UFPA

UNIVERSIDADE FEDERAL DO PARA

ESTUDO E APLICAÇÃO DE TÉCNICAS DE ESPARSIDADE NA PROGRAMAÇÃO ORIENTADA A OBJETO E VIABILIDADE PRÁTICA DE SUA UTILIZAÇÃO EM PROGRAMA PARA ANÁLISE DE FLUXO DE CARGA

Andrey da Costa Lopes

Wellington Alex dos Santos Fonseca

2.º semestre / 2004

CENTRO TECNOLOGICO

UNIVERSIDADE FEDERAL DO PARÁ

CAMPUS UNIVERSITÁRIO DO GUAMÁ

BELÉM PARÁ

UNIVERSIDADE FEDERAL DO PARÁ
CENTRO TECNOLÓGICO
CURSO DE ENGENHARIA ELÉTRICA

ANDREY DA COSTA LOPES
WELLINGTON ALEX DOS SANTOS FONSECA

ESTUDO E APLICAÇÃO DE TÉCNICAS DE ESPARSIDADE NA PROGRAMAÇÃO ORIENTADA A OBJETO E VIABILIDADE PRÁTICA DE SUA UTILIZAÇÃO EM PROGRAMA PARA ANÁLISE DE FLUXO DE CARGA

TRABALHO SUBMETIDO AO COLEGIADO DO CURSO DE ENGENHARIA ELÉTRICA PARA OBTENÇÃO DO GRAU DE ENGENHEIRO ELETRICISTA OPÇÃO ELETROTÉCNICA

Belém
2005

Estudo e Aplicação de Técnicas de Esparsidade na Programação Orientada a Objeto e Viabilidade Prática de sua Utilização em Programa para Análise de Fluxo de Carga

Este Trabalho foi julgado em _____/_____ adequado para obtenção do Grau de Engenheiro Eletricista – Opção Eletrotécnica, e aprovado na sua forma final pela banca examinadora que atribuiu o conceito _____.

_____

Prof. Dr. Ghendy Cardoso Junior

ORIENTADOR

_____

Eng.º Msc. Edgar Modesto Amazonas Filho

CO-ORIENTADOR

_____

Prof. Dr. Marcus Vinicius Alves Nunes

MEMBRO DA BANCA EXAMINADORA

_____

Eng.º Msc. José Adolfo da Silva Sena

MEMBRO DA BANCA EXAMINADORA

_____

Prof. Dr. Orlando Fonseca Silva

COORDENADOR DO CURSO DE ENGENHARIA ELÉTRICA

Belém – PA

2005

“Fé inabalável só o é a que pode encarar frente a frente a razão, em todas as épocas da humanidade”.

Allan Kardec.

DEDICATÓRIA

Dedico este trabalho primeiramente a Deus por ter iluminado meu caminho durante toda essa jornada do conhecimento.

Aos meus pais, por todo carinho, compreensão e educação que me foram dados, valores estes que contribuíram positivamente para a formação do meu caráter. E ainda pela oportunidade e estrutura que me foi concedida para que eu pudesse desenvolver os meus estudos.

À minha namorada, Adrianne Ribeiro, pela companhia, onde teve participação marcante em minha vida, me incentivando e me apoiando nesses últimos anos de graduação.

Andrey da Costa Lopes

A Deus por ser capaz de olhar a todos os homens sem, contudo, perder de vista os méritos de cada um.

Ao meu pai, João Dutra da Fonseca, pelo amor e orientação dados a mim. Seu incentivo me proporcionou dedicação, perseverança e a certeza de que meus objetivos seriam alcançados.

À minha mãe, Ana Maria dos Santos Fonseca, pelo amor e compreensão que me foram dados. Seu amor e carinho nunca deixaram qualquer moléstia ou enfermidade prevalecer em minha vida.

Ao meu filho, Wallef Douglas da Silva Fonseca, que tanto amo. Sua integração a minha vida me trouxe muita alegria e motivação.

Wellington Alex dos Santos Fonseca

AGRADECIMENTOS

A todos os Professores que contribuíram para nossa formação acadêmica e ao Grupo do GSEI pela oportunidade do tema deste trabalho.

Ao Profº Dr. Eng. Ghendy Cardoso Junior que aceitou ser nosso orientador e que não hesitou em transmitir seus conhecimentos.

Ao Engenheiro Msc. Edgar Amazonas que contribui com o programa de Fluxo de Carga sem tratamento de esparsidade, suas orientações no uso do C++ Builder e no foco dos objetivos deste trabalho.

Ao Engenheiro Msc. Adolfo Sena por ter contribuído com a classe de matrizes sem esparsidade, o que permitiu a comparação dos resultados deste trabalho quanto ao teste de desempenho e validação das rotinas desenvolvidas.

À discente do curso de Engenharia de Computação Fabíola Graziela Noronha pela elaboração gráfica dos Fluxogramas.

A todos os nossos amigos e colegas, da Universidade e do GSEI, que contribuíram direta ou indiretamente para o êxito deste trabalho.

Andrey da Costa Lopes

Wellington Alex dos Santos Fonseca

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

As operações matriciais com matrizes esparsas de dimensões elevadas em algumas aplicações, como no problema de Fluxo de Carga, torna-se ineficiente e freqüentemente impossível alocar espaço de memória para todos os seus elementos. Caso seja alocado espaço suficiente para tal armazenamento em memória computacional, poderia ser inadequado, em tempo de máquina, executar todos os laços em cima da matriz a procura de elementos não-nulos. Obviamente algum tipo de esquema de armazenamento é requerido, um que armazene só elementos não-nulos da matriz, justamente com informações auxiliares suficientes para determinar a posição de um elemento e sua utilização em operações matriciais. Este trabalho apresenta o esquema de armazenamento Representação Completa e Ordenada por Comprimento de Linha (RCOCL), juntamente com os algoritmos das principais operações matriciais e suas respectivas implementações em Programação Orientada a Objeto, sendo bastante satisfatório os resultados obtidos em termos de desempenho quando aplicado ao Fluxo de Carga. A escolha do Fluxo de Carga para testar a eficiência dos algoritmos implementados deve-se ao elevado grau de esparsidade das matrizes envolvidas.

CAPÍTULO 1 – INTRODUÇÃO

1.1 – VISÃO GERAL DO ASSUNTO.

Ao se trabalhar com grandes sistemas elétricos de potência, o que se tem é uma quantidade elevada de variáveis que definem seus aspectos como um todo. O problema passa a ser, então, a procura de esquemas numéricos que garantam a determinação prática e precisa destas variáveis.

No planejamento e na operação de sistemas de energia elétrica é muito comum a utilização de matrizes para representar a rede elétrica em programas de simulação, cabendo destacar: o Fluxo de Potência, sistema de equações não-lineares; o curto-circuito, sistema de equações lineares; o Estimador de Estados, sistema de equações não-lineares. Em geral as matrizes utilizadas na análise de Sistemas Elétricos de Potência possuem um número de elementos nulos muito maior que o número de elementos não-nulos, como por exemplo, a matriz Jacobiana e a matriz de Admitância nodal da rede elétrica.

Define-se matriz esparsa como aquela para a qual é vantajosa a utilização do fato de que muitos de seus elementos são iguais a zero para fins de economia de memória e cálculos. Esta definição é geral e envolve dois aspectos básicos que são espaço de memória e volume de cálculos.

Considerando a definição de matrizes esparsas, pode-se estabelecer os princípios básicos das chamadas técnicas de esparsidade: minimizar a quantidade de dados armazenados; minimizar o número de operações realizadas; preservar a esparsidade.

Assim, técnicas de esparsidade são de fundamental importância para o desenvolvimento de programas computacionais eficientes, visto que, em aplicações que envolvem a modelagem da rede elétrica a percentagem de elementos não-nulos é muito pequena e a quantidade de aritmética envolvida é muito menor se comparada com técnicas não esparsas.

Com o advento do computador, com sua rapidez e operações seqüenciais, vários métodos e algoritmos foram propostos, procurando atender a essas necessidades, principalmente na resolução de sistemas algébricos lineares, esparsos e de grande porte, que surgem na solução numérica dos problemas matemáticos na área de sistemas de energia elétrica (MOROZOWSKI, 1981).

A Programação Orientada a Objeto tem sido aceita na engenharia de sistemas de Potência como uma alternativa viável à tradicional programação procedural. Recentemente, as bibliotecas orientadas a objeto, desenvolvidas para matrizes esparsas, têm recebido atenção significativa de pesquisadores na área da engenharia e ciência da computação

(PANDIT *et al*., 2001). Diante disso, é proposto, no presente trabalho, a descrição e implementação de uma classe de matrizes esparsas em C++.

O presente estudo, além dos aspectos teóricos citados anteriormente, e dos algoritmos implementados utilizando os recursos de Programação Orientada a Objeto, objetiva apresentar os resultados de testes realizados com as rotinas desenvolvidas.

1.2 – JUSTIFICATIVA DO TRABALHO

O presente trabalho tem por objetivo aplicar os conceitos, métodos e procedimentos desenvolvidos sob o título geral de “técnica de esparsidade” na Programação Orientada a Objeto, assim como sua aplicação num estudo de caso específico na área de Sistema de Potência, especificamente programação para análise de Fluxo de Carga.

Essas técnicas buscam explorar as características próprias dos sistemas de equações algébricas lineares que traduzem, matematicamente, o comportamento de redes elétricas e, por analogia, de quaisquer redes de estruturas que possam assim ser representadas.

Desta forma, consegue-se ampliar, substancialmente, o potencial de computadores digitais de qualquer porte, através da redução dos requisitos de memória e tempo de processamento associados à solução de sistemas lineares de grande dimensão.

Espera-se assim, com a publicação deste trabalho, tornar acessível as idéias que viabilizam a solução digital de redes com uma grande quantidade de barras, traduzidas em milhares de equações lineares ou não lineares simultâneas, em computadores de pequeno porte.

1.3 – ESTRUTURA DO TRABALHO

No **capítulo 2** é feita uma abordagem geral sobre Técnicas de Esparsidade, envolvendo o conceito de grau de esparsidade, a necessidade e a definição de alguns dos principais esquemas de armazenamento compacto para matrizes esparsas, abordando também os problemas de *fill-ins*, na solução de sistemas lineares, e a solução de tal problema utilizando esquemas de ordenação.

No **capítulo 3** é feita uma abordagem teórica sobre a Programação Orientada a Objeto, utilizando a linguagem C++. É válido ressaltar que o aprofundamento maior neste assunto está acima do escopo deste trabalho, pois não faz parte do objetivo maior, que é aplicar os conceitos de Programação Orientada a Objeto no desenvolvimento das rotinas de Esparsidade.

No **capítulo 4** é mostrado o esquema de armazenamento Representação Completa e Ordenada por Comprimento de Linha (RCOCL), juntamente com os algoritmos das principais operações matriciais e suas respectivas implementações em Programação Orientada a Objeto e os resultados obtidos em termos de desempenho quanto ao tempo de processamento dos mesmos.

No **capítulo 5** é feito um estudo de caso envolvendo as técnicas de esparsidade quando aplicado ao problema de Fluxo de Carga. Neste capítulo são descritas algumas características do programa computacional desenvolvido para a análise de Fluxo de Carga, além de uma abordagem teórica do método de Newton-Raphson, método adotado para o desenvolvimento do programa. Por último serão apresentados gráficos mostrando o desempenho do software de Fluxo de Carga com e sem esparsidade.

Finalmente, **no capítulo 6** serão apresentadas as conclusões finais do trabalho.

CAPÍTULO 2 – REVISÃO DAS TÉCNICAS DE ESPARSIDADE

2.1 – INTRODUÇÃO

O maior componente computacional nos cálculos de redes elétricas é a solução de equações matriciais, cujas matrizes são geralmente esparsas (possuem grande quantidade de elementos nulos). Assim, técnicas de esparsidade têm se mostrado favorável ao desenvolvimento de métodos de solução.

O princípio básico das técnicas de esparsidade resume-se em métodos que visam preservar a esparsidade, minimizar a quantidade de dados armazenados, assim como o número de operações realizadas sobre estruturas como matrizes esparsas e/ou vetores esparsos.

Existem várias técnicas de programação para o tratamento da esparsidade, dependendo da simetria da matriz, percentagem de esparsidade, blocos de zeros, etc.

Neste capítulo será abordada uma breve descrição das técnicas de esparsidade, mostrando o conceito de grau de esparsidade, apresentando alguns esquemas de armazenamento compacto de matrizes esparsas, assim como as técnicas de ordenação, utilizadas na solução de um sistema linear esparso.

2.2 – DEFINIÇÃO DE GRAU DE ESPARSIDADE

O grau de esparsidade de uma matriz é definido como a porcentagem de elementos nulos dessa matriz (MONTICELLI, 1983). Em particular, a matriz admitância nodal de uma sistema de NB barras e NR ramos, com referência no nó-terra, tem um grau de esparsidade dado pela Equação (2.1)

$$\mathrm{GE} = \frac{\mathrm{NB}^2 - (\mathrm{NB} + 2 \cdot \mathrm{NR})}{\mathrm{NB}^2} \cdot 100\% \qquad (2.1)$$

Para um sistema com 100 barras ($\mathrm{NB} = 100$) e 200 ramos ($\mathrm{NR} = 200$), o grau de esparsidade é de 95%. Para um sistema com $\mathrm{NB} = 1000$ e $\mathrm{NR} = 2000$, o grau de esparsidade é de $99{,}5\%$. Nos exemplos precedentes, considerou-se que uma barra tem em média quatro ramos ligados a ela ($\mathrm{NR} = 2 \cdot \mathrm{NB}$). Em sistemas reais, entretanto, este número em geral é menor que quatro ($\mathrm{NR} < 2 \cdot \mathrm{NR}$), significando que os graus de

esparsidade são ainda maiores que os estimados anteriormente. Outra observação importante é que o grau de esparsidade cresce com as dimensões da rede. Isto se deve ao fato de o número médio de ramos ligados a uma barra ser praticamente independente das dimensões da rede (na expressão acima para uma relação $\frac{NR}{NB}$ constante, GE é uma função crescente de NB.

2.3 – ESQUEMAS DE ARMAZENAMENTO COMPACTO DE MATRIZES ESPARSAS

O estudo de vários problemas relacionados com redes elétricas de potência passa pela resolução de um sistema de equações algébricas lineares do tipo:

$$\mathbf{A.x} = \mathbf{b} \tag{2.2}$$

onde

A $\rightarrow$ é a matriz dos coeficientes, de ordem N. Possui valores numéricos e estrutura quadrada, estabelecendo as relações lineares entre as variáveis;
x $\rightarrow$ é o vetor das incógnitas ou variáveis. Tem dimensão Nx1;
b $\rightarrow$ é vetor dos termos independentes, também com dimensão Nx1.

Nos sistemas de potência em particular, as equações matriciais em questão apresentam certas peculiaridades que podem contribuir para adequação junto aos critérios citados acima. Mais precisamente, a matriz **A** (equivale à matriz admitância de barra) quando de elevada dimensão, apresenta três características marcantes, a saber:

- É esparsa, ou seja, apresenta uma pequena porcentagem de elementos não-nulos, de modo que se pode ilustrar da seguinte maneira:

  Para uma rede com N = 500 barras (excluindo a barra de referência), onde o número médio de barras ligadas a cada barra é k = 3, existem três elementos significativos não diagonais por linha. Em 500 linhas haverá 150 elementos não-nulos fora da diagonal principal, ou melhor, 0,6% do número de elementos da matriz. Adicionando os 500 elementos diagonais, tem-se 2000 elementos não-nulos para os 25000 elementos da matriz.

- É simétrica em estrutura, isto é, seus elementos não-nulos estão dispostos de forma simétrica à diagonal principal. Quando não se verifica a presença de transformadores

com relação de transformação complexa (defasadores), também é numericamente simétrica.

- É diagonalmente dominante, isto é, em cada linha, o módulo do elemento diagonal é maior ou igual a soma dos módulos dos elementos não diagonais. Além disso, os elementos diagonais se concentram nas proximidades da diagonal principal.

Diante disso, a forma da matriz **A** pode ser aproximada pela Fig. 2.1.

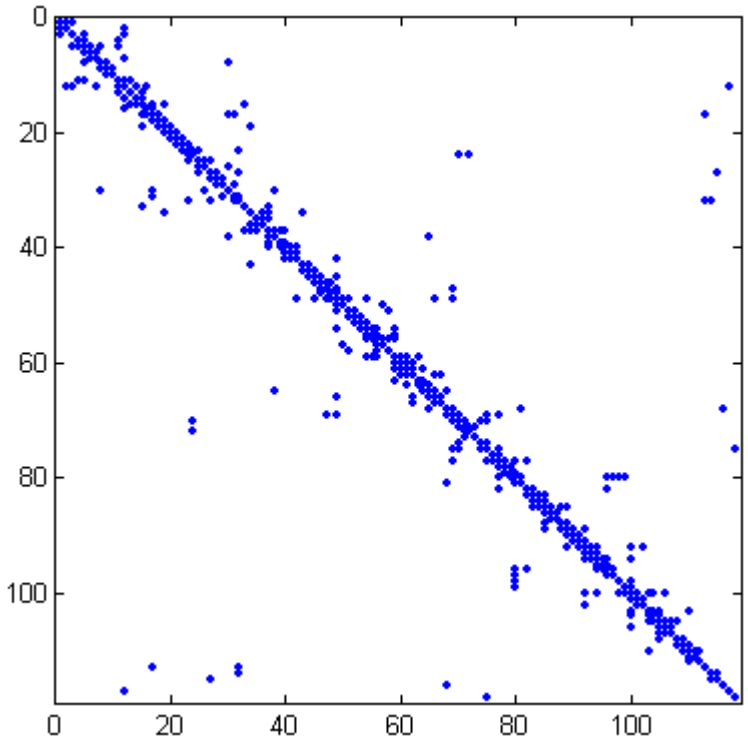

FIGURA 2.1 – Modelo de matriz esparsa, usado em Sistemas de Potência.

A área escura simbolizando os elementos não-nulos mostra a diagonal principal e suas proximidades. É bom lembrar que estas formas só são evidenciadas para um N elevado.

Torna-se facial então notar as vantagens que a matriz dos coeficientes pode oferecer, quando se almeja praticidade e precisão. Uma matriz com poucos elementos diferentes de zero necessitaria de um número reduzido de cálculos no processamento da solução de um sistema linear. Isso tornaria esse processamento mais rápido e diminuiria a propagação dos erros de arredondamento.

Armazenar compactamente uma matriz esparsa é o primeiro passo para resolver um sistema linear esparso. Uma quantia significativa de pesquisa tem sido feita sobre esquemas de armazenamento para matrizes esparsas.

É importante que uma estrutura de dado para uma matriz esparsa seja compacta, quanto ao armazenamento de seus elementos não-nulos, e ao mesmo tempo seus elementos sejam facilmente acessados. Usualmente esses dois atributos não são alcançáveis simultaneamente.

Os esquemas de armazenamento por índice armazenam os elementos não-nulos com informações auxiliares na qual podem ser utilizados para determinar onde o elemento não-nulo está localizado dentro da matriz original, dados cruciais em operações matriciais comuns. Alguns desses métodos podem requerer o armazenamento de três até cinco vezes o numero de elementos não-nulo da matriz.

A idéia básica é armazenar somente os elementos não-nulos da matriz (mais algumas informações adicionais) utilizando um conjunto de vetores e apontadores de tal forma que o espaço total de memória utilizado seja menor que o requerido para armazenar toda a matriz.

Existem muitos esquemas propostos para o armazenamento de matrizes esparsas considerando matrizes simétricas e assimétricas. Alguns apresentam facilidades para alterações de seus elementos enquanto que outros apresentam facilidades para operações utilizando as matrizes. Portanto, para cada esquema a eficiência difere.

A escolha do esquema de armazenamento a ser utilizado depende do problema que se quer resolver. A eficiência da resolução do problema pode variar em função do esquema utilizado.

Um esquema ideal de armazenamento indexado para matrizes esparsas gerais, em larga escala, teria as seguintes características:

- Usa uma quantidade mínima de espaço em memória;
- A posição original de um elemento na matriz pode ser recuperada facilmente;
- A álgebra fundamental de matriz pode ser feita com o mínimo esforço;
- Pode ser implementada eficientemente em computadores de alta performance;
- Minimiza a latência do acesso de memória;
- É de fácil compreensão.

Neste tópico serão descritos alguns dos principais esquemas de armazenamento compacto de matrizes esparsas usados na literatura.

2.3.1 – ESQUEMA DE ARMAZENAMENTO INDEXADO

Considere que $\mathbf{A} = \{\mathbf{a}_{i,j}\}$ é uma matriz esparsa n × n, para $0 \leq \mathbf{i}, \mathbf{j} \leq n - 1$, sendo que n é a ordem da matriz. Os elementos não-nulos da matriz **A**, para o esquema de armazenamento indexado (HU, 1995), são definidos como $\mathbf{a}_{i,j}$, seus índices linha após linha são definidos como $\mathbf{k}_{i,j} = (\mathbf{i} \times n + \mathbf{j})$; seus índices armazenados coluna após coluna são definidos como $\mathbf{k}_{i,j} = (\mathbf{j} \times n + \mathbf{i})$. Os índices serão armazenados na ordem crescente.

De acordo com a definição acima, pode-se armazenar uma simples matriz quadrada **A** de quarta ordem, linha após linha, ao checar a lista dos elementos $\mathbf{a}_{0,0}$, $\mathbf{a}_{0,1}$, $\mathbf{a}_{0,2}$, $\mathbf{a}_{0,3}$. $\mathbf{a}_{1,0}$, $\mathbf{a}_{1,1}$ ... , $\mathbf{a}_{3,3}$ e indicar o índice $(\mathbf{i} \times n + \mathbf{j})$ para cada elemento não-nulo $\mathbf{a}_{i,j}$, $0 \leq \mathbf{i}, \mathbf{j} \leq n - 1$. Tem-se uma melhor visualização dessa matriz na Figura 2.2, com o seu respectivo armazenamento indicado na Tabela 2.1.

$$\begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ \begin{pmatrix} 3 & 0 & 4 & 0 \\ 0 & 0 & 5 & 0 \\ 1 & 0 & 0 & 7 \\ 0 & 2 & 0 & 1 \end{pmatrix} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} \end{array}$$

FIGURA 2.2 – Matriz quadrada A de ordem 4×4.

TABELA 2.1 – Armazenamento indexado linha após linha para a matriz A

$\mathbf{k}_{i,j}$	0	2	6	8	11	13	15
$\mathbf{a}_{i,j}$	3	4	5	1	7	2	1

Similarmente, pode-se armazenar os elementos da matriz **A** coluna após coluna como mostrado na Tabela 2.2.

TABELA 2.2 – Armazenamento indexado coluna após coluna para a matriz A

$\mathbf{k}_{i,j}$	0	2	7	8	9	14	15
$\mathbf{a}_{i,j}$	3	1	2	4	5	7	1

Vantagens desse esquema de armazenamento indexado:

- Armazena somente os elementos não-nulos de uma matriz esparsa e associa somente um índice para cada elemento não-nulo. Acredita-se que o esquema de armazenamento usa o mínimo de espaço em memória para armazenar o índice de uma matriz esparsa, visto que alguns esquemas de armazenamento devem

armazenar todos os elementos não-nulos de uma matriz esparsa, e devem ter no mínimo um índice para indicar a posição original de cada elemento não-nulo.

- É fácil recuperar a localização original de um elemento através de seu índice. Suponha que o índice linha após linha para um elemento não-nulo é **k**. Efetuando-se a divisão inteira de **k** com n tem-se a posição da linha, enquanto que extraindo-se o resto da divisão inteira de **k** com n tem-se a posição da coluna.

- A transposta da matriz é também muito fácil de determinar. Por exemplo, o índice linha após linha para um elemento não-nulo de uma matriz A é **k**, então sua posição lógica dentro de $\mathbf{A}^t$ utiliza o resto da divisão inteira de **k** com n como a nova posição da linha, assim como a divisão inteira de **k** com n como a nova posição da coluna.

- É fácil realizar adição e subtração de matrizes esparsas. Aqueles elementos que possuem a mesma posição lógica têm o mesmo índice. Para realizar a adição ou subtração, é necessário somente somar ou subtrair o correspondente elemento e colocar o resultado na correspondente posição da matriz resultante. Aqueles elementos que aparecem somente em uma das matrizes necessitam ser inseridos, na seqüência, junto com os seus índices, dentro da matriz resultante.

- Este esquema de armazenamento utiliza apenas duas divisões para recuperar a posição original de um elemento não-nulo.

2.3.2 – ESQUEMA DE KNUTH

O Esquema de KNUTH (1968) baseia-se na compressão de dados da matriz esparsa tendo em vista a utilização de 7 (sete) vetores, sendo que um desses vetores é utilizado para armazenamento dos elementos não-nulos da matriz **A**, quatro vetores são utilizados na obtenção de elementos de u ma certa linha ou coluna da matriz e dois vetores para a posição da linha e coluna dos elementos. Este esquema possui a vantagem de poder acrescentar ou eliminar elementos facilmente e varrer as linhas e colunas eficientemente.

A descrição dos sete vetores utilizados na compressão dos dados é definida na Tabela 2.3.

TABELA 2.3 – Vetores e apontadores usados no armazenamento da matriz.

NA(k)	Armazena os elementos não-nulos da matriz **A**, em qualquer ordem.
I(k)	Armazena a linha do elemento de **NA(k)** encontrado na posição **k**.
J(k)	Armazena a coluna do elemento de **NA(k)** encontro na posição **k**.
NR(k)	Indica a posição **k** do próximo elemento não-nulo da linha onde se encontra o elemento de **NA(K)**. Caso este elemento não exista **k** = 0.
NC(k)	Indica a posição **k** do próximo elemento não-nulo da coluna onde se encontra o elemento de **NA(k)**. Caso este elemento não exista **k** = 0.
JR(i)	apontador de inıcio de linha **i** para a posição **k** de **NA(k)** do primeiro elemento não-nulo dessa linha.
JC(j)	Apontador de inıcio de coluna **j** para a posição **k** de **NA(k)** do primeiro elemento não-nulo dessa coluna.

Como um exemplo, considere a matriz **A** da Figura 2.3.

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

FIGURA 2.3 – Matriz esparsa no formato cheia usada na compressão de dados.

O armazenamento da matriz **A** é dado pela Tabela 2.4.

TABELA 2.4 – Preenchimento dos vetores e apontadores usados na compressão dos dados.

Posição k	**1**	**2**	**3**	**4**	**5**	**6**	**7**
NA(k)	6	9	4	7	5	2	8
I(k)	1	2	2	2	3	4	4
J(k)	2	1	2	4	1	2	4
NR(k)	0	3	4	0	0	7	0
NC(k)	3	5	6	7	0	0	0

Posição i, j	**1**	**2**	**3**	**4**
JR(i) = k	1	2	5	6
JC(j) = k	2	1	0	4

Para localizar o elemento $\mathbf{a_{4,4}}$ tem-se a posição **k** do primeiro elemento não-nulo da linha 4 dada por **JR(**4**)** = 6, onde na posição 6 está armazenado o elemento da linha **i** = 4 e coluna **j** = 2. O próximo elemento não-nulo está na posição **NR(**6**)** = 7, sendo que na posição 7 está armazenado o elemento da linha **i** = 4 e coluna **j** = 4, ou seja, **NA(**7**)** = 8 = $\mathbf{a_{4,4}}$. Como **NR(**7**)** = 0, não há mais elementos não-nulos na linha 4.

Para obtenção da linha e coluna do elemento armazenado na posição 5 tem-se **I(5)** = 3 e **J(5)** = 1, portanto, $\mathbf{a_{3,1}}$.

2.3.3 – ESQUEMA DE ZOLLENKOPF

É um esquema bastante utilizado em sistemas de potência, sendo que, não é o melhor esquema em termos de economia de armazenamento, mas apresenta boa flexibilidade de utilização.

O esquema de ZOLLENKOPF (1987) também pode ser utilizado para matrizes simétricas estrutural e numericamente. Ou seja:

- Estruturalmente simétrica – se $\mathbf{A_{i,j}} \neq 0$ então $\mathbf{A_{j,i}} \neq 0$.
- Numericamente assimétrica – em geral $\mathbf{A_{i,j}} \neq \mathbf{A_{j,i}}$.

2.3.3.1 – MATRIZES NUMERICAMENTE SIMÉTRICAS

Os elementos não-nulos são armazenados por comprimento de coluna dentro do vetor **CE**. Os índices para linha dos elementos dentro de **CE** são armazenados dentro de uma tabela paralela **ITAG**. A tabela acompanhante, **LNXT**, contém a localização do próximo elemento não-nulo dentro de **CE**, na ordem ascendente. A entrada 0 dentro de **LNXT** indica o ultimo termo de uma coluna.

As posições de início de uma coluna dentro de **CE** são armazenadas dentro da tabela **LCOL**. A tabela **NOZE** contém o número de elementos não-nulos dentro de cada coluna.

Como pode ser visto do exemplo, a posição de armazenamento inutilizada do vetor reservado **CE** e **LNXT** também deve ser ocupado por valores iniciais. As posições vaga do vetor **CE** e da ultima posição de tabela **LNXT** devem ser definidos como zero. As outras posições vagas de **LNXT** devem ser numeradas consecutivamente.

Aparte disso, a ordem da matriz (número de colunas e de linhas) é armazenado dentro de N e a primeira localização vaga dentro das tabelas **CE**, **ITAG** e **LNXT** devem ser armazenadas dentro de LF (no exemplo dado é N = 6 e LF = 21).

$$A = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{bmatrix} \times & \times & & \times & & \\ \times & \times & \times & & \times & \\ & \times & \times & & & \times \\ \times & & & \times & \times & \\ & \times & & \times & \times & \times \\ & & \times & & \times & \times \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix}$$

FIGURA 2.4 – Matriz esparsa no formato convencional usada na compressão de dados.

TABELA 2.5 – Preenchimento dos vetores e apontadores usados na compressão dos dados.

	LCOL	NOZE	ITAG	LNXT	CE
1	1	3	1	2	a_{11}
2	4	4	2	3	a_{21}
3	8	3	4	0	a_{41}
4	11	3	1	5	a_{12}
5	14	4	2	6	a_{22}
6	18	3	3	7	a_{32}
7	–	–	5	0	a_{52}
8	–	–	2	9	a_{23}
9	–	–	3	10	a_{33}
10	–	–	6	0	a_{63}
11	–	–	1	12	a_{14}
12	–	–	4	13	a_{44}
13	–	–	5	0	a_{54}
14	–	–	2	15	a_{25}
15	–	–	4	16	a_{45}
16	–	–	5	17	a_{55}
17	–	–	6	0	a_{65}
18	–	–	3	19	a_{36}
19	–	–	5	20	a_{56}
20	–	–	6	0	a_{66}
21	–	–	–	22	0
22	–	–	–	23	0
23	–	–	–	24	0
24	–	–	–	0	0

2.3.3.2 – MATRIZES SIMÉTRICAS EM ESTRUTURA E ASSIMÉTRICAS NUMERICAMENTE

O modo de armazenamento de uma matriz assimétrica com um modelo simétrico de elementos não-nulos difere do caso simétrico em dois pontos. Primeiro, os termos diagonais são armazenados dentro de uma tabela separada **DE**. Segundo, os termos fora da diagonal são armazenados em ambas as direções, isto é, eles são armazenados por comprimento de coluna dentro de **CE**, em seguida, por comprimento de linha na tabela paralela **RE**. Devido à

simetria na estrutura, é assumido que a tabela **ITAG** contenha os índices das linhas dos elementos armazenados dentro de **CE**, como também, os índices das colunas dos elementos armazenados dentro de **RE**. No caso que a linha e a coluna não tenham termos fora da diagonal, a respectiva posição dentro da tabela **LCOL** é definida como zero.

$$A = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{bmatrix} \times & \times & & \times & & \\ \times & \times & \times & & \times & \\ & \times & \times & & & \times \\ \times & & & \times & \times & \\ & \times & & \times & \times & \times \\ & & \times & & \times & \times \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix}$$

FIGURA 2.5 – Matriz esparsa no formato convencional usada na compressão de dados.

TABELA 2.6 – Preenchimento dos vetores e apontadores usados na compressão dos dados.

	LCOL	NOZE	DE	ITAG	LNXT	CE	RE
1	1	3	a_{11}	2	2	a_{21}	a_{12}
2	3	4	a_{22}	4	0	a_{41}	a_{14}
3	6	3	a_{33}	1	4	a_{12}	a_{21}
4	8	3	a_{44}	3	5	a_{32}	a_{23}
5	10	4	a_{55}	5	0	a_{52}	a_{25}
6	13	3	a_{66}	2	7	a_{23}	a_{32}
7	–	–	–	6	0	a_{63}	a_{36}
8	–	–	–	1	9	a_{14}	a_{41}
9	–	–	–	5	0	a_{54}	a_{45}
10	–	–	–	2	11	a_{25}	a_{52}
11	–	–	–	4	12	a_{45}	a_{54}
12	–	–	–	6	0	a_{65}	a_{56}
13	–	–	–	3	14	a_{36}	a_{63}
14	–	–	–	5	0	a_{56}	a_{65}
15	–	–	–	–	16	0	0
16	–	–	–	–	17	0	0
17	–	–	–	–	18	0	0
18	–	–	–	–	19	0	0
19	–	–	–	–	20	0	0
20	–	–	–	–	21	0	0
21	–	–	–	–	22	0	0
22	–	–	–	–	23	0	0
23	–	–	–	–	24	0	0
24	–	–	–	–	0	0	0

2.4 – TÉCNICAS DE APROVEITAMENTO DA ESPARSIDADE

Além de armazenar matrizes esparsas em um formato compacto e de fácil recuperação da posição original de seus elementos, há a necessidade de se aplicar técnicas que aproveitem a esparsidade da matriz e diminuam o número de operações sobre os elementos da matriz.

Uma técnica bastante utilizada é a ordenação, cuja função é preservar a esparsidade e diminuir o número de operações elementares durante a solução de um sistema linear esparso.

2.4.1 – RESOLUÇÃO DE SISTEMAS DE EQUAÇÕES ALGÉBRICAS LINEARES ENVOLVENDO MATRIZES ESPARSAS

Uma das operações envolvendo matrizes esparsas, que merece uma certa atenção especial, é a solução de um sistema de equações algébricas lineares definida pela Equação 2.3.

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.3}$$

$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b} \tag{2.4}$$

Como a inversa da matriz **A**, na maioria das situações, não é esparsa, solucionar a equação (2.3) através da equação (2.4) contraria as técnicas de esparsidade.

Um processo adequado para solucionar tal equação seria o processo de eliminação de variáveis (realização de combinações lineares entre equações) para a obtenção da solução do problema sem a necessidade de se inverter a matriz de coeficientes, tal processo é chamado de Eliminação de Gauss, onde também pode ser chamado de fatoração triangular ou triangularização.

Pode-se resolver a equação (2.3), através da eliminação de Gauss, partindo-se de uma matriz aumentada contendo **A** e **b** e obtendo-se uma nova matriz aumentada que contém a solução **x** como mostrada na equação (2.5):

$$\left[\mathbf{A} \mid \mathbf{b}\right] \Rightarrow \left[\mathbf{I} \mid \mathbf{x}\right] \tag{2.5}$$

Em que **I** é a matriz identidade. Para a obtenção de **I** a partir de **A** deve-se realizar operações (combinações lineares) entre as linhas de **A**, de forma a obter os seguintes passos:

- Tornar todos os elementos do triângulo inferior de **A** iguais a 0;
- Tornar todos os elementos da diagonal de **A** iguais a 1;
- Tornar todos os elementos do triângulo superior de **A** iguais a 0.

As operações acima serão aplicadas também sobre o vetor **b** que resultará na solução **x**.

Durante o processo de triangularização, podem aparecer elementos não-nulos em determinadas posições de uma linha, onde, antes, havia elementos nulos. Estes novos elementos são chamados de *fill-ins*.

2.4.2 – ORDENAÇÃO

Durante o processo de solução do sistema linear dado pela equação (2.1), utilizando-se o processo de Eliminação Gaussiana (EG), faz-se necessário a aplicação de procedimentos que objetivam atender as exigências de memória e rapidez de processamento. São essencialmente dois métodos distintos, mas que devem ser desenvolvidos em sincronia, pois de sua interrelação depende um bom desempenho de um processo de triangularização.

Durante o processo de Eliminação Gaussiana, depara-se com o problema dos *fill-ins*, que corrompem o desempenho das operações envolvendo matrizes esparsas. Uma forma de contornar esse problema seria aplicar técnicas de ordenação.

A ordenação é definida como o processo pelo qual se escolhe uma seqüência de eliminação que será útil para atingir um determinado objetivo. Neste caso especificamente, a diminuição do número de elementos não-nulos que apareciam numa EG convencional.

A escolha de uma seqüência adequada garante a esparsidade, conseqüentemente minimiza o armazenamento de informações e ainda diminui o número de operações aritméticas (O que torna menores os erros de arredondamento). As Figuras 2.6 e 2.7 podem ilustrar este fato.

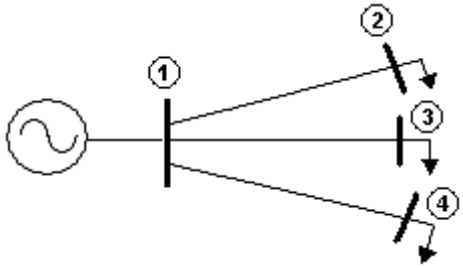

FIGURA 2.6 – Sistema de Quatro barras não ordenado

$$\begin{bmatrix} X & X & X & X \\ X & X & & \\ X & & X & \\ X & & & X \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

FIGURA 2.7 – Matriz Admitância Genérica do Sistema

$$\begin{bmatrix} X & X & X & X \\ X & X & \otimes & \otimes \\ X & \otimes & X & \otimes \\ X & \otimes & \otimes & X \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

FIGURA 2.8 – Matriz Admitância após Normalização da Primeira Linha

A matriz de admitâncias de barras do sistema da Figura. 2.6 cujas barras foram enumeradas desordenadamente, é mostrada na Figura 2.7. Os *X*'s representam os elementos não-nulos e os espaços, os elementos nulos da matriz. Na Figura. 2.7, a matriz está no seu estado inicial (nenhum passo da EG foi implementado). Posteriormente, o primeiro passo da eliminação (a normalização da primeira linha) foi realizado e se obtém então a matriz observada na Fig. 2.8. O símbolo $\otimes$ representa os elementos não-nulos introduzidos neste passo. Com isso, é possível notar a destruição da esparsidade em apenas uma etapa realizada.

Renumerando então o sistema da Figura. 2.9, de forma que a barra 1 passe a ser a nova barra 4 e vice-versa (matriz inicial representada na Figura. 2.10), realiza-se novamente o primeiro passo e obtem-se como resultado a matriz apresentada na Figura 2.11. Neste caso, a nova matriz é tão esparsa quanto a inicial, o que é bastante vantajoso.

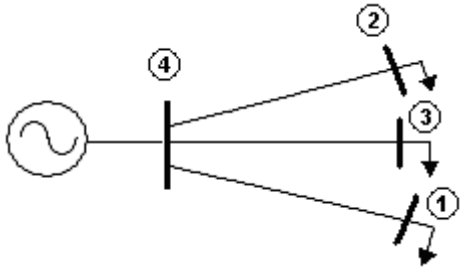

FIGURA 2.9 – Sistema renumerado (Ordenado) em ordem crescente do número de interligações entre cada barra.

$$\begin{matrix} & 4 & 2 & 3 & 1 \\ \begin{bmatrix} X & & & X \\ & X & & X \\ & & X & X \\ X & X & X & X \end{bmatrix} & \begin{matrix} 4 \\ 2 \\ 3 \\ 1 \end{matrix} \end{matrix}$$

FIGURA 2.10 – Sua nova Matriz Admitância

$$\begin{matrix} & 4 & 2 & 3 & 1 \\ \begin{bmatrix} X & & & X \\ & X & & X \\ & & X & X \\ X & X & X & X \end{bmatrix} & \begin{matrix} 4 \\ 2 \\ 3 \\ 1 \end{matrix} \end{matrix}$$

FIGURA 2.11 – Matriz Admitância após Normalização da Primeira linha

Pode-se ainda ilustrar que, ao se conservar a esparsidade, é possível diminuir o número de operações aritméticas, assim como as exigências de memória necessária ao armazenamento da matriz fatorada.

Pode-se diminuir o número de fill-ins gerados no processo de fatoração trocando posições de linhas e colunas da matriz renumerando os nós do circuito.

Existem basicamente três esquemas de ordenação que são aplicados de acordo com o tipo de processamento e/ou topologia do sistema (TINNEY & WALKER, 1967):

- **Esquema I** – Fatorar a matriz na ordem inversa do grau dos nós, ou seja, as colunas ou linhas com menos elementos (na matriz original) devem ser fatoradas primeiro;
- **Esquema II** – A cada passo do procedimento eliminatório, a próxima linha ou coluna a ser fatorada é aquela que tiver o menor número de elementos naquele estágio da fatoração;
- **Esquema III** – Em cada estágio simula-se a fatoração de todas as colunas ou linhas restantes e toma-se aquela que gerar o menor número de fill-ins.

2.5 – CONCLUSÃO

Este capítulo apresentou uma breve introdução e revisão das técnicas de esparsidade, descrevendo alguns dos principais esquemas de armazenamento compacto de matrizes esparsas.

Alguns esquemas abordados apresentaram facilidade para alterações e inserção de seus elementos enquanto que outros apresentam facilidades para operações utilizando as matrizes. É relevante destacar que existe uma relação fundamental entre o acesso de cada elemento e a velocidade das operações matriciais.

A escolha do esquema de armazenamento a ser utilizado depende do problema que se quer resolver, ou seja, a eficiência da resolução do problema pode variar em função do esquema utilizado.

CAPÍTULO 3 – PROGRAMAÇÃO ORIENTADA A OBJETO

3.1 – INTRODUÇÃO

Recentemente, as bibliotecas orientadas a objeto desenvolvidas para matrizes esparsas, têm recebido atenção significativa de pesquisadores na área da engenharia e ciência da computação.

Desta forma a Programação Orientada a Objeto tem sido aceita na engenharia de Sistemas de Potência como uma alternativa viável à tradicional programação procedural. Em todos esses desenvolvimentos, C++ tem sido uma das escolhas universais para implementação (PANDIT et al., 2001).

A razão de se optar por C++ como linguagem de implementação é que ela é uma das poucas linguagens que suporta ambas das abstrações de dados, ou seja, é uma linguagem híbrida que combina a capacidade numérica de C (Programação Estrutural) com a Programação Orientada a Objeto (HAKAVIK & HOLEN, 1994).

Neste capítulo será abordada uma breve descrição teórica da Programação Orientada a Objeto aplicada a C++, definindo os principais termos e conceitos utilizados no desenvolvimento do presente trabalho, além de outros conceitos importantes também presentes.

3.2 – LINGUAGEM C++

A linguagem C teve origem na linguagem B (desenvolvido por Ken Thompson, em 1970), sendo desenvolvido por Denis Richard em 1972.

O ano de 1978 foi um ano histórico para a linguagem C, sendo editado o livro "*The C Programing language*", responsável pela divulgação de C. A linguagem C tem sido utilizada em programas estruturados.

Em 1980, desenvolveu o C++ como um superconjunto de C e foi inicialmente chamado de C com classes. Este apresenta uma série de vantagens em relação ao C, que se mostrou extremamente eficiente nos vários campos da programação, alem de ser uma linguagem orientada a objetos.

3.3 – CONCEITOS DE PROGRAMAÇÃO ORIENTADA A OBJETO (POO)

Para tentar solucionar o problema do baixo reaproveitamento de código, tomou corpo a idéia da programação orientada a objeto (POO). A POO não é nova, sua formulação inicial data de 1960. Porém somente a partir dos anos 90 é que passou a ser usada. Hoje, todas as grandes empresas de desenvolvimento de programas têm desenvolvido os seus softwares usando a programação orientada a objeto.

Ela difere da programação estruturada, pois funções e dados formam juntas o objeto. Esta abordagem cria uma nova forma de analisar, projetar e desenvolver programas. De uma forma mais abstrata e genérica, que permite um maior reaproveitamento dos códigos e facilita a manutenção.

A POO não é somente uma nova forma de programar, é uma nova forma de atuar sobre um problema, de forma abstrata, utilizando conceitos do mundo real.

A POO tem uma série de conceitos que auxiliam as pessoas a delinear claramente o problema e a identificar os objetos e seus relacionamentos.

Descreve-se a seguir os conceitos básicos de análise orientada a objeto, isto é, a abstração, o objeto, as classes, os atributos, os métodos, as heranças, o polimorfismo etc.

3.3.1 – ABSTRAÇÃO

É o processo de identificação do objeto e seus relacionamentos. A análise orientada a objeto permite concentrar-se no objeto e sua função, sem se preocupar em como ele o faz. A abstração se dá em diferentes níveis: inicialmente abstrai-se o objeto, de um conjunto de objetos, cria-se um conjunto de classes relacionadas, e então cria-se uma biblioteca de classes.

3.3.2 – OBJETO (OU INSTÂNCIA)

Objetos são coisas do mundo real ou imaginário, que podem de alguma forma ser identificados, como uma pedra, uma caneta, etc.

Um objeto tem determinadas propriedades que o caracterizam, e que são armazenadas no próprio objeto. As propriedades de um objeto são chamadas ainda de atributos.

O objeto interage com o meio e em função de excitações que sofre, realiza determinadas ações que alteram o seu estado (seus atributos). Os atributos de um objeto são dinâmicos, eles sofrem alterações com o tempo.

Para a POO, um objeto é uma entidade única que reúne atributos e métodos, ou seja, reúne as propriedades do objeto e as reações às excitações que sofre.

A instância é um outro nome que se dá ao objeto, geralmente se refere a um objeto específico.

3.3.3 – CLASSES

Para a POO, uma classe é um conjunto de código de programação que incluem a definição dos atributos e dos métodos necessários para a criação de um ou mais objetos.

A classe contém todas as descrições da forma do objeto, é um molde para a criação do objeto, é uma matriz geradora de objetos, é uma fábrica de objetos. Uma classe também é um tipo definido pelo usuário.

Os objetos com a mesma estrutura de dados e com as mesmas operações são agrupados em uma classe. Um objeto contém uma referência implícita a sua classe, e sabe a qual classe pertence.

Abaixo, é apresentado um exemplo de uma classe em C++ e seus componentes:

```
class Ponto
{
      private:                                  // Membros de dados privados
      int x;                                    // Membro de dado (variável x)
      int y;                                    // Membro de dado (variable y)

      public:                                   // Funções e membros de dados públicos
      char NomeObj[ 30 ];                       // Membro de dados (variável NomeObj)
      Ponto( int a, int b ) { x =a; y = b; }    // Construtor
      void SetPonto( int a, int b ) { x =a; y = b; }  // Função membro
      int GetX( ) { return x; }                 // Função membro
      int GetY( ) { return y; }                 // Função membro
};
```

3.3.4 – ACESSANDO DADOS E FUNÇÕES MEMBRO

Dentro do escorpo de uma classe, os membros da classe são imediatamente acessíveis por todas as funções membros daquela classe podendo ser referenciados por nome. Fora do escorpo de uma classe, os membros da classe são referenciados através de um dos *handles* (um nome de objeto, uma referência a um objeto ou um ponteiro para um objeto) de um objeto.

Os operadores utilizados para acessar membros de classes são divididos entre o operador de seleção de membro ponto (.) que é combinado com o nome de um objeto ou uma referência a um objeto para acessar os membros do objeto e o operador de seleção de

membro seta (->) que é combinado com um ponteiro para um objeto para acessar os membros daquele objeto.

Tem-se em seguida um exemplo de como acessar os dados e funções membros.

```
Class A {
public:
int x;
void print() { cout << x << endl; }
};

int main()
{
        A obj,              // Cria objeto do tipo A
        *objPtr = &obj,     // Ponteiro para obj
        &objRef = obj;      // Referência para obj
        // Atribui 7 a x e imprime usando o nome do objeto
        obj.x = 7;          // Atribui 7 para o membro de dado x
        obj.print();        // Chama a função membro print
        // Atribui 8 a x e imprime usando uma referência
        objRef.x = 8;       // Atribui 8 para o membro de dado x
        objRef.print();     // Chama a função membro print
        // Atribui 10 a x e imprime usando um ponteiro
        objPtr->x = 10;     // Atribui 10 para o membro de dado x
        objPtr->print();    // Chama a função membro print
        return 0;
}
```

3.3.5 – CONSTRUTOR

Construtor é uma função-membro que é executada quando é criada uma instância de classe, garantindo a inicialização consistente de uma instância de classe. O nome do construtor é o mesmo que o nome da classe. Pode-se declarar um construtor de duas maneiras:

- Construtor padrão (defaut);
- Construtor parametrizado, onde pode-se inicar os atributos do objetos através de variáveis parametrizadas no construtor.

O próprio compilador define qual construtor será chamado (se os dois forem declarados) no momento em que uma instância de classe está sendo criada. Um construtor nunca é chamado por um objeto da classe, mas sempre quando é criada uma instância dessa classe. O construtor não deve retornar nenhum valor. O motivo é que ele é chamado diretamente pelo sistema e não há como recuperar um valor de retorno.

3.3.6 – DESTRUTOR

O destrutor é uma função membro chamada pelo sistema, quando um objeto sai de escorpo ou, quando em alocação dinâmica, tem seu ponteiro desalocado.

Sua tarefa é limpar o ambiente de trabalho do objeto, o que geralmente significa desalocar memória, fechar arquivo ou sinalizar a outros objetos o que está sendo destruído.

O nome do destrutor é o mesmo nome da classe precedido do caractere “til” (~), sendo que este não recebe nenhum argumento e não retorna nenhum valor.

3.3.7 – ATRIBUTOS (PROPRIEDADES/VARIÁVEIS)

A todo objeto pode-se relacionar alguns atributos (propriedades). Na programação orientada a objeto, os atributos são definidos na classe e armazenados de forma individual ou coletiva pelos objetos.

Quando um atributo é dividido entre todos os objetos criados, ele é armazenado na classe, assim definido como atributo de classe (coletivo). Quando um atributo é individual ele é armazenado no objeto, portanto, definido como atributos de objeto (individual).

3.3.8 – MÉTODOS (SERVIÇOS/FUNÇÕES)

A todo objeto pode-se relacionar determinados comportamentos, ações e reações.

Na POO, as ações ou comportamentos dos objetos são chamados de métodos. Um método é uma função, um serviço fornecido pelo objeto.

Os comportamentos dos objetos são definidos nas classes através dos métodos e servem para manipular e alterar os atributos do objeto (alteram o estado do objeto).

Os objetos reagem ao meio que o envolve de acordo com as excitações que sofre.

Em POO essas excitações são representadas por mensagens que são enviadas a um objeto. Uma mensagem pode ser gerada pelo usuário, por exemplo, ao clicar o mouse.

3.3.9 – HERANÇA

Herança é a propriedade de se criar classes que se ampliam a partir de definições básicas de classes mais simples e genéricas para classes mais complexas e específicas.

Na POO, herança é o mecanismo em que uma classe derivada compartilha métodos e atributos de sua classe base.

A herança está relacionada às hierarquias e as relações entre os objetos.

As técnicas de POO facilitam o compartilhamento de código através dos conceitos de herança. Além do maior compartilhamento do código a POO reduz a codificação em função da maior clareza dos diagramas desenvolvidos, como mostrado na Figura 3.1.

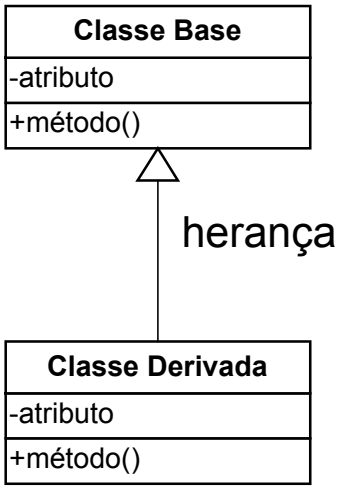

FIGURA 3.1 – Representação de uma herança simples

Após a criação de uma classe derivada, esta tem acesso a todos os dados públicos da classe base. Todos os métodos declarados como públicos da classe base estão automaticamente disponíveis para a classe derivada. Se uma classe derivada quiser acessar os dados privados da classe base não terá acesso. Para possibilitar este acesso, os dados privados na classe base devem ser declarados como protegidos.

Pode-se também chamar um método da classe base como se fosse chamar um método desta mesma classe.

O uso de uma biblioteca de classes oferece uma grande vantagem sobre o uso de uma biblioteca de funções: o programador pode criar classes derivadas de classes bases de bibliotecas. Isto significa que, sem alterar a classe base, é possível adicionar características diferentes que a tornarão capaz de executar o desejado.

O uso de classes derivadas aumenta a eficiência da programação pela não necessidade da criação de códigos repetitivos.

A uma função de biblioteca não se pode adicionar outras implementações a não ser que ela seja reescrita ou que se tenha seu código fonte para alterá-la e recompilá-la.

Abaixo é apresentado um exemplo de herança em C++.

```
class Ponto  // Classe Base
{
      private:   // Membros de dado privados
      int x;                                          // Membro de dado (variável x)
      int y;                                          // Membro de dado (variable y)

      public:    // Funções membro e membros de dado públicas
      Ponto( int a, int b ) { x =a; y = b; }          // Construtor
      void SetPonto( int a, int b ) { x =a; y = b; }  // Função-membro
      int GetX( ) { return x; }                       // Função-membro
      int GetY( ) { return y; }                       // Função-membro
};

class Circulo: private Ponto          // Classe derivada (herança)
{
     private:
     int raio;

     public:
     Circulo( int a, int b, int r ): Ponto( int a, int b) { raio = r;}          // Construtor
     void SetCirculo(int a, int b, int r) { SetPonto( a, b ); raio = r;}    // Função-membro
};
```

No exemplo acima, a classe derivada “Circulo” possui em seu construtor uma chamada para o construtor da classe base Ponto.

Pode-se também chamar um método da classe base como se fosse chamar um método desta mesma classe. No exemplo acima se tem esta situação na definição do método SetCirculo, em que este faz uma chamada ao método SetPonto da classe base.

3.3.9.1 – HERANÇA PÚBLICA E DERIVADA

Quando se cria uma classe derivada com a clausula "**public**", estamos dizendo que os dados públicos da classe base serão dados públicos na classe derivada e, os dados protegidos serão os dados protegidos da classe derivada. A classe derivada não terá acesso aos dados privados. Pode-se criar uma classe com a clausula "**private**", com isso tanto os dados públicos como os protegidos da classe base serão dados privados da classe derivada. Com isso, a classe derivada não terá acesso a nenhum dado da classe base.

3.3.9.2 – HERANÇA MÚLTIPLA

Quando uma classe herda as propriedades de uma única classe base ela é definida como herança simples, enquanto que se uma classe tem mais de uma classe base ela é definida como herança múltipla.

Quando uma classe herda as características de mais de uma classe-base. A Figura 3.2 mostra o exemplo de herança múltipla.

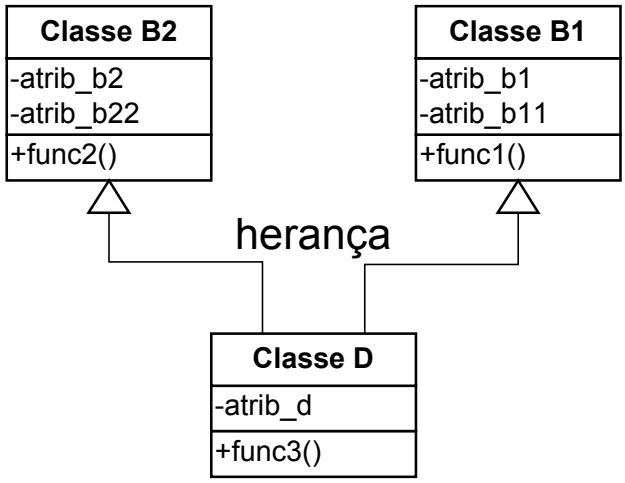

FIGURA 3.2 – Herança Múltipla

3.3.10 – POLIMORFISMO

A palavra polimorfismo em Programação orientada a objetos assume o sentido de que com um único nome para uma função-membro pode-se definir várias funções distintas. Duas ou mais funções-membro podem ter o mesmo nome, mas um código independente. A situação anterior é bastante utilizada em classes derivadas a partir de herança simples ou múltipla.

Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma classe base podem invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos, especializados para cada classe derivada, referenciando-se a um objeto do tipo da classe base. A decisão sobre o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada durante a execução.

É importante observar que, quando polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução.

O polimorfismo é possível devido à disponibilidade da herança e outras propriedades do C++. Em C++ é possível para um ponteiro de uma classe base apontar para um objeto criado de uma classe derivada, sem ter que se preocupar com o casamento de tipos.

3.3.11 – SOBRECARGA DE OPERADORES

Para certos tipos de classes, especialmente classes matemáticas, a notação de chamada de função não traz boa legibilidade para determinados tipos de manipulações com objetos. Para estes tipos de classes seria mais plausível usar o extenso conjunto de operadores primitivos de C++ para especificar manipulações de objetos.

Embora a sobrecarga de operadores não esteja presente em todas as linguagens de programação orientada a objeto, como a linguagem Java, em C++ é permitido ao programador sobrecarregar a maioria dos operadores primitivos para serem usados conforme o contexto em que são usados.

Define-se a sobrecarga de operador como as tarefas que determinado operador realiza sobre uma classe definida pelo programador, onde para sobrecarregar um determinado operador, escreve-se uma definição de função; o nome da função deve ser a palavra-chave “operator” seguida pelo símbolo do operador que está sendo sobrecarregado.

A sobrecarga de operadores não é automática, o programador deve escrever funções que sobrecarregam operadores para executar as operações desejadas.

Abaixo segue um exemplo de sobrecarga de operador em C++.

```
class complex
{
      private:
      float real;
      float imag;

      public:
      complex( float r, float i) { real = r; imag = i; }
      complex &operator+( complex & );
};

complex &complex::operator+( complex &a )
{
  static complex c(0,0);
  c.real = real + a.real;
  c.imag = imag + a.imag;
  return c;
}

void main(void)
{
  complex a(1,1), b(2,2), c(0,0);
  c = a + b;
}
```

3.4 – CONCLUSÃO

Este capítulo abordou uma breve revisão sobre os conceitos de Programação Orientada a Objeto, assim como sua aplicação na linguagem de programação C++, visto que esta é uma boa plataforma para o desenvolvimento de rotinas de esparsidade por suportar ambas as abstrações de dados, ou seja, combina a capacidade numérica de C (Programação Estrutural) com a Programação Orientada a Objeto.

Um dos pontos fortes da linguagem C++ é a sobrecarga de operadores, bastante utilizada nas rotinas de esparsidade, encasulando muitas das operações matriciais através de seus respectivos operadores matemáticos, tornando os códigos mais legíveis e fáceis de depurar.

CAPÍTULO 4 – REPRESENTAÇÃO COMPLETA E ORDENADA POR COMPRIMENTO DE LINHA (RCOCL)

4.1 – INTRODUÇÃO

Neste capítulo, será feita a descrição do esquema de armazenamento compacto RCOCL (GUSTAVSON, 1978) para matrizes esparsas, assim como a descrição dos algoritmos das principais operações matriciais utilizadas. Sua descrição e notação utilizam conceitos de programação orientada a objeto, como o conceito de classes e objetos. Os algoritmos das operações matriciais objetivam bom desempenho, boa alocação de memória, baixa complexidade e boa legibilidade para aplicações envolvendo matrizes esparsas de grandes dimensões.

Com o objetivo de consolidar os tópicos abordados neste capítulo, serão apresentados alguns gráficos no intuito de mostrar o desempenho dos algoritmos desenvolvidos. Para tal estudo foram feitas, primeiramente, medições do tempo de processamento das principais operações matriciais, para uma seqüência de matrizes com grau de esparsidade de 96%.

Os testes foram desenvolvidos em um microcomputador AMD Duron XP 1100MHz, com 247 MB de memória RAM.

4.2 – ARMAZENAMENTO RCOCL (REPRESENTAÇÃO COMPLETA E ORDENADA POR COMPRIMENTO DE LINHA)

Quando uma matriz esparsa de grande dimensões de ordem $\mathbf{m} \times \mathbf{n}$, onde $\mathbf{m}$ e $\mathbf{n} \in \mathrm{IN}^*$, contém somente alguns elementos não-nulos (um caso típico), é certamente ineficiente (e freqüentemente impossível) alocar espaço de memória para todos os $\mathbf{m} \cdot \mathbf{n}$ elