da matriz, percorrendo-a por linha. O vetor JA armazena o índice da coluna de cada elemento do vetor AA, e o vetor IA, armazena de forma acumulativa a quantidade de elementos não nulos após percorrer a i-ésima linha mais 1, e possui dimensão $(n+1)$, onde n é a ordem da matriz. O valor do primeiro elemento do vetor IA é sempre 1, pois o Matlab®não aceita índice 0.

2.2.1 Algoritmo Estrutura CSR

A estrutura CSR, como mostrada anteriormente, transforma uma matriz em três vetores. O algoritmo abaixo representa essa estrutura implementada em ambiente matlab de forma serial:

```
IA(1) = 1
para   ii = 1, ··· , n
       para   jj = 1, ··· , n
              se   A(ii, jj) ≠ 0
                   cont = cont + 1
                   AA(cont) = A(ii, jj)
                   JA(cont) = jj
              fim
              IA(ii + 1) = cont + 1
       fim
fim
```

O algoritmo acima possui dois laços de repetição, que percorre a matriz por linha, variando o índice da coluna no laço mais interno. É ultilizada também uma estrutura condicional, no qual são armazenados somente os elementos não nulos da matriz. O vetor AA é montado através de um contador, que está no índice desse vetor, de tal forma que

os elementos são armazenados a medida que o contador é ativado. O vetor JA também utiliza o contador, e captura o índice da coluna de cada elemento da matriz A armazenado no vetor AA, e finalmente, o vetor IA, que fornece de forma acumulativa a quantidade de elementos não nulos, vetor que possui o primeiro elemento com o valor 1. Portanto, o vetor IA começa a armazenar os valores dos elementos da matriz a partir de $IA(2)$.

2.3 Estrutura CSC (Compressed Sparse Colunm)

Na estrutura CSC [9], [10], a matriz é percorrida por colunas, onde o armazenamento dos elementos da matriz é feito de forma semelhante ao método *CSR*, utilizando três vetores AA, JA e IA. O vetor AA armazena os elementos não nulos da matriz percorrendo-a por coluna. O vetor JA armazena de forma acumulativa a quantidade de elementos não nulos após percorrer a j-ésima coluna mais 1, e possui dimensão $n+1$, onde o primeiro elemento desse vetor é sempre o valor 1,e finalmente, o vetor IA, que armazena o índice da linha de cada elemento do vetor AA. Nesse tipo de estrutura, os vetores AA e IA possuem nz elementos, ou seja, a quantidade de elementos não nulos da matriz, e o vetor JA possui dimensão $n+1$, onde n é a ordem da matriz. Usando (2.1) do exemplo anterior:

$$A = \begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 5 & 3 & 2 & 0 & 0 \\ 0 & 3 & 6 & 8 & 0 \\ 0 & 0 & 5 & 9 & 4 \\ 0 & 0 & 0 & 3 & 7 \end{bmatrix} \tag{2.3}$$

$$\begin{aligned} AA &= \begin{bmatrix} 1 & 5 & 4 & 3 & 3 & 2 & 6 & 5 & 8 & 9 & 3 & 4 & 7 \end{bmatrix} \\ JA &= \begin{bmatrix} 1 & 3 & 6 & 9 & 12 & 14 \end{bmatrix} \\ IA &= \begin{bmatrix} 1 & 2 & 1 & 2 & 3 & 2 & 3 & 4 & 3 & 4 & 5 & 4 & 5 \end{bmatrix} \end{aligned} \tag{2.4}$$

2.3.1 Algoritmo Estrutura CSC

Fazendo uma análise da estrutura CSC percebe-se que a diferença em relação a CSR está na forma de percorrer a matriz. A estrutura CSR percorre a matriz por linha, vari-

ando o índice da coluna dos elementos, já a estrutura CSC percorre a matriz por coluna, variando o índice das linhas no laço mais interno, conforme mostrado a seguir:

$$
\begin{array}{l}
JA(1) = 1 \\
para \quad jj = 1, \cdots, n \\
\qquad para \quad ii = 1, \cdots, n \\
\qquad\qquad se \quad A(ii, jj) \neq 0 \\
\qquad\qquad\qquad cont = cont + 1 \\
\qquad\qquad\qquad AA(cont) = A(ii, jj) \\
\qquad\qquad\qquad IA(cont) = ii \\
\qquad\qquad fim \\
\qquad\qquad JA(jj + 1) = cont + 1 \\
\qquad fim \\
fim
\end{array}
$$

No algoritmo acima, o vetor AA contém os elementos diferentes de zero da matriz A variando o índice da linha de cada elemento. O vetor IA armazena o índice da linha de cada elemento do vetor AA, e por último, o vetor JA, que possui dimensão $n+1$, sendo n a ordem da matriz utilizada. O vetor JA tem como primeiro valor $JA(1) = 1$ e armazena de forma acumulativa a quantidade de elementos por coluna $n + 1$.

2.4 Estrutura CDS (Compressed Diagonal Storage)

De acordo com [12], [13] e [14], para resolução de sistemas lineares com matrizes que possuem diagonais definidas, ou seja, os elementos da matriz estão distribuídos em diagonais, é interessante utilizar tal estrutura para fazer o armazenamento dos elementos dessas matrizes. Esse tipo de matriz é chamada matriz de banda, onde existem constantes p e q não negativas chamadas *halfbandwidth* a esquerda e a direita, de tal forma que $a(i, j) \neq 0$, se $i - p \leq j \leq i + q$. A estrutura CDS (Compressed Diagonal Storage), consiste em armazenar uma matriz considerando apenas os elementos das diagonais de acordo com [10]. No processo de armazenamento dos formatos de banda, podem existir armazenamentos de elementos nulos na estrutura. Considere a matriz não simétrica definida por:

$$A = \begin{bmatrix} 10 & -3 & 0 & 0 & 0 & 0 \\ 3 & 9 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 0 & 0 & 8 & 7 & 5 & 0 \\ 0 & 0 & 0 & 9 & 9 & 13 \\ 0 & 0 & 0 & 0 & 2 & -1 \end{bmatrix} \tag{2.5}$$

Utilizando a Estrutura CDS, o armazenamento é feito de forma a transformar a matriz A em uma matriz A', com dimensão (6:-1:1), onde 6 é a ordem da matriz A e -1 e 1 o número de diagonais não-nulas, ou seja, neste exemplo tem-se a diagonal principal, uma diagonal acima, uma diagonal abaixo, adjascentes a diagonal principal, utilizando o mapeamento $VAL(i,j) = a_{i,i+j}$, donde:

$$A' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 10 & -3 & 0 & 0 & 0 & 0 & & & & & \\ & 0 & 0 & 0 & 0 & 3 & 9 & 6 & 0 & 0 & 0 & 0 & & & & \\ & & 0 & 0 & 0 & 0 & 7 & 8 & 7 & 0 & 0 & 0 & 0 & & & \\ & & & 0 & 0 & 0 & 8 & 7 & 5 & 0 & 0 & 0 & 0 & 0 & & \\ & & & & 0 & 0 & 0 & 0 & 9 & 9 & 13 & 0 & 0 & 0 & 0 & \\ & & & & & 0 & 0 & 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.6}$$

VAL(: ,-1)	0	3	7	8	9	2
VAL(: ,0)	10	9	8	7	9	-1
VAL(: ,+1)	-3	6	7	5	13	0

(2.7)

sendo p o número de diagonais não-nulas abaixo da diagonal e q o número de diagonais não-nulas acima da diagonal. No exemplo (2.5), $p = 1$ e $q = 1$, onde temos uma diagonal não-nula acima da diagonal principal e uma diagonal não nula abaixo da diagonal principal.

Observe os 2 elementos zeros contidos na matriz (2.7). Esses elementos não fazem parte da Matriz A. Todas as diagonais foram armazenadas sem os respectivos indices de linha e coluna desses elementos, mas sim a posição das diagonais na matriz, sendo a diagonal principal VAL(:,0), a diagonal acima de VAL(:+1)e a diagonal abaixo da principal de VAL(:,-1).

2.4.1 Algoritmo Estrutura CDS

A estrutura CDS possui um diferencial em relação as estruturas mostradas anteriormente que utilizam três vetores para fazer o armazenamento da matriz. Nessa estrutura, de acordo com [12] e [13] são armazenadas as diagonais da matriz A, formando a matriz VAL, de tal forma que estas diagonais são indiciadas pela sua posição abaixo e acima da diagonal principal, que é o ponto de referência, como foi visto anteriormente. Nâo são utilizados vetores para armazenar os índices de linha e coluna dos elementos, que são localizados apenas pela posição da diagonal na matriz. Dessa maneira, o numero de linhas da matriz VAL vai depender diretamente do tipo de matriz utilizada, sendo mais indicada para matrizes de banda, matrizes que possuem diagonais definidas, ou melhor dizendo, matrizes que possuem os elementos não nulos dispostos nas diagonais. O algoritmo dessa estrutura deve reconhecer as diagonais que possuem pelo menos um elemento não nulo e fazer seu armazenamento. Um fato interessante é o armazenamento de elementos nulos na matriz VAL, em casos da diagonal não ser formada totalmente com elementos diferentes de zero.

```
Diagonais acima da principal
Para   k = 1, 2, 3, ··· , (n − 1)
       para   ii = 1, 2, 3, ··· , n
              para   j = ii + k, ··· , n
                     se   j = ii + k
                            Val(k, ii) = A(ii, j)
                            parar
                     fim
              fim
       fim
ind(k) = k
fim
```

Diagonal principal
para $l = 1, 2, 3, \cdots, n$
$Val0(l) = A(l, l)$
$ind0 = 0$
fim
Diagonais abaixo da principal
para $k = 1, 2, 3, \cdots, (n-1)$
para $ii = 1, 2, 3, \cdots, n$
para $j = ii + k, \cdots, n$
se $j = ii + k$
$Val2(n-k, ii+k) = A(j, ii)$
parar
fim
fim
fim
$ind1(k, 1) = -(n - k)$
fim
$VAL = [Val2; Val0; Val]$
$IND = [ind1; ind0; ind]$

```
Removendo os zeros da matriz VAL
II = []
[Mv, nv] = tamanho(VAL)
icount = 0   igg = 0
para   ii = 1, ··· , Mv
       para   jj = 1, ··· , nv
              se   (VAL(ii, jj) == 0)
                     icount = icount + 1
              fim
       fim
              se   (icount == n)
                     igg = igg + 1
                     II(igg) = ii
              fim
icount = 0
fim
VAL(II, :) = []
IND(II, :) = []
```

No algoritmo acima, a matriz é percorrida primeiramente pelas diagonais superiores, ou seja, as diagonais acima da diagonal principal, começando pela diagonal adjacente a principal em direção a diagonal mais distante. No segundo laço, os elementos da diagonal principal são armazenados e finalmente, são os elementos das diagonais inferiores ou abaixo da diagonal principal, percorrendo-a da mesma forma mostrada anteriormente, da diagonal adjascente a principal para a mais distante. Assim, os elementos são alocados na matriz VAL, cuja quantidade de linhas é diretamente proporcional ao número de diagonais não nulas da matriz A. Observa-se no algoritmo as variáveis val,$val0$ e $val2$, onde val e $val2$ são matrizes e suas dimensões são $(n-1)\times n$ e o vetor $val0$, que possui dimensão n. Posteriormente, essas duas matrizes e o vetor se transformam na matriz VAL. Conforme quantidade de diagonais não nulas da matriz A utilizada, o algoritmo remove as linhas totalmente nulas da matriz VAL. O vetor IND por sua vez, armazena a posição relativa da diagonal em relação a diagonal principal.

2.5 Estrutura Skyline

A última estrutura apresentada no presente trabalho é muito utilizada no formato para matrizes simétricas. De acordo com [15], a estrutura skyline é muito utilizada no método de eliminação de Gauss na resolução de sistema de equações algébricas lineares. Mas para que o foco do trabalho fosse mantido, a realização do estudo comparativo entre as estruturas esparsas aplicadas aos métodos iterativos Jacobi e Gauss-seidel, foi estudada e implementada a estrutura skyline aplicada a matrizes não-simétricas. Conforme [16], a estrutura skyline para matrizes não-simétricas consiste em armazenas os elementos percorrendo a matriz até a i-ésima linha, de modo semelhante ao CSR, mas nessa estrutura são armazenados elementos nulos que estejam entre os elementos diferentes de zero, ou seja, se existem elementos nulos em determinada linha que contenha o primeiro e o último elemento não nulos, por exemplo, os demais elementos dessa linha são armazenados, de acordo com o esquema abaixo. Seja a matriz A definida por:

$$A = \begin{bmatrix} 11 & 12 & 0 & 14 & 0 & 0 & 0 & 0 \\ 0 & 22 & 23 & 0 & 25 & 0 & 0 & 0 \\ 31 & 0 & 33 & 34 & 0 & 0 & 0 & 0 \\ 0 & 42 & 0 & 0 & 0 & 45 & 46 & 0 \\ 0 & 0 & 0 & 0 & 55 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 65 & 66 & 67 & 0 \\ 0 & 0 & 0 & 0 & 75 & 0 & 77 & 78 \\ 0 & 0 & 0 & 0 & 0 & 0 & 87 & 88 \end{bmatrix} \tag{2.8}$$

Transformando a matriz no formato Skyline:

$$VAL = \begin{bmatrix} 11 & 12 & 0 & 14 & 22 & 23 & 0 & 25 & 31 & 0 & 33 & 34 & 42 & 0 & 0 & 0 & 45 & 46 & 55 & 65 & 66 \\ 67 & 75 & 0 & 77 & 78 & 87 & 88 \end{bmatrix}$$

$$Rowptr = \begin{bmatrix} 1 & 5 & 9 & 13 & 19 & 20 & 23 & 27 & 29 \end{bmatrix} \tag{2.}$$

$$Fstcol = \begin{bmatrix} 1 & 2 & 1 & 2 & 5 & 5 & 5 & 7 \end{bmatrix}$$

A estrutura Skyline é definida pelos vetores VAL, $Rowptr$ e $Fstcol$, que armazenam informações sobre as matrizes que são transformadas nessa estrutura. O vetor VAL armazena os elementos da matriz percorrendo-a até a i-ésima linha, de tal forma que o primeiro e o último elemento não nulos e todos os demais elementos entre eles são armazenados. Uma característica desse modelo de armazenamento é guardar todos os elementos que estejam entre o primeiro e o último elemento, inclusive os nulos em cada linha. No exemplo acima, pode ser observado que existem zeros alocados no vetor VAL. O vetor $Rowptr$ terá sempre seu primeiro elemento $Rowptr(1) = 1$ e dimensão $(n+1)$, sendo n a ordem da matriz. O vetor $Rowptr$ tem a função de guardar de forma acumulativa a quantidade de elementos por linha armazenados no vetor VAL somando de uma unidade, e por último, o vetor $Fstcol$, que armazena o índice da coluna do primeiro elemento diferente de zero em cada linha, informando ao algoritmo quando começa o primeiro elemento não nulo em cada linha.

2.5.1 Algoritmo Estrutura Skyline

```
Para   ii = 1, 2, 3, ··· , n
para   jj = 1, 2, 3, ··· , n
            se   A(ii, jj) ≠ 0
                 Fstcol = [Fstcol, jj]
                 parar
            fim
       fim
fim
cont = 0
para   iii = 1, 2, 3, ··· , n
       para   jjj = 1, 2, 3, ··· , n
            se   A(iii, jjj) ≠ 0
                 d(iii) = jjj
            fim
       fim
       para   e = Fstcol(iii), ··· , d(iii)
            cont = cont + 1
            VAL(cont) = A(iii, e)
       fim
       Rowptr(iii + 1) = cont + 1
fim
```

O algoritmo acima possui duas estruturas de repetição independentes, onde a primeira captura o primeiro elemento diferente de zero em cada linha, ao mesmo tempo que armazena o índice da coluna desse elemento. O segundo laço armazena o último elemento não nulo em cada linha em um vetor auxiliar d. O vetor VAL é montado numa estrutura de repetição que é controlada pelos vetores $Fstcol$ e $Rowptr$, informando ao algoritmo quais os elementos serão armazenados no vetor VAL.

Capítulo 3

Métodos numéricos para solução de sistemas algébricos lineares

3.1 Método da Iteração (Jacobi)

O método da Iteração também conhecido como Jacobi [2], consiste em resolver um sistema linear de forma a obter a solução aproximada do problema, através de uma solução inicial arbitrária X^0, e com uma sequência de aproximações X^k a partir de X^{k-1}. Tais aproximações tendem para a solução exata do problema na forma $\lim_{k\to\infty} X^k = \overline{X}$, onde $\overline{X}$ é a solução exata do sistema. Seja o sistema $Ax = b$, escrito na forma matricial:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2(n-1)} & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3(n-1)} & a_{3n} \\ & \vdots & & \ddots & \vdots & \\ a_{(n-1)1} & a_{(n-1)2} & a_{(n-1)3} & \cdots & a_{(n-1)(n-1)} & a_{(n-1)n} \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{n(n-1)} & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{(n-1)} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{(n-1)} \\ b_n \end{bmatrix} \tag{3.1}$$

onde A é uma matriz não-singular.

Primeiramente, o método de Jacobi consiste em explicitar as incógnitas $X = x_1, x_2, \cdots, x_n$, isolando as mesmas no lado esquerdo da igualdade:

$$\begin{cases} x_1 = & (0x_{11} + ax_{12} + ax_{13} + \cdots + ax_{1n})/a_{11} \\ x_2 = & (ax_{12} + 0x_{22} + ax_{23} + \cdots + ax_{2n})/a_{22} \\ x_3 = & (ax_{31} + ax_{32} + 0x_{33} + \cdots + ax_{3n})/a_{33} \\ & \vdots \\ x_n = & (ax_{n1} + ax_{n2} + ax_{n3} + \cdots + 0x_{nn})/a_{nn} \end{cases} \tag{3.2}$$

Para explicitar as variáveis, a matriz A se transforma na matriz α e o vetor b resulta no vetor β como segue:

$$\begin{cases} \alpha_{ij} = & -\dfrac{a_{ij}}{a_{ii}}, se \quad i \neq j \\ \alpha_{ij} = & 0, se \quad i = j \\ \beta_i = & \dfrac{b_i}{a_{ii}} \end{cases} \tag{3.3}$$

$$\alpha = \begin{bmatrix} 0 & \alpha_{12} & \alpha_{13} & \cdots & \alpha_{(1,n-1)} & \alpha_{1,n} \\ \alpha_{21} & 0 & \alpha_{23} & \cdots & \alpha_{(2,n-1)} & \alpha_{2,n} \\ \alpha_{31} & \alpha_{32} & 0 & \cdots & \alpha_{(3,n-1)} & \alpha_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \alpha_{(n-1,1)} & \alpha_{(n-1,2)} & \alpha_{(n-1,3)} & \cdots & 0 & \alpha_{(n-1,n)} \\ \alpha_{n,1} & \alpha_{n,2} & \alpha_{n,3} & \cdots & \alpha_{(n,n-1)} & 0 \end{bmatrix} \tag{3.4}$$

$$\beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \vdots \\ \beta_{(n-1)} \\ \beta_n \end{bmatrix} \tag{3.5}$$

Calculando o valor de cada componente do vetor X, tem-se:

$$x_i^k = \beta_i + \sum_{j=1}^{n} \alpha_{ij} x_j^{k-1} \tag{3.6}$$

onde:

$i = 1, 2, 3, ..., n$

$j = 1, 2, 3, ..., n$

$k = 1, 2, 3, ...$ o número de iterações.

ou na forma matricial:

$$X^k = \beta + \alpha X^{k-1} \tag{3.7}$$

3.1.1 Algoritmo método da Iteração (Jacobi)

$$
\begin{array}{l}
Enquanto \quad erro \geq tol \quad e \quad it \neq itermax \\
it = it + 1 \\
\qquad para \quad k = 1, 2, 3, \cdots, n \\
\qquad s = 0 \\
\qquad\qquad para \quad j = 1, 2, 3, \cdots, n \\
\qquad\qquad s = s + \alpha(k, j) * X0(j); \\
\qquad\qquad fim \\
\qquad Q(k) = s; \\
\qquad X(k) = Q(k) + \beta(k) \\
\qquad erroL1(k) = valor\ absoluto(X(k) - X0(k)) \\
\qquad L2sum = L2sum + (X(k) - X0(k))^2 \\
\qquad fim \\
erro = max(erroL1) \\
resL1(it) = log10(erro/n) \\
resL2(it) = log10(sqrt(L2sum/n)) \\
X0 = X \\
L2sum = 0 \\
fim
\end{array}
$$

O algoritmo do método de Jacobi funciona com o critério de parada, onde a variável *tol* é a tolerância de erro desejada do vetor solução, e a variável *erro* onde é atribuída um valor, como por exemplo 1. Dessa forma, enquanto a variável *erro*, que é renovada a cada iteração, não possuir valor absoluto menor que a tolerância, as iterações continuam. No caso da condição citada não ser alcançada, foi estabelecido o número de iterações máximas para que o algoritmo não realize infinitas iterações. A variável *itermax* estabelece o número máximo de iterações.

3.2 Método de Gauss-Seidel

O método de Gauss-Seidel, conforme visto em [2], pode ser classificado como uma modificação do método da Iteração(Jacobi). A diferença está na utilização do vetor solução X da seguinte forma: x_1 no passo atual para calcular x_2, x_1 e x_2 no passo atual para calcular x_3 e assim sucessivamente, ao contrário do método de Jacobi, onde é calculada

a solução no passo atual usando todos os valores calculados na iteração anterior. Utilizando essa estrutura, em muito dos casos observa-se a convergência alcançada com menor número de iterações do que no método de Jacobi, mas o inverso também pode acontecer, dependendo da matriz do sistema utilizado, a convergência no método de Jacobi e divergência no método de Gauss-Seidel e vice-versa. Seja o sistema linear dado por $Ax = b$, escrito na forma matricial como segue:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2(n-1)} & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3(n-1)} & a_{3n} \\ & \vdots & & \ddots & \vdots & \\ a_{(n-1)1} & a_{(n-1)2} & a_{(n-1)3} & \cdots & a_{(n-1)(n-1)} & a_{(n-1)n} \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{n(n-1)} & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{(n-1)} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{(n-1)} \\ b_n \end{bmatrix} \tag{3.8}$$

Segue o mesmo passo realizado no método da Iteração (Jacobi), onde é feita a explicitação das variáveis, obtendo a matriz α e o vetor β, chegando na expressão:

$$\begin{cases} x_1^{k+1} = \beta_1 + \sum_{j=1}^{n} \alpha_{1j} x_j^k \\ x_2^{k+1} = \beta_2 + \alpha_{21} x_1^{k+1} + \sum_{j=2}^{n} \alpha_{2j} x_j^k \\ x_i^{k+1} = \beta_i + \sum_{j=1}^{i-1} \alpha_{ij} x_j^{k+1} + \sum_{j=i}^{n} \alpha_{ij} x_j^k \\ x_n^{k+1} = \beta_n + \sum_{j=1}^{n-1} \alpha_{nj} x_j^{k+1} + \alpha_{nn} x_n^k \end{cases} \tag{3.9}$$

onde:
$i = 1, 2, 3, ..., n$
$j = 1, 2, 3, ..., n$
$k = 1, 2, 3, ...$ o número de iterações.

Existem condições suficientes mas não necessárias para a convergência desses métodos, que serão abordados posteriormente neste trabalho.

3.2.1 Algoritmo Método de Gauss-Seidel

$Enquanto \quad erro \geq tol \quad e \quad it \neq itermax$
$it = it + 1$
$\quad para \quad k = 1, 2, 3, \cdots, n$
$\quad s = 0$
$\quad\quad para \quad j = 1, 2, 3, \cdots, n$
$\quad\quad s = s + \alpha(k, j) * X0(j);$
$\quad\quad fim$
$\quad Q(k) = s;$
$\quad X(k) = Q(k) + \beta(k)$
$\quad erroL1(k) = valor\ absoluto(X(k) - X0(k))$
$\quad L2sum = L2sum + (X(k) - X0(k))^2$
$\quad X0(k) = X(k)$
$\quad fim$
$erro = max(erroL1)$
$resL1(it) = log10(erro/n)$
$resL2(it) = log10(sqrt(L2sum/n))$
$X0 = X$
$L2sum = 0$
fim

A diferença encontrada no método de Gauss-Seidel em relação ao método de Jacobi é a atualização da variável $X0$ ao final de cada laço de repetição que controla as componentes do vetor X.

3.3 Convergência dos Métodos da Iterativos

Nesta seção é apresentada uma condição suficiente sobre a matriz do sistema linear, para que a sequência de aproximações determinada pelos métodos iterativos convirja para a solução exata do sistema linear considerado. Para isto, são necessários alguns conceitos da teoria da álgebra linear matricial. Uma exposição mais detalhada desta teoria pode ser encontrada em [2], capítulo 7.

Definição 3.1. *Uma norma em um espaço vetorial de matrizes é dita canônica quando as seguintes condições ocorrem.*

1. *Se* $A = [a_{ij}]$ *então* $|a_{ij}| \leq \|A\|, \forall\ i, j.$

2. *Da inequação* $|A| \leq |B|$ *segue que* $\|A\| \leq \|B\|$, *onde* $A = [a_{ij}]$, $B = [b_{ij}]$, $|A| = [|a_{ij}|]$ *e* $b = [|b_{ij}|]$

agora, para apresentar uma condição suficiente, considere o sistema linear na forma

$$X^k = \beta + \alpha.X^{k-1} \tag{3.10}$$

com $\alpha = [\alpha_{ij}]$, $\beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \vdots \\ \beta_n \end{bmatrix}$ *e* $X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$ *Tem-se o seguinte resultado (conforme [2])*

Teorema 3.1. *Para o sistema linear escrito na forma reduzida* (3.10)*, o processo de iteração converge para a solução exata do sistema se alguma norma canônica de* α *for menor que a unidade, isto é,* $\|\alpha\| < 1$*, é uma condição suficiente para convergência do processo iterativo.*

Demonstração. Iniciando com um vetor solução inicial X^0, constrói-se uma sequência de aproximações:

$$\begin{aligned} X^{(1)} &= \beta + \alpha X^{(0)} \\ X^{(2)} &= \beta + \alpha X^{(1)} \\ X^{(3)} &= \beta + \alpha X^{(2)} \\ &\vdots \\ X^{(k)} &= \beta + \alpha X^{(k-1)} \end{aligned} \tag{3.11}$$

donde

$$X^{(k)} = (E + \alpha + \alpha^2 + \cdots + \alpha^{(k-1)})\beta + \alpha^k X^{(0)} \tag{3.12}$$

onde E é a matriz identidade de ordem n. Como $\|\alpha\| < 1$, tem-se $\left\|\alpha^k\right\| \to 0$ quando $k \to \infty$, segue que (veja seção 7.10 de [2])

$$\lim_{k\to\infty} \alpha^k = 0 \tag{3.13}$$

e

$$\lim_{k\to\infty} (E + \alpha + \alpha^2 + \cdots + \alpha^{(k-1)}) = \sum_{k=0}^{\infty} \alpha^k = (E - \alpha)^{-1} \tag{3.14}$$

Logo, aplicando o limite quando $k \to \infty$ em (3.12),tem-se:

$$X = \lim_{k\to\infty} X^k = (E - \alpha)^{-1}\beta \tag{3.15}$$

Dessa forma, está provada a convergência do processo iterativo. Além disso, a partir de (3.15), tem-se:

$$(E - \alpha)X = \beta$$
$$\text{ou}$$
$$X = \alpha X + \beta$$

o que mostra que o limite do vetor X é a solução do sistema linear, como a matriz $(E-\alpha)$ é não-singular, a solução X é única.

□

Corolário 3.2. *O método iterativo para o sistema $X^k = \beta + \alpha X^{k-1}$ converge se:*

1. $\|\alpha\|_m = \max\limits_{1\le i\le n} \sum\limits_{j=1}^{n} |\alpha_{ij}| < 1$
2. $\|\alpha\|_l = \max\limits_{1\le j\le n} \sum\limits_{i=1}^{n} |\alpha_{ij}| < 1$
3. $\|\alpha\|_k = \sqrt{\sum\limits_{i=1}^{n}\sum\limits_{j=1}^{n} |\alpha_{ij}|^2} < 1$

Onde as normas acima são canônicas e definidas anteriormente. Em particular, o método iterativo definitivamente converge se os elementos da matriz α satisfazem a inequação

$|\alpha_{ij}| < \frac{1}{n}$, com $1 \leq i \leq n$ e $1 \leq j \leq n$.

onde n é o número de incógnitas do sistema $X^k = \beta + \alpha X^{k-1}$.

Demonstração: Segue diretamente do Teorema(3.1) e pelo fato das três normas da matriz α serem normas canônicas, conforme [2] capítulo 7.

Definição 3.2. *Seja A uma matriz Quadrada não singular, é chamada diagonalmente dominante se atender umas das condições:*

$$|a_{jj}| > \sum_{j=1}^{n} |a_{ij}|, \quad para \quad i \neq j \quad i = 1, 2, 3, \cdots, n. \tag{3.16}$$

$$|a_{ii}| > \sum_{i=1}^{n} |a_{ij}|, \quad para \quad j \neq i \quad j = 1, 2, 3, \cdots, n. \tag{3.17}$$

Para o critério da soma das linhas, o item (1) do Corolário (3.2) é diretamente satisfeito. Com relação ao critério da soma das colunas, considerando o sistema $\sum_{j=1}^{n} a_{ij}x_j = b_i$, com $(i = 1, 2, 3, \cdots, n)$, faça $x_i = \frac{z_i}{a_{ii}}$, com $(i = 1, 2, 3, \cdots, n)$. Assim, segue que z_i são as variáveis desconhecidas do sistema $\sum_{j=1}^{n} \frac{a_{ij}}{a_{jj}} z_j = b_i$, para o qual os processos de iteração convergem se e somente se convergem para o sistema original. Colocando este último sistema na forma modificada $X^k = \beta + \alpha X^{k-1}$, tem-se que o critério da soma das colunas implica no item (2) do Corolário (3.2) para esta matriz α.

3.3.1 Normas utilizadas para convergência

As normas utilizadas para convergência dos métodos iterativos apresentadas neste trabalho, de acordo com [1], são as normas L1 e L2 ilustradas abaixo:

$$L_1 = \max \left| (x_i^{k+1} - x_i^k) \right|, \quad \text{com } 1 \leq i \leq n. \tag{3.18}$$

$$L_2 = \sqrt{\sum_{i=1}^{n} \max (x_i^{k+1} - x_i^k)^2}, \quad \text{com } 1 \leq i \leq n. \tag{3.19}$$

onde x_i^{k+1} é a componente do vetor solução X no passo de iteração atual, x_i^k, a componente do vetor solução X no passo de iteração anterior. Tais normas são implementadas no trabalho na forma a seguir com objetivo de melhor visualização da convergência, com a utilização das normas em escala logarítmica, onde nos algoritmos implementados a variável utilizada são chamadas $resL1$ e $resL2$

$$resL1 = \log \frac{\max \left| (x_i^{k+1} - x_i^k) \right|}{n}, \quad \text{com } 1 \leq i \leq n. \tag{3.20}$$

$$resL2 = \log \sqrt{\frac{\sum_{i=1}^{n} \max (x_i^{k+1} - x_i^k)^2}{n}}, \quad \text{com } 1 \leq i \leq n. \tag{3.21}$$

onde n é a ordem do sistema linear.

3.3.2 Critério de parada métodos Jacobi e Gauss-Seidel

Computacionalmente, todo método numérico necessita de um critério de parada, para que o algoritmo gere o resultado com a precisão desejada. Esse critério no método da Iteração(Jacobi) é feito usando expressão:

$$max \left| X^k - X^{k-1} \right| \langle \epsilon \tag{3.22}$$

onde $\epsilon \rangle 0$ é a tolerância entre a solução atual X^k e a solução anterior X^{k-1}.

3.3.3 Condicionamento de matrizes

Segundo [17] o condicionamento de matrizes possui o seguinte aspecto teórico. Considere o sistema linear escrito na forma $Ax = b$, onde A é uma matriz não-singular e b um vetor não-nulo. Através da perturbação no valor dos elementos da matriz A na forma $A+\delta A$, onde δA é o valor dessa perturbação, (a teoria mais detalhada pode ser encontrada em [17] capítulo 2), o sistema passa a ser escrito na forma $(A + \delta A)(x + \delta x_A) = b$, onde δx_A é a variação da solução exata do sistema linear, resultando na expressão para o erro relativo, onde $\|.\|$ é uma norma matricial:

$$\frac{\|\delta x_A\|}{\|x + \delta x_A\|} \leq \|A^{-1}\| \, \|A\| \, \frac{\|\delta A\|}{\|A\|} \tag{3.23}$$

Da mesma maneira, perturbando o vetor dos termos independentes b, tem-se a seguinte expressão:

$$\frac{\|\delta x_b\|}{\|x\|} \leq \|A^{-1}\| \, \|A\| \, \frac{\|\delta b\|}{\|b\|} \tag{3.24}$$

onde o valor de $\|A^{-1}\| \, \|A\|$ é a máxima mudança relativa da solução exata para um sistema linear perturbado. Esse valor é chamado de condicionamento da matriz A representado por:

$$cond(A) = \|A^{-1}\| \, \|A\| \tag{3.25}$$

onde $\|A\|$ é a norma da matriz A do sistema linear e $\|A^{-1}\|$ é a norma da matriz inversa de A.

Na teoria, quanto mais próximo de 1, melhor o condicionamento de uma matriz. Em uma matriz bem condicionada, pequenas perturbações no vetor b resultam em pequenas variações no valor da solução exata do sistema linear. Em contrapartida, numa matriz mal condicionada, pequenas perturbações no vetor b resultam em grandes variações na solução exata do sistema. O condicionamento de matrizes merece importância, pois o uso do computador resulta em erros de arredondamento ("round-off"), já citado no presente

trabalho. O erro pode mudar drasticamente a solução exata do sistema linear, quando a matriz A dos coeficientes for mal condicionada .

3.4 Método Jacobi Estrutura CSR

Neste trabalho são apresentadas algumas estruturas para armazenamento de matrizes esparsas, de acordo com [9] , [10], [12] e [13]. Após o estudo do funcionamento dessas estruturas, foi possível a implementação das mesmas aos métodos iterativos, proporcionando diminuição expressiva do tempo de processamento e quantidade de memória utilizada para armazenamento das matrizes do sistema. A primeira estrutura a ser aplicada nos métodos iterativos é a CSR, que consiste em transformar a matriz A do sistema $Ax = b$ em três vetores distintos: AA, IA e JA. Analizando o método de Jacobi, observa-se que dentro do algoritmo onde são realizadas as iterações o produto matriz-vetor, onde cada elemento da matriz α, que foi obtida através de uma operação elementar na matriz A, é relacionado com o vetor solução inicial X^0, ou o vetor solução no passo anterior X^{k-1}, pois a cada iteração atual X^k é utilizado o vetor solução do passo anterior, como pode ser observado abaixo:

$$X^k = \beta + \alpha X^{k-1} \tag{3.26}$$

Na multiplicação matriz-vetor, onde n é a ordem do sistema, tem-se:

$$x_i^k = \beta_i + \sum_{j=1}^{n} \alpha_{ij} x_j^{k-1}, onde \quad i = 1, 2, 3..., n \tag{3.27}$$

a estratégia utilizada para incorporar a estrutura CSR no método de Jacobi é substituir a matriz α pelo vetor AA, vetor que contém os elementos diferentes de zero percorrendo a matriz por linha, ao mesmo tempo que os vetores JA e IA controlam o índice da coluna de cada elemento e a quantidade de elementos por linha de forma acumulativa, respectivamente. Dessa forma, é reduzido o tempo de processamento e operações desnecessárias com elementos nulos. Apresentamos a seguir o algoritmo do método de Jacobi com a técnica de armazenamento de matrizes esparsas CSR.

$Enquanto \quad erro \geq tol \quad e \quad it \neq itermax$
$it = it + 1$
$\quad para \quad k = 1, 2, 3, \cdots, n$
$\quad s = 0$
$\quad c = IA(k+1) - IA(k)$
$\quad\quad para \quad ii = 1, \cdots, c$
$\quad\quad cont = cont + 1$
$\quad\quad j = JA(cont)$
$\quad\quad s = s + AA(cont) * X0(j);$
$\quad\quad fim$
$\quad Q(k) = s;$
$\quad X(k) = Q(k) + B(k)$
$\quad erroL1(k) = valor\ absoluto(X(k) - X0(k))$
$\quad L2sum = L2sum + (X(k) - X0(k))^2$
$\quad fim$
$erro = max(erroL1)$
$resL1(it) = log10(erro/n)$
$resL2(it) = log10(sqrt(L2sum/n))$
$X0 = X$
$L2sum = 0$
fim

No algoritmo acima, o vetor IA fornece a quantidade de elementos diferentes de zero por linha, e portanto, o laço mais externo realiza somente as repetições para esses elementos. No laço mais interno existe um contador que percorre o vetor AA, ao mesmo tempo que o índice da coluna de cada elemento é fornecida pelo vetor JA. Assim, as operações são feitas somente nos elementos não nulos da matriz do sistema.

3.5 Método Jacobi Estrutura CSC

Nesse tipo de estrutura, os elementos da matriz são armazenados percorrendo-a por coluna, ou seja, a variação do índice das linhas está no laço mais interno na montagem do algoritmo. Apresentamos o algoritmo do método de Jacobi com estrutura de armazenamento de matrizes esparsas CSC.

$Enquanto \quad erro \geq tol \quad e \quad it \neq itermax$
$it = it + 1$
$Q = zeros(n, 1)$
$cont = 0$
$\quad para \quad k = 1, 2, 3, \cdots, n$
$\quad c = JA(k + 1) - JA(k)$
$\quad\quad para \quad i = 1, \cdots, c$
$\quad\quad cont = cont + 1$
$\quad\quad s = IA(cont)$
$\quad\quad Q(s) = Q(s) + AA(cont) * X0(k)$
$\quad\quad fim$
$\quad fim$
$\quad\quad para \quad f = 1, 2, 3, \cdots, n$
$\quad\quad X(f) = Q(f) + \beta(f)$
$\quad\quad erroL1(f) = valor\ absoluto(X(f) - X0(f))$
$\quad\quad L2sum = L2sum + (X(f) - X0(f))^2$
$\quad\quad fim$
$erro = max(erroL1)$
$resL1(it) = log10(erro/n)$
$resL2(it) = log10(sqrt(L2sum/n))$
$X0 = X$
$L2sum = 0$
fim

No algoritmo acima, a variável c contém o número de elementos não nulos por coluna fornecidos pelo vetor JA. A variável s informa o índice da linha de cada elemento no vetor IA, que é percorrido através de um contador, onde simultaneamente os elementos diferentes de zero são fornecidos pelo vetor AA. Dessa forma, o vetor Q gera a solução do produto matriz-vetor, que é somado ao vetor β em cada iteração. Esse mecanismo é repetido de acordo com o número de iterações necessárias para convergência da solução com a tolerância de erro desejada.

3.6 Método Jacobi estrutura CDS

O método de jacobi com a estrutura CDS consiste na implementação da estrutura esparsa CDS mostrada neste trabalho ao método numérico iterativo de Jacobi. Uma propriedade da estrutura CDS é sua aplicação em matrizes de banda, matrizes que possuem diagonais definidas, com os elementos não nulos concentrados nas diagonais. Nessa estrutura, as diagonais após armazenadas em vetores, são utilizadas na multiplicação matriz-vetor $\alpha * X0$, onde α é previamente transformada na estrutura CDS, sendo esta a principal diferença do método de Jacobi para matrizes densas. Apresentamos o algoritmo do método de Jacobi com a estrutura de armazenamento de matrizes esparsas CDS.

$Enquanto \quad erro \geq tol \quad e \quad it \neq itermax$
$it = it + 1$
z=$tamanho(ind)$
$d = 0$
$Q = zeros(n)$
    $para \quad jj = 1, 2, 3, \cdots, z$
    $d = ind(jj)$
        $para \quad ii$ =$valor\ máximo\ entre(1, -d + 1), \cdots,$$valor\ mínimo\ entre(n, n - d)$
        $Q(ii) = Q(ii) + val(ii, jj) * X0(ii + d)$
        fim
    fim
        $para \quad k = 1, 2, 3, \cdots, n$
        $X(k) = Q(k) + \beta(k)$
        $erroL1(k)$ =$valor\ absoluto(X(k) - X0(k))$
        $L2sum = L2sum + (X(k) - X0(k))^2$
        fim
$erro = max(erroL1)$
$resL1(it) = log10(erro/n)$
$resL2(it) = log10(sqrt(L2sum/n))$
$X0 = X$
$L2sum = 0$
fim

É observado no algoritmo acima a variável z, que armazena o tamanho do vetor ind, limitando o laço de repetição logo abaixo. Isso se deve ao fato da estrutura CDS ter a quantidade de vetores para o armazenamento dos elementos proporcional ao número de diagonais não nulas da matriz utilizada. A variável d contém o valor de cada elemento do vetor ind, indicando a posição de cada diagonal não nula. O segundo laço do algoritmo possui um diferencial nos valores inicial e final do laço, pois são fornecidos pelos valores máximos e mínimos. Logo após, é realizada a multiplicação matriz vetor, gerando o vetor Q. Para finalizar, o vetor Q é somado ao vetor β, tendo como resultado o vetor solução no passo atual X.

3.7 Método Jacobi Estrutura Skyline

O método de jacobi usando a estrutura esparsa skyline, consiste na utilização da estrutura skyline para matrizes não simétricas mostradas neste trabalho, implementada ao método numérico de Jacobi. O uso da estrutura Skyline para matrizes simétricas não foi utilizada nesse trabalho, por motivo de todas as estruturas implementadas são para utilização em qualquer tipo de matriz, de tal forma que o resultado numérico para todas as estruturas são idênticas, sendo a diferença observada no tempo de processamento e quantidade de memória utilizada para armazenamento dos elementos das matrizes. Apresentamos o algoritmo do método de Jacobi com a técnica de armazenamento de matrizes esparsas Skyline.

$Enquanto \quad erro \geq tol \quad e \quad it \neq itermax$
$it = it + 1$
$Q = zeros(n, 1)$
$\quad Para \quad k = 1, 2, 3, \cdots, n$
$\quad col = fstcol(k)$
$\quad ii = rowptr(k)$
$\quad iiend = rowptr(k + 1) - 1$
$\quad\quad para \quad j = ii, \cdots, iiend$
$\quad\quad Q(k) = Q(k) + VAL(j) * X0(col + j - ii)$
$\quad\quad fim$
$\quad X(k) = Q(k) + B(k)$
$\quad erroL1(k) = abs(X(k) - X0(k))$
$\quad L2sum = L2sum + (X(k) - X0(k))^2$
$\quad fim$
$erro = max(erroL1)$
$resL1(it) = log10(erro/n)$
$resL2(it) = log10(sqrt(L2sum/n))$
$X0 = X$
$L2sum = 0$
fim

No algoritmo acima, a variável *col* armazena o índice da coluna do primeiro elemento diferente de zero em cada linha, ii fornece a quantidade de elementos entre o primeiro e último diferente de zero. A partir desses valores, o vetor VAL é multiplicado com o vetor $X0$, chegando ao resultado idêntico ao método iterativo sem a estrutura esparsa.

3.8 Método de Gauss-Seidel Estrutura CSR

Com o objetivo de obter eficiência computacional, é implementada a estrutura CSR ao método de Gauss-Seidel. A estrutura CSR, como mostrada anteriormente, transforma uma matriz em três vetores. Segue o algoritmo do método de Gauss-Seidel com estrutura CSR.

$Enquanto \quad erro \geq tol \quad e \quad it \neq itermax$
$it = it + 1$
$cont = 0$
    $para \quad k = 1, 2, 3, \cdots, n$
    $c = IA(k+1) - IA(k)$
    $s = 0$
        $para \quad i = 1, \cdots, c$
        $cont = cont + 1$
        $j = JA(cont)$
        $s = s + AA(cont) * X0(j)$
        fim
    $Q(k) = s$
    $X(k) = Q(k) + B(k)$
    $erroL1(k) = valor\ absoluto(X(k) - X0(k))$
    $L2sum = L2sum + (X(k) - X0(k))^2$
    $X0(k) = X(k)$
fim
$erro = max(erroL1)$
$resL1(it) = log10(erro/n)$
$resL2(it) = log10(sqrt(L2sum/n))$
$X0 = X$
$L2sum = 0$
fim

Observa-se que o esquema para implementar o método de Gauus-Seidel com estrutura CSR é bem semelhante ao método de Jacobi. A variável c armazena o número de elementos diferentes de zero percorrendo a matriz por linha, que posteriormente será o valor que limita o laço de repetição. Logo depois, o vetor JA guarda o índice da coluna de cada elemento, que é armazenado na variável j. Dessa forma, a multiplicação matriz-vetor é realizada e a cada repetição do laço os elementos do vetor solução X são atualizados para a próxima iteração, sendo esta a diferença do método de Gauss-Seidel com o método de Jacobi.

3.9 Método de Gauss-Seidel Estrutura CSC

Ao implementar o método de Gauss-Seidel com estrutura CSC, tendo em vista que esse tipo de estrutura esparsa percorre a matriz por coluna, não foi obtida eficiência computacional com o sistema linear na forma $Ax = b$. Isso ocorre por que a forma de percorrer os elementos da matriz na formato CSC não se adqua ao método de Gauss-Seidel, que necessita da componente x_1 do vetor solução para calcular x_2, x_1 e x_2 para calcular x_3, e assim sucessivamente. Para resolver esse problema, a solução é transformar o sistema $Ax = b$, no esquema $x^T A^T = b^T$, conforme esquema abaixo:

$$\begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_n \end{bmatrix} \cdot \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & a_{4n} \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}^T = \begin{bmatrix} b_1 & b_2 & b_3 & \dots & b_n \end{bmatrix} \quad (3.28)$$

O sistema nessa forma se adequa melhor a estrutura do método de Gauss-Seidel, que necessita a atualização de cada elemento do vetor solução dentro do mesmo passo de iteração. Apresentamos o algoritmo do método de Gauss-Seidel com a estrutura CSC.

$Enquanto \quad erro \geq tol \quad e \quad it \neq itermax$
$it = it + 1$
$cont = 0$
$\quad para \quad k = 1, 2, 3, \cdots, n$
$\quad c = JA(k+1) - JA(k)$
$\quad\quad para \quad i = 1, \cdots, c$
$\quad\quad cont = cont + 1$
$\quad\quad s = IA(cont)$
$\quad\quad Q(k) = Q(k) + AA(cont) * X0(s)$
$\quad\quad fim$
$\quad X(k) = Q(k) + \beta(k)$
$\quad erroL1(k) = valor\ absoluto(X(k) - X0(k))$
$\quad L2sum = L2sum + (X(k) - X0(k))^2$
$\quad X0(k) = X(k)$
$\quad fim$
$erro = max(erroL1)$
$resL1(it) = log10(erro/n)$
$resL2(it) = log10(sqrt(L2sum/n))$
$X0 = X$
$L2sum = 0$
fim

No método de Gauss-Seidel com estrutura CSC, a variável c recebe o valor da quantidade de elementos diferentes de zero por coluna da matriz α, que foi transformada no vetor AA. A variável s recebe o índice da linha de cada elemento do vetor AA, utilizando um contador para percorrer esse vetor. Logo após é feita a multiplicação matriz-vetor $\alpha * X0$, o vetor Q contém o resultado dessa multiplicação, que somado ao vetor β resulta na iteração do vetor solução X no passo atual.

3.10 Método Gauss-Seidel Estrutura CDS

A estrutura esparsa CDS, nessa etapa do trabalho, é implementada ao método de Gauss-Seidel, de tal forma que a matriz A é transformada na matriz VAL, que contém a quantidade de vetores proporcional a quantidade de diagonais não nulas. Segue abaixo o algoritmo:

$Enquanto \quad erro \geq tol \quad e \quad it \neq itermax$
$it = it + 1$
$z = length(ind)$
$d = 0$
$Q = zeros(n)$
$\quad para \quad ll = 1, 2, 3, \cdots, n$
$\quad\quad para \quad jj = 1, 2, 3, \cdots, z$
$\quad\quad d = ind(jj)]$
$\quad\quad\quad para \quad ii = (valor\ máximo(1, -d+1)) : valor\ mínimo(n, n-d)$
$\quad\quad\quad\quad se \quad ii == ll$
$\quad\quad\quad\quad Q(ii) = Q(ii) + val(ii, jj) * X0(ii + d)$
$\quad\quad\quad\quad parar$
$\quad\quad\quad\quad fim$
$\quad\quad\quad fim$
$\quad\quad fim$
$\quad X(ll) = Q(ll) + B(ll)$
$\quad erroL1(ll) = valor\ absoluto(X(ll) - X0(ll))$
$\quad L2sum = L2sum + (X(ll) - X0(ll))^2$
$\quad X0(ll) = X(ll)$
$\quad fim$
$erro = max(erroL1)$
$resL1(it) = log10(erro/n)$
$resL2(it) = log10(sqrt(L2sum/n))$
$X0 = X$
$L2sum = 0$
fim

No algoritmo, a variável z armazena o tamanho do vetor ind, que possui dimensão igual ao número de diagonais não-nulas da matriz do sistema. A variável d recebe o valor da posição de cada diagonal não nula que posteriomente vai ser percorrida pelo laço de repetição. Dessa forma, o produto matriz-vetor $\alpha * X0$ é realizado com a transformação da matriz α na matriz VAL, que contém somente as diagonais que possuem ao menos um elemento não-nulo.

3.11 Método Gauss-Seidel Estrutura Skyline

O último algoritmo a ser implementado no trabalho é o método de Gauss-Seidel com estrutura Skyline. Segue o esquema de implementação:

$Enquanto \quad erro \geq tol \quad e \quad it \neq itermax$
$it = it + 1$
$\quad para \quad k = 1, 2, 3, \cdots, n$
$\quad col = fstcol(k)$
$\quad ii = rowptr(k)$
$\quad iiend = rowptr(k+1) - 1$
$\quad\quad para \quad j = ii, \cdots, iiend$
$\quad\quad Q(k) = Q(k) + val(j) * X0(col + j - ii)$
$\quad\quad fim$
$\quad X(k) = Q(k) + B(k)$
$\quad erroL1(k) = valor\ absoluto(X(k) - X0(k))$
$\quad L2sum = L2sum + (X(k) - X0(k))^2$
$\quad X0(k) = X(k)$
$\quad fim$
$erro = max(erroL1)$
$resL1(it) = log10(erro/n)$
$resL2(it) = log10(sqrt(L2sum/n))$
$X0 = X$
$L2sum = 0$
fim

Capítulo 4

Análise dos Resultados

4.1 Matriz Esparsa Aleatória

Com objetivo de validar os códigos computacionais do presente trabalho, são feitas simulações com uma matriz esparsa gerada de forma aleatória de dimensão 200x200, posteriormente escolhido o vetor solução x com todas as componentes do vetor com valores iguais a 1, e finalmente calculado o vetor b. Segue o algoritmo gerador de matrizes com valores aleatórios e com alguns de seus elementos nulos. O valor da tolerância de erro da solução em todos os sistemas simulados é de 1.0×10^{-09}.

```
Criando a matriz de forma aleatória
n = 200
A = zeros(n, n)
    para  j = 1, 2, 3, ··· , n
        para  i = 1, 2, 3, ··· , n
        Aux = rand
            se  Aux > 0.2
            A(i, j) = 0
            senão
            A(i, j) = Aux
            fim
        fim
    fim
Montando a diagonal principal dominante
para  i = 1, 2, 3, ··· , n
    para  j = 1, 2, 3, ··· , n
        se  i == j
        A(i, j) = 10 + A(i, j)
        fim
    fim
fim
```

O algoritmo para gerar a matriz aleatória utiliza a função do Matlab®*rand*. A matriz é gerada no primero laço de repetição, onde a estrutura condicional atribui valores aleatórios e zeros. Na segunda estrutura com laços de repetição, a matriz recebe elementos com maior valor na diagonal principal, transformando-a numa matriz diagonalmente dominante, afim de garantir a convergência dos métodos iterativos e gerar a solução do sistema.

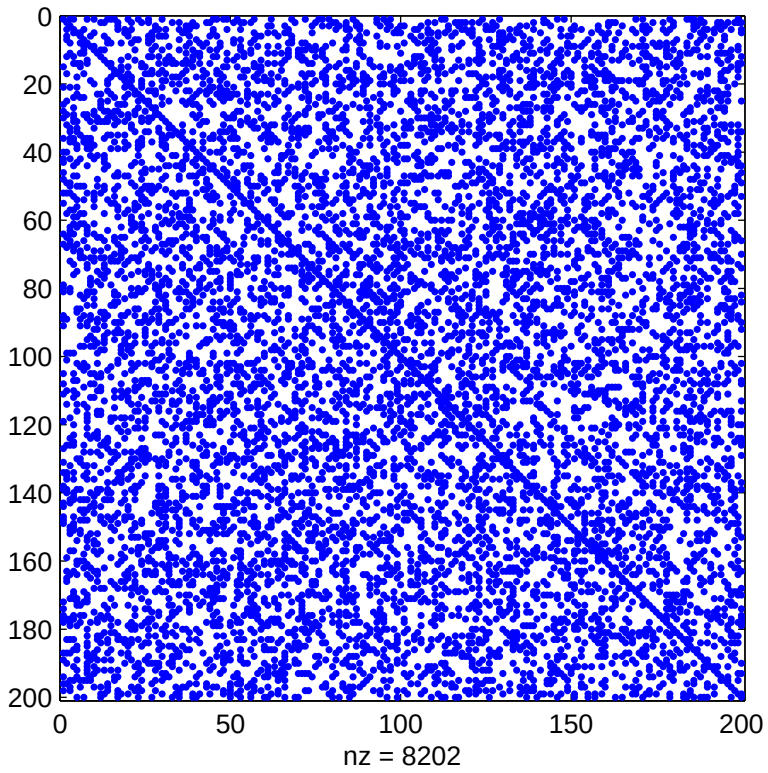
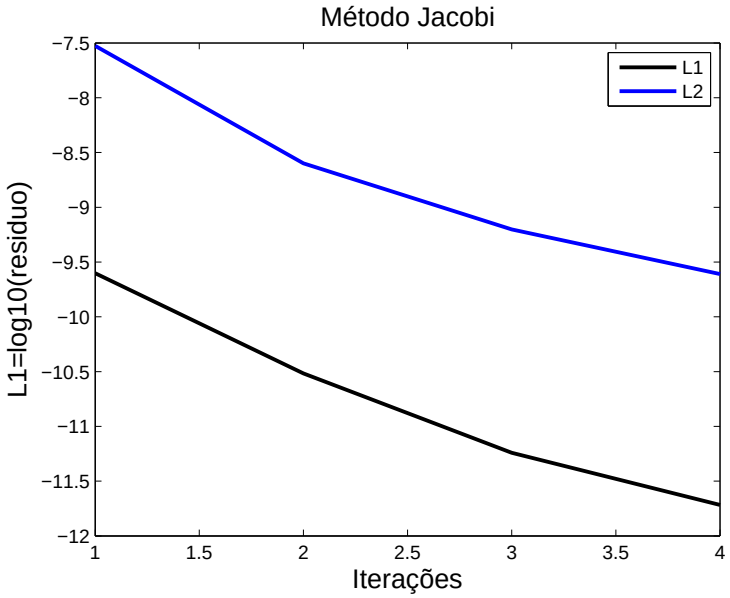


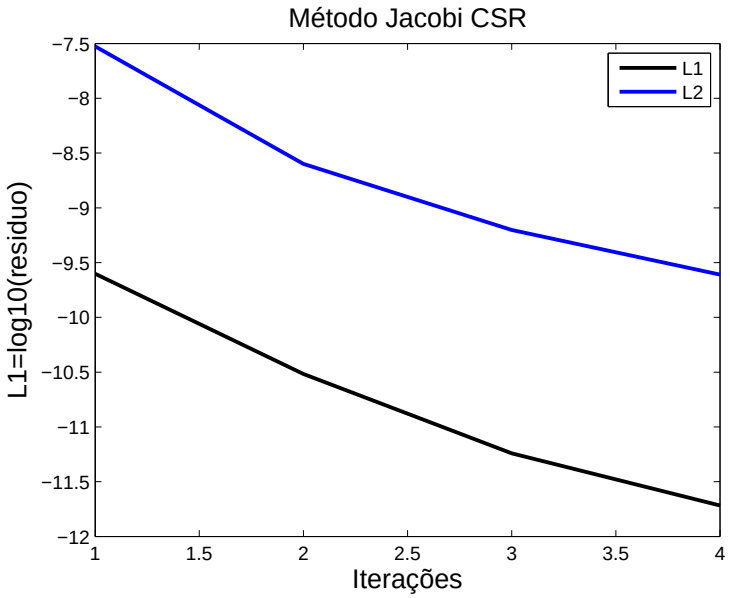
Figura 4.1: Diagrama de esparsidade matriz 200x200 gerada de forma aleatória e grau de esparsidade GE=79,49 % e $\gamma = 0.7009$

4.2 Validação da implementação dos códigos (gráficos de convergência)

Após obter o sistema linear com a matriz gerada de forma aleatória, a solução desse sistema é calculada pelos métodos iterativos, afim de comprovar o funcionamento de cada um, com a obtenção da mesma solução em todos os códigos. A seguir os gráficos de convergência de cada um e a solução encontrada dentro da margem de erro desejada, no caso 1×10^{-09}.



(a) Convergência Jacobi denso

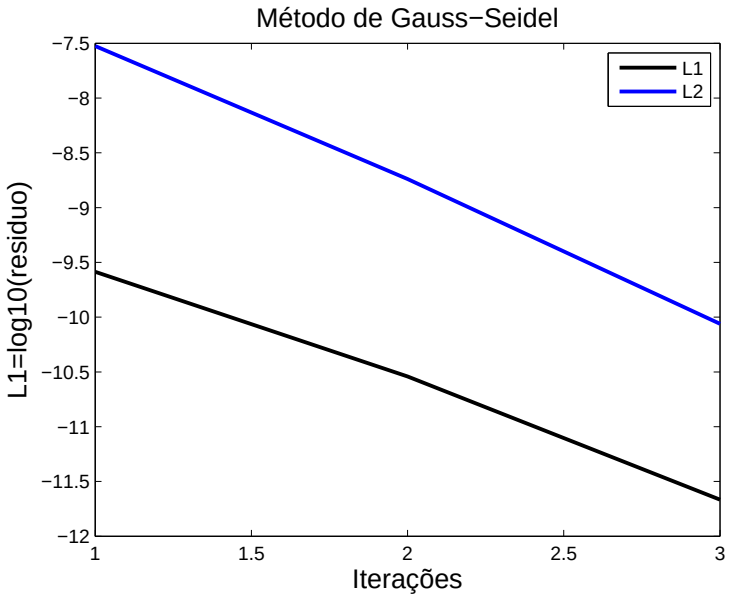


(b) Convergência Jacobi CSR

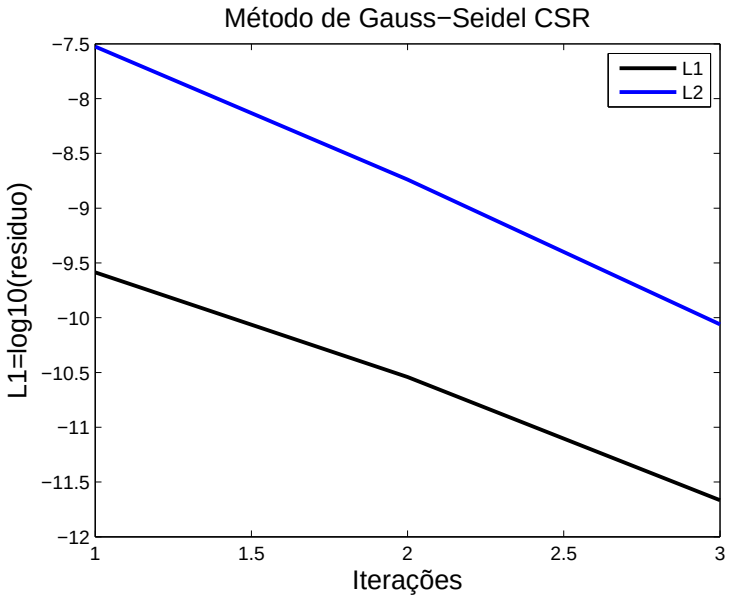
Figura 4.2: Gráficos de convergência método de Jacobi denso e com estrurura CSR com sistema linear formado pela matriz 200x200 gerada de forma aleatória

Observa-se nos gráficos de convergência da solução da matriz de dimensão 200x200 que a convergência dos método de Jacobi para matrizes densas e com a técnica de armazenamento de matrizes esparsas CSR são idênticos, com a obtenção das mesmas soluções numéricas, comprovando que os algoritmos são implementados sem comprometer o funcionamento do método numérico. Esse fato se repete também com o método de Jacobi

implementado com as outras técnicas de armazenamento CSC, CDS e Skyline. A seguir os gráficos de convergência do método de Gauss-Seidel para matrizes densas e com as técnica de armazenamento de matrizes esparsas CSR.



(a) Convergência Gauss-Seidel denso



(b) Convergência Gauss-Seidel CSR

Figura 4.3: Gráficos de convergência método de Gauss-Seidel denso e com estrurura CSR com sistema linear formado pela matriz 200x200 gerada de forma aleatória

O método de Gauss-Seidel implementado com as quatro estruturas de armazenamento de matrizes esparsas geram os mesmos gráficos de convergência e consequentemente a

mesma solução numérica. Os gráficos anteriores mostram a convergência do método de Gauss-Seidel para matrizes densas e com a técnica CSR, onde é observada a mesma convergéncia e quantidade de iterações, fato que se repete com as outras técnicas de armazenamento de matrizes esparsas CSC, CDs e Skyline.

4.3 Matrizes esparsas do tipo banda

Após estudos sobre os métodos numéricos para solução de sistemas de equações algébricas lineares, estruturas de armazenamento de matrizes esparsas e implementação dessas estruturas aos métodos numéricos iterativos, são realizadas simulações com sistemas gerados a partir de um código implementado em matlab de solução da equação de Laplace 2D, através do método de elementos finitos, de acordo com [7], na qual são gerados sistemas lineares de dimensões 100x100, 200x200, 400x400, 600x600, 800x800, e finalmente 1000x1000 com matrizes dos coeficientes pentadiagonais. Para comparação da complexidade dos algoritmos, foram tomados 5 amostras de tempo de cada sistema, com a realização dos cálculos de média e desvio padrão. O microcomputador utilizado para as simulações possui a seguinte configuração: Processador intel(R) Pentium(R) Dual CPU modelo T2390 1.86GHz com 1.93GB de memória RAM, 160GB de HD, sistema operacional Windows XP e ambiente Matlab ®versão 2008a. A tabela a seguir contém o número de elementos não-nulos, o grau de esparsidade e o valor de γ para cada matriz do tipo banda pentadiagonal:

Matrizes	nnz (elementos nao-nulos)	Grau de esparsidade	valor de γ
Matriz 100x100	344	96.56 %	0.2683
Matriz 200x200	780	98.05 %	0.2569
Matriz 400x400	1582	99.01 %	0.2295
Matriz 600x600	2824	99.21 %	0.2421
Matriz 800x800	3806	99.40 %	0.2333
Matriz1000x1000	4604	99.54 %	0.2210

Tabela 4.1: Tabela com grau de esparsidade e valores de γ

Na matriz pentadiagonal de dimensão 100x100, percebe-se a grande quantidade de elementos nulos, se a mesma fosse totalmente preenchida de elementos não-nulos, teria um total de 10000 elementos, mas nesse caso possui 344 elementos indicado por nnz. Na matriz de dimensão 200x200, observa-se 780 elementos diferentes de zero contra 40000 elementos se fosse totalmente preenchida com elementos não-nulos. A matriz 400x400

tem-se 1582 elementos diferentes de zero contra 160000 elementos se fosse densa. A de dimensão 600x600, nnz=2824, ou seja, 2824 elementos diferentes de zero contra 360000 elementos se fosse densa. A matriz 800x800 é formada por 3806 elementos diferentes de zero, e teria 640000 elementos se fosse totalmente preenchida com elementos não-nulos. E finalmente a matriz de dimensão 1000x1000, que possui nnz=4604 elementos diferentes de zero e 1000000 de elementos se fosse densa.

4.4 Condicionamento das matrizes

Após apresentação dos conceitos sobre condicionamento de matrizes no capítulo 3, segue os valores do condicionamento das matrizes pentadiagonais de dimensões100x100, 200x200, 400x400, 600x600, 800x800 e 1000x1000. para o cálculo do condicionamento das matrizes é utilizada a função do Matlab®*cond*. Importante ressaltar que quanto mais próximo de 1, melhor é o condionamento da matriz do sistema linear. Em contrapartida, os valores do condicionamento obtidos nas matrizes utilizadas no trabalho estão dentro dos valores típicos de matrizes oriundas do método de elementos finitos.

Matrizes	Condicionamento
100x100	32.1634
200x200	89.1262
400x400	325.3713
600x600	724.5654
800x800	1.2867e+03
1000x1000	2.0118e+03
200x200 aleatória	1.56

Tabela 4.2: Tabela condicionamento das matrizes

4.5 Alocação de memória

Através desses diagramas percebe-se que o número de elementos da matriz diminui de forma significativa se não considerados os elementos nulos. Esse fato reflete na quantidade de memória utilizada para o armazenamento desses elementos. De acordo com [18] e [19], cada elemento de uma matriz ocupa 4 bytes de memória. No caso do Matlab®, por ter dupla precisão, tem-se 8 bytes de memória por elemento. Se 1KB (kilobyte) tem 1024 bytes e 1MB (megabyte) tem 1024 KB, é possível calcular do uso de memória para armazenamento dos elementos com e sem as técnicas de armazenamanto de matrizes esparsas.

Assim, é ilustrada a seguir a quantidade de elementos das matrizes de dimensões 100x100, 200x200, 400x400, 600x600, 800x800, 1000x1000 na forma densa e esparsa, ressaltando que a quantidade de memória é calculada para as técnicas de armazenamento CSR e CSC, que não guardam elementos nulos, ao contrário das estruturas CDS e skyline. O cálculo da quantidade de memória é realizada da seguinte forma:

Para uma matriz de ordem n, com a estrutura CSR, o vetor IA possui tamanho $n+1$, já os vetores JA e AA possuem tamanho igual a quantidade de elementos diferentes de zero. Para estrutura CSC, os vetores IA e AA possuem dimensão igual a quantidade de elementos diferentes de zero da matriz e o vetor JA possui dimensão $n+1$. Por exemplo, para a matriz 100x100 utilizada nas simulações, tem-se:

vetor IA possui 101 elementos, o vetores AA e JA possuem 344 elementos cada um, totalizando 789 elementos. Se cada elemento ocupa 8 bytes de memória, vem:

789 elementos x 8 bytes = 6312 bytes, transformando para MB.

$$\frac{6312}{(1024)^2} = 0,0060 \quad MB$$

Matrizes	Número de elementos matriz na forma densa	Número de Elementos matriz sem elementos nulos
Matriz 100x100	10000	344
Matriz 200x200	40000	780
Matriz 400x400	160000	1582
Matriz 600x600	360000	2824
Matriz 800x800	640000	3806
Matriz1000x1000	1000000	4604

Tabela 4.3: Tabela com a quantidade de elementos das matrizes na forma densa e esparsa utilizadas no trabalho

Matrizes	Memória Matriz na forma Densa em MB	Memória Matriz sem elementos nulos em MB
100x100	0.0763	0.0060
200x200	0.3052	0.0134
400x400	1.2207	0.0272
600x600	2.7466	0.0477
800x800	4.8828	0.0642
1000x1000	7.6294	0.0779

Tabela 4.4: Tabela alocação de memória das matrizes em MB

É importante ressaltar que o cálculo do uso de memória de cada matriz é baseado nas estruturas CSR e CSC, que não armazenam elementos nulos, diferente das estruturas CDS e Skyline, que podem fazer o armazenamento desses elementos. A quantidade de memória utilizada para armazenar as matrizes diminui consideravelmente, levando em conta apenas os elementos diferentes de zero e os vetores que indicam a posição dos mesmos na matriz original. No caso da matriz 1000x1000, o uso da memória é 108 vezes menor. A seguir o gráfico de uso de memória para armazenamento dos elementos da matriz do sistema, com e sem o uso das técnicas de armazenamento esparsas.

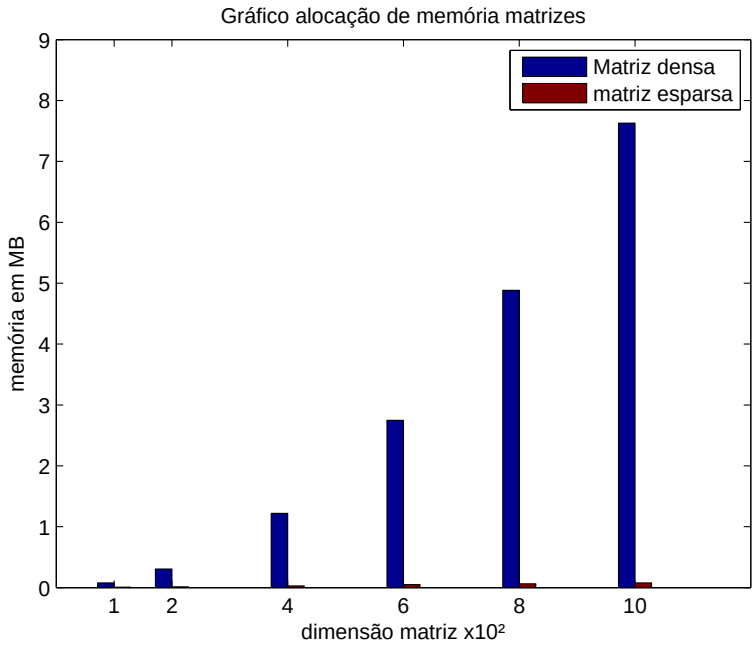


Figura 4.4: Gráficos do uso de memória para armazenamento de matrizes esparsas CSR, CSC

Após essa etapa, obtém-se os cálculos da média aritmética dos tempos de processa-

mento e desvio padrão utilizando o Matlab®com as funções *mean e std*, para os métodos Jacobi e Gauss-Seidel, na forma para matrizes densas e com as estruturas de armazenamento esparsas CSR, CSC, CDS e Skyline. As tabelas com as amostras de tempo de processamento em segundos de cada sistema utilizada para os cálculos de média e desvio padrão se encontram no Apêndice A do presente trabalho. A seguir as tabelas de média e desvio padrão.

Média de tempo	Jacobi Denso	Jacobi CSR	Jacobi CSC	Jacobi CDS	Jacobi Skyline
Matriz 100x100	0.1822	0.0405	0.0471	0.0511	0.0658
Matriz 200x200	1.7441	0.1279	0.1441	0.1112	0.5020
Matriz 400x400	28.3366	0.6690	0.7216	0.5419	6.3948
Matriz 600x600	140.5388	2.2150	2.3306	1.7608	29.6968
Matriz 800x800	458.3527	5.1572	5.5057	4.2701	88.3208
Matriz1000x1000	1.3857e+03	10.5562	11.0702	8.6509	207.8036

Tabela 4.5: Média de tempo método Jacobi (tempo em segundos)

Média de tempo	Seidel Denso	Seidel CSR	Seidel CSC	Seidel CDS	Seidel Skyline
Matriz 100x100	0.0997	0.0277	0.0373	0.0897	0.0448
Matriz 200x200	0.9614	0.0705	0.0981	0.5632	0.2752
Matriz 400x400	18.2474	0.3802	0.4397	6.1898	3.3736
Matriz 600x600	92.1530	1.1010	1.2782	26.3003	15.7409
Matriz 800x800	254.1326	2.4761	3.1391	73.0671	60.1471
Matriz1000x1000	621.7253	4.7334	6.1479	169.5121	135.4533

Tabela 4.6: Média de tempo método Gauss-Seidel (tempo em segundos)

Valores relativos ao desvio padrão no tempo de processamento dos métodos Jacobi e Gauss-Seidel.

Desvio padrão	Jacobi Denso	Jacobi CSR	Jacobi CSC	Jacobi CDS	Jacobi Skyline
Matriz 100x100	0.0035	0.0011	0.0052	0.0032	0.0040
Matriz 200x200	0.0319	0.0040	0.0223	0.0123	0.0164
Matriz 400x400	0.6620	0.0144	0.0319	0.0300	0.0816
Matriz 600x600	2.4879	0.0667	0.1173	0.1366	0.0661
Matriz 800x800	5.1365	0.1203	0.2411	0.2568	0.2163
Matriz1000x1000	37.5878	0.1223	0.0884	0.2350	0.5336

Tabela 4.7: Desvio padrão tempo método Jacobi (tempo em segundos)

Desvio padrão	Seidel Denso	Seidel CSR	Seidel CSC	Seidel CDS	Seidel Skyline
Matriz 100x100	0.0018	0.0007	0.0020	0.0061	0.0037
Matriz 200x200	0.4911	0.0062	0.0069	0.0282	0.0049
Matriz 400x400	0.7597	0.0203	0.0248	0.2209	0.0800
Matriz 600x600	8.2945	0.0293	0.0149	0.9515	0.1387
Matriz 800x800	3.8482	0.0566	0.1268	1.4542	0.2012
Matriz1000x1000	16.0969	0.0877	0.2121	2.8323	0.9612

Tabela 4.8: Desvio padrão tempo método Gauss-Seidel (tempo em segundos)

A tabela (4.5) com a média de processamento de cada algoritmo com o método de Jacobi, mostra a grande diferença no tempo de processamento do método na forma para matrizes densas e com utilização das técnicas de armazenamento de matrizes esparsas CSR, CSC, CDS e Skyline, onde o método de Jacobi com estrutura CDS obtém melhor eficiência atingindo o menor tempo de processamento. No método de Gauss-Seidel, conforme tebela (4.6), todas as técnicas de armazenamento de matrizes esparsas se mostram mais eficientes, gastando menos tempo de processamento do que o método de Gauss-Seidel para matrizes densas, onde a estrutura CSR obtém o menor tempo para gerar a solução do sistema linear. Outro ponto a ser observado é o desvio padrão, que aumenta a medida que o tempo de processamento é maior. Isso é justificado pelo modo que os tempos de processamento são obtidos, com o uso da função *tic toc* do Matlab®, que registra o tempo de outros processos que são realizados no computador.

4.6 Complexidade de um algoritmo

A complexidade de um algoritmo, de acordo com [20], consiste na quantidade de "trabalho"para sua execução, expressa em função das operações fundamentais, as quais variam de acordo com o algoritmo, e em função do volume de dados. A complexidade pode ser temporal eu espacial. A espacial representa o espaço de memória usado para executar um algoritmo e a temporal o tempo necessário para execução. Para o estudo da complexidade são usados três perspectivas: pior caso, médio caso e melhor caso. neste trabalho será utilizado o pior caso, representada por $O(n)$, que consiste em considerar o pior dos casos ocorrido num determinado processo de execução, por exemplo: Se existirem cinco baús, sendo que em apenas um existe algo dentro e deseja-se encontrar este baú, a complexidade pior caso será $O(5)$, pois no pior caso o baú cheio será encontrado na quinta tentativa. A seguir as tabelas com o polinômio de complexidade de cada algoritmo obtidos através da função do Matlab®*polifit* pelo processo de interpolação.

Métodos iterativos	Polinômio de complexidade
Jacobi para matrizes densas	$O(n) = 4.5401.10^{-06}n^3 - 4.4754.10^{-03}n^{02} + 1.4301n - 1.17457.10^{02}$
Jacobi com estrutura CSR	$O(n) = 1.3990.10^{-08}n^3 - 5.2011.10^{-06}n^2 + 1.8798.10^{-03}n - 1.2742.10^{-01}$
Jacobi com estrutura CSC	$O(n) = 1.3634.10^{-08}n^3 - 3.8729.10^{-06}n^2 + 1.3753.10^{-03}n - 7.3583.10^{-02}$
Jacobi com estrutura CDS	$O(n) = 1.0919.10^{-08}n^3 - 3.1976.10^{-06}n^2 + 9.5604.10^{-04}n - 2.9354.10^{-02}$
Jacobi com estrutura Skyline	$O(n) = 4.1753.10^{-07}n^3 - 2.7851.10^{-04}n^2 + 7.4044.10^{-02}n - 5.537$

Tabela 4.9: Polinômio de complexidade método de Jacobi para matrizes densas e com as estruturas de armazanamento de matrizes esparsas CSR, CSC, CDS e Skyline

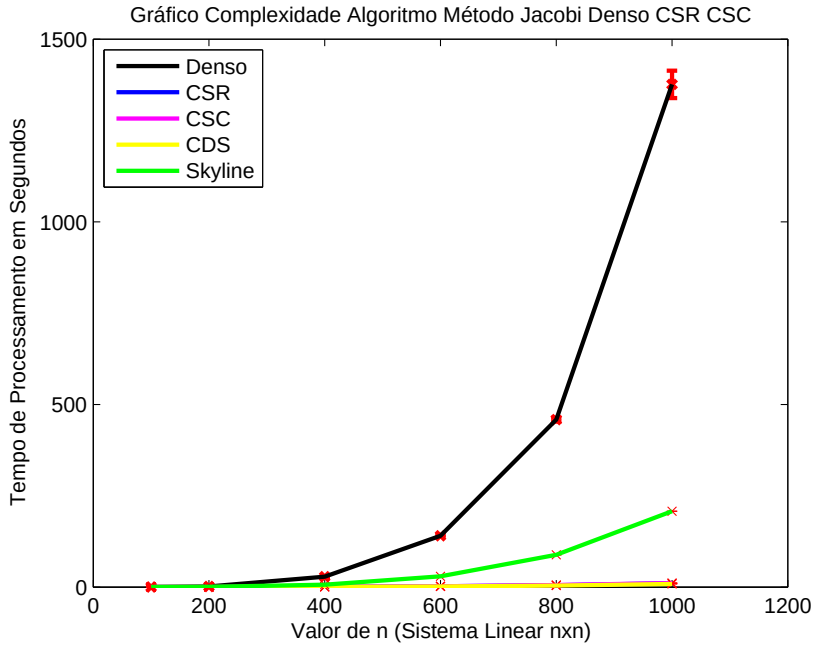


Figura 4.5: Gráficos de eficiência dos métodos Jacobi tradicional e com as estruturas de armazenamento esparsas CSR, CSC, CDS e Skyline

Para melhor visualização dos resultados do método com as estruturas CSR e CSC, tem-se o gráfico a seguir:

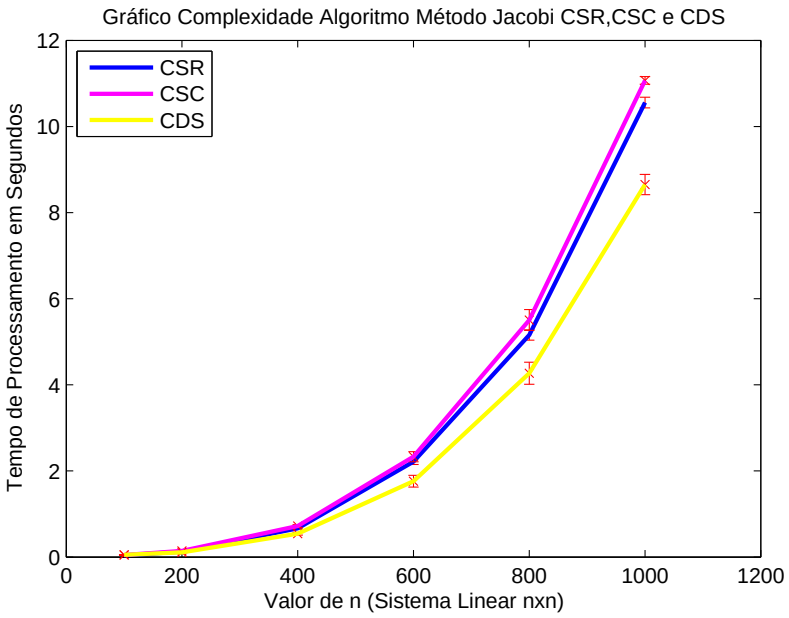


Figura 4.6: Gráficos de eficiência dos métodos Jacobi com as estruturas de armazenamento esparsas CSR, CSC e CDS

A partir da obtenção dos polinômios de complexidade de cada método é possível observar o valor dos coeficientes e chegar a conclusão que o método que consome maior tempo de processamento, possui maior valor no coeficiente de terceiro grau do polinômio de complexidade, conforme tabelas (4.9) e (4.10), pois a curvas de complexidade são mais acentuadas nesses casos. Em contrapartida, nos métodos em que os tempos de processamento são menores, as cuvas de complexidade são mais suaves e coeficiente de terceiro grau do polinômio possui menor valor. No método de Jacobi para matrizes densas, o valor coeficiente de n^3 é da ordem de 10^{-06}, gerando a curva mais acentuada entre os métodos de Jacobi, pois utiliza a estrutura com laços de repetição que não conseguem identificar elementos nulos, efetuando operações matemáticas com esses elementos desnecessariamente. Já o método de Jacobi com estrutura de armazenamento de matrizes esparsas CDS, obteve menor valor no coeficiente de n^3, na ordem de 10^{-08}, ressaltando que todas as outras estruturas de armazenamento de matrizes esparsas foram mais eficientes que o método para matrizes densas, consumindo menor tempo de processamento para realizar a mesma tarefa.

Métodos iterativos	Polinômio de complexidade
Gauss-Seidel para matrizes densas	$O(n) = 1.3559.10^{-06}n^{03} - 1.008.10^{-03}n^2 + 2.9584.10^{-01}n - 2.3928.10$
Gauss-Seidel com estrutura CSR	$O(n) = 4.6325.10^{-09}n^{03} - 1.8069.10^{-07}n^{02} + 2.8899.10^{-04}n - 8.4681.10^{-03}$
Gauss-Seidel com estrutura CSC	$O(n) = 7.0305.10^{-09}n^{03} - 1.3340.10^{-06}n^2 + 4.5631.10^{-04}n + 2.0081.10^{-03}$
Gauss-Seidel com estrutura CDS	$O(n) = 1.0919.10^{-8}n^3 - 3.1976.10^{-6}n^2 + 9.5604.10^{-4}n - 2.9354.10^{-2}$
Gauss-Seidel com estrutura Skyline	$O(n) = 2.4296.10^{-07}n^{03} - 1.2440.10^{-04}n^2 + 1.7759.10^{-02}n - 4.4486.10.10^{-01}$

Tabela 4.10: Polinômio de complexidade método de Gauss-Seidel para matrizes densas e com as estruturas de armazanamento de matrizes esparsas CSR, CSC, CDS e Skyline

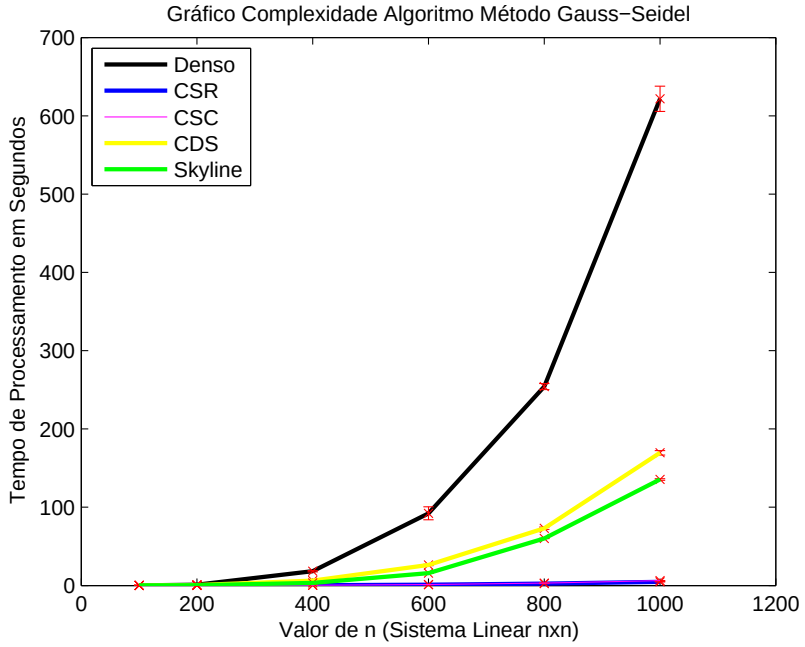


Figura 4.7: Gráficos de complexidade dos métodos Gauss-Seidel tradicional e com as estruturas de armazenamento esparsas CSR, CSC, CDS e skyline

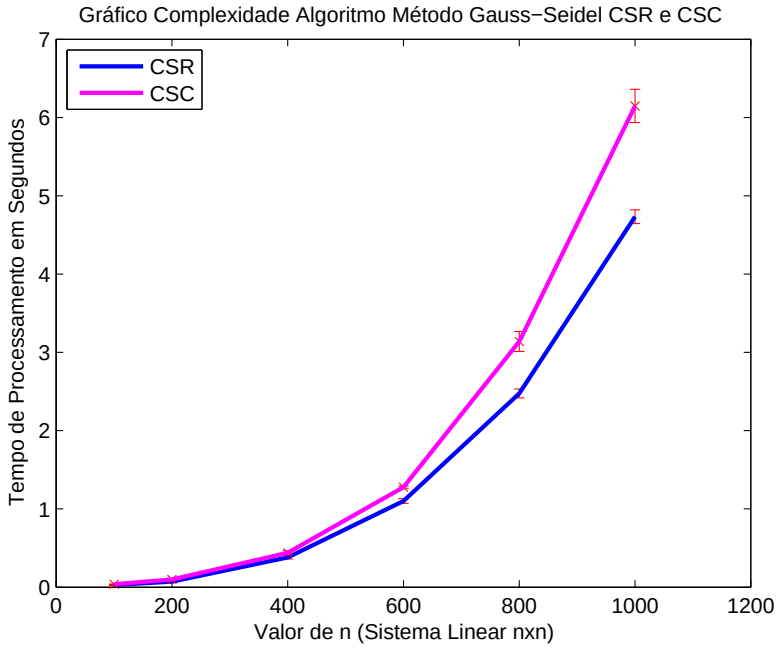


Figura 4.8: Gráficos de eficiência dos métodos Gauss-Seidel com as estruturas de armazenamento esparsas CSR e CSC

No método de Gauss-Seidel, o método para matrizes densas tem coeficiente de terceiro grau do polinômio de complexidade na ordem de 10^{-06}, bem maior que o método com estrutura de armazenamento de matrizes esparsas CSR, que obteve o menor valor no coeficiente de terceiro grau utilizando esse método iterativo, na ordem de 10^{-09}. Dessa forma, tem-se a conclusão que a complexidade de um algoritmo mostra o esforço computacional que cada algoritmo tem em realizar determinada tarefa. A seguir os gráficos com as curvas de complexidade de cada algoritmo, com as simulações feitas com as matrizes pentadiagonais de dimensões 100x100, 200x200, 400x400, 600x600, 800x800 e 1000x1000, onde é possível observar a inclinação da curva de complexidade de cada algoritmo.

4.7 Conclusões

Ao longo do presente trabalho foram apresentadas 4 técnicas computacionais de armazenamento de matrizes esparsas, criadas com o intuito de promover eficiência computacional em áreas da ciência que fazem uso de resolução de sistemas lineares de grande porte. É apresentado um pouco do desenvolvimento das estruturas de armazenamento, que de acordo com [6], os métodos iterativos ganham espaço com o desenvolvimento das estruturas de armazenamento esparsas e a evolução dos computadores atuais, que se tornaram mais confiáveis e eficientes. Através das simulações realizadas com os sistemas lineares esparsos, obtém-se tabelas e gráficos comparativos dos métodos Jacobi e Gauss-Seidel, com as estruturas de armazenamento CSR (Compressed Sparse Row), CSC (Compressed Sparse Column), CDS (compressed Diagonal Storage) e Skyline. Os códigos computacionais implementados com método de Jacobi atingem melhor eficiência com a estrutura CDS, 160 vezes mais rápida do que o método de jacobi para matrizes densas, dentre as quatro técnicas de armazenamento de matrizes esparsas estudadas, com simulações feitas nos sistemas lineares com dimensões 100x100, 200x200, 400x400, 600x600, 800x800 e 1000x1000. A segunda estrutura com melhor desempenho é a CSR, com o desempenho 131 vezes mais rápida que o método de jacobi para matrizes densas, seguida pela CSC, 125 vezes, a Skyline quase 7 vezes mais eficiente, e por último, o método de Jacobi, sem a utilização das estruturas esparsas. Observa-se do ponto de vista matemático, que todos os algoritmos fornecem a mesma solução para o problema, fato observado nos gráficos de convergência apresentados anteriormente. No método de Gauss-Seidel, a estrutura esparsa mais eficiente com menor tempo de processamento é a CSR, 131 vezes mais eficiente que o método de Gauss-Seidel para matrizes densas, seguido pelo método de Gauss-Seidel com estrutura CSC, 101 vezes mais eficiente, a CDS, quase 72 vezes, a Skyline, 4 vezes mais rápido e por último, o método de Gauss-seidel sem técnicas de armazenamento de matrizes esparsas. É constatada a eficiência no tempo de processamento nos dois métodos iterativos, com todas as estruturas de armazenamento de matrizes esparsas, o que mostra a importância dessas técnicas de armazenamento, quando se trabalha com matrizes de grande porte e muitos elementos nulos. No aspecto computacional, relativo a quantidade de memória utilizada, é fácil perceber a economia de memória gerada pelas estruturas de armazenamento, já que elementos nulos não são considerados quando utilizada as estruturas CSR e CSC, que armazenam somente elementos diferentes de zero, onde obtém-se 98 vezes menos quantidade de memória para alocação dos elementos da matriz 1000x1000. As estruturas CDS e skyline acabam guardando alguns elementos nulos, dependendo da distrubuição dos elementos na matriz, mas também conseguem diminuir consideravelmente

a quantidade de elementos a ser armazenado.

4.8 Trabalhos Futuros

Com o intuito de prosseguir os estudos sobre métodos iterativos com uso das técnicas de armazenamento de matrizes esparsas, algumas sugestões de trabalhos futuros são citadas:

1. Estudo e implementação de outras estruturas de armazenamento de matrizes esparsas, como por exemplo, COO (Coordinate Formats), JDS (Jagged Diagonal Storage) entre outras, aos métodos iterativos.

2. Estudo e implementação de outros métodos iterativos de resolução de sistemas de equações algébricas lineares com a inclusão das técnicas de armazenamento de matrizes esparsas.

3. Estudo dos métodos iterativos de resolução de sistemas de equações algébricas não-lineares com estrutura de armazenamento de matrizes esparsas.

4. Otimização dos códigos computacionais apresentados nesse trabalho, visto que os mesmos foram implementados de forma serial, podendo ser usada vetorização e outras técnicas para aumentar ainda mais a eficiência computacional.

Referências

[1] Rodney Josué Biezuner. Notas de aula algebra linear numérica. Acessado em: 15 janeiro de 2014,

http://arquivoescolar.org/bitstream/arquivo-e/150/1/alg_lin_num.pdf

.

[2] B. P. Demidovich and I. A. Maron. *Computational mathematics:Theory, Algorithms, and Applications*. Mir Publishers, 3 edition, 1981.

[3] Humberto Lima Soriano Silvio De Souza Lima. *Método de Elementos Finitos em Análise de Estruturas*. 1 edition, 2003.

[4] Randall J. Leveque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, 1 edition, 1955.

[5] FRS O.C. Zienkiewicz, CBE. *The Finite Element Method: Its Basis and Fundamentals*. British Library Cataloguing in Publication Data, 6 edition, 2005.

[6] Yousef Saad. *Iterative Methods for Sparse of Linear Systems*. Society for Industrial and Applied Mathematics, 2 edition, 2003.

[7] Diomar Cesar Lobão. Notas de aula estrutura de dados e algoritmos.

[8] Miroslav Tuma. Direct methods for sparse matrices. Acessado em: 18 jan. 2014

http://www2.cs.cas.cz/~tuma/ps/direct.pdf

.

[9] Lúcia Catabriga. Amazenamento de matrizes esparsas. Acessado em: 16 julho 2013,

http://www.inf.ufes.br/~luciac/cn/aula_armazenamento.pdf

.

[10] World Community Grid. Cds sparse structure, 1995. 4 ago. 2013,

http://netlib.org/utk/papers/templates/node94.html#SECTION00931400000000000000

.

[11] Nathan Bell. Sparse matrix representations and iterative solvers, 2011. Acessado em: 11 fev. 2014,

http://www.bu.edu/pasi/files/2011/01/NathanBell1-10-1000.pdf

.

[12] . Sparse matrix storage formats. Acessado em: 16 set. 2013,

http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/GUID-9F

.

[13] Antonios - Kornilios Kourtis. Data compression techniques for performance improvement of memory-intensive applications on shared memory architectures. Acessado em: 22 set. 2013,

http://people.inf.ethz.ch/akourtis/phd/phd-en.pdf

.

[14] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. Siam, 1994.

[15] . System matrix assembling and the skyline storage scheme. Acessado em: 11 nov. 2013,

http://www.dgp.toronto.edu/~tu/thesis/node57.html

.

[16] . Data compression techniques for performance improvement of memory-intensive applications on shared memory architectures. Acessado em: 08 dez. 2013,

http://www.cs.indiana.edu/classes/p573/notes/sparsemat/sparsemat.html

.

[17] David S Watkins. *Fundamentals of matrix computations*, volume 64. John Wiley & Sons, 2004.

[18] Mathworks. Strategies for efficient use of memory. Acessado em: 11 Dez 2013,

http://www.mathworks.com/help/matlab/matlab_prog/strategies-for-efficient-use

.

[19] Marcus Vinicius Chaffin Costa. Matlab básico com aplicações em engenharia. Acessado em: 26 Fevereiro 2014,

`http://fga.unb.br/marcus.chaffim`

.

[20] Gonçalo Madeira. Complexidade computacional. Acessado em: 13 junho de 2014,

`http://w3.ualg.pt/~hshah/algoritmos/aula8/Aula8.htm`

.

Capítulo 5

Apêndice A

5.1 Tabelas com amostras de tempo das matrizes esparsas do tipo banda

As simulações computacionais são realizadas com os sistemas lineares gerados pelas matrizes pentadiagonais mostradas anteriormente, utilizando a função *tic-toc* do Matlab®, que acaba contabilizando o tempo do sistema operacional. Por esse motivo, são tomadas 5 amostras de tempo de cada sistema. A seguir as tabelas com as amostras de tempo de processamento dos métodos Jacobi e Gauss Seidel, na forma para matrizes densas e com a inclusão das estruturas de armazenamento para matrizes esparsas CSR, CSC, CDS e Skyline, com os tempos de processamento medido em segundos.

Tempos Jacobi Denso	Amostra 1	Amostra 2	Amostra 3	Amostra 4	Amostra 5
Matriz 100x100	0.1780	0.1861	0.1830	0.1792	0.1845
Matriz 200x200	1.6996	1.7745	1.7501	1.7245	1.7716
Matriz 400x400	27.7202	27.6681	28.5672	28.4654	29.2619
Matriz 600x600	138.7623	137.6854	142.1380	140.2779	143.8304
Matriz 800x800	454.4909	463.5940	451.6279	459.6758	462.3748
Matriz1000x1000	1354.7170	1337.6548	1430.0062	1359.1450	1398.9637

Tabela 5.1: Amostras de tempo método Jacobi para matrizes densas (tempo em segundos)

A tabela contém cinco amostras de tempo de processamento de cada sistema formado pelas seis matrizes pentadiagonais apresentadas anteriormente.

Tempos Jacobi CSR	Amostra 1	Amostra 2	Amostra 3	Amostra 4	Amostra 5
Matriz 100x100	0.0399	0.0396	0.0417	0.0396	0.0415
Matriz 200x200	0.1256	0.1262	0.1274	0.1255	0.1350
Matriz 400x400	0.6887	0.6532	0.6532	0.6783	0.6597
Matriz 600x600	2.2136	2.2337	2.2059	2.3041	2.1178
Matriz 800x800	5.2360	5.2671	5.1636	5.1607	4.9584
Matriz1000x1000	10.6817	10.5780	10.6290	10.5302	10.3621

Tabela 5.2: Amostras de tempo método Jacobi com estrutura CSR (tempo em segundos)

Tempos de processamento	Amostra 1	Amostra 2	Amostra 3	Amostra 4	Amostra 5
Matriz 100x100	0.0470	0.0553	0.0407	0.0461	0.0466
Matriz 200x200	0.1791	0.1281	0.1273	0.1536	0.1322
Matriz 400x400	0.7590	0.7310	0.6963	0.7403	0.6816
Matriz 600x600	2.2820	2.2823	2.3678	2.2068	2.5141
Matriz 800x800	5.3072	5.9174	5.4091	5.3862	5.5087
Matriz1000x1000	11.1362	11.0489	11.0755	10.9330	11.1574

Tabela 5.3: Amostras de tempo método Jacobi com estrutura CSC (tempo em segundos)

Tempos Jacobi CDS	Amostra 1	Amostra 2	Amostra 3	Amostra 4	Amostra 5
Matriz 100x100	0.0555	0.0490	0.0509	0.0473	0.0527
Matriz 200x200	0.1088	0.1224	0.1212	0.0919	0.1118
Matriz 400x400	0.5518	0.5412	0.5535	0.4919	0.5713
Matriz 600x600	1.9303	1.6046	1.6858	1.8753	1.7081
Matriz 800x800	4.3827	3.9378	4.0550	4.4825	4.4923
Matriz1000x1000	8.9941	8.4564	8.7210	8.6764	8.4064

Tabela 5.4: Amostras de tempo método Jacobi com estrutura CDS (tempo em segundos)

Tempos Jacobi Skyline	Amostra 1	Amostra 2	Amostra 3	Amostra 4	Amostra 5
Matriz 100x100	0.0609	0.0686	0.0674	0.0699	0.0623
Matriz 200x200	0.4983	0.4998	0.5231	0.4786	0.5103
Matriz 400x400	6.3450	6.3992	6.5280	6.3862	6.3154
Matriz 600x600	29.6892	29.779	29.7320	29.6830	29.6008
Matriz 800x800	88.5260	88.3024	88.5406	88.2007	88.0345
Matriz1000x1000	207.9963	207.0243	208.1320	207.5123	208.3531

Tabela 5.5: Amostras de tempo método Jacobi com estrutura Skyline (tempo em segundos)

Tabelas tempo de processamento do método de Gauss-Seidel denso, com estrutura CSR, CSC, CDS e Skyline.

Tempos de processamento	Amostra 1	Amostra 2	Amostra 3	Amostra 4	Amostra 5
Matriz 100x100	0.0990	0.0994	0.1023	0.1005	0.0974
Matriz 200x200	1.3928	1.1238	1.0264	11669	1.1472
Matriz 400x400	17.5649	19.2866	18.3562	17.4479	18.5814
Matriz 600x600	104.4861	92.6149	88.4546	81.7813	93.4281
Matriz 800x800	251.4922	257.5565	258.9873	252.0473	250.5798
Matriz1000x1000	627.0884	601.3996	633.1385	608.3677	638.6325

Tabela 5.6: Amostras de tempo método Gauss-Seidel (tempo em segundos)

Tempos de processamento	Amostra 1	Amostra 2	Amostra 3	Amostra 4	Amostra 5
Matriz 100x100	0.0284	0.0280	0.0271	0.0266	0.0282
Matriz 200x200	0.0781	0.0699	0.0728	0.0609	0.0706
Matriz 400x400	0.3856	0.3489	0.3736	0.3909	0.4021
Matriz 600x600	1.1065	1.0861	1.0583	1.1280	1.1262
Matriz 800x800	2.5368	2.4697	2.5306	2.4116	2.4318
Matriz1000x1000	4.7665	4.7917	4.6605	4.8260	4.6222

Tabela 5.7: Amostras de tempo método Gauss-Seidel com estrutura CSR (tempo em segundos)

Tempos de processamento	Amostra 1	Amostra 2	Amostra 3	Amostra 4	Amostra 5
Matriz 100x100	0.0401	0.0387	0.0360	0.0356	0.0361
Matriz 200x200	0.1039	0.1054	0.0900	0.0991	0.0920
Matriz 400x400	0.4635	0.4429	0.4169	0.4114	0.4636
Matriz 600x600	1.2577	1.2749	1.2969	1.2737	1.2877
Matriz 800x800	3.0953	3.3505	3.0148	3.0917	3.1431
Matriz1000x1000	5.9216	6.1088	6.4799	6.0285	6.2008

Tabela 5.8: Amostras de tempo método Gauss-Seidel com estrutura CSC (tempo em segundos)

Tempos de processamento	Amostra 1	Amostra 2	Amostra 3	Amostra 4	Amostra 5
Matriz 100x100	0.0849	0.0972	0.0865	0.0954	0.0844
Matriz 200x200	0.6029	0.5353	0.5626	0.5774	0.5380
Matriz 400x400	6.0166	6.1725	6.5204	5.9684	6.2713
Matriz 600x600	26.5775	25.5303	26.2016	27.7722	25.4199
Matriz 800x800	72.1127	75.2508	73.3692	71.4309	73.1718
Matriz1000x1000	166.6271	171.6134	169.2360	173.1008	166.9833

Tabela 5.9: Amostras de tempo método Gauss-Seidel com estrutura CDS (tempo em segundos)

Tempos de processamento	Amostra 1	Amostra 2	Amostra 3	Amostra 4	Amostra 5
Matriz 100x100	0.0425	0.0440	0.0431	0.0429	0.0513
Matriz 200x200	0.2693	0.2795	0.2756	0.2803	0.2712
Matriz 400x400	3.3022	3.4882	3.3133	3.3398	3.4243
Matriz 600x600	15.5813	15.8304	15.9100	15.6208	15.7618
Matriz 800x800	60.1127	60.2508	60.3692	59.8309	60.1718
Matriz1000x1000	136.4197	134.1002	134.8287	136.0806	135.8371

Tabela 5.10: Amostras de tempo método Gauss-Seidel com estrutura Skyline (tempo em segundos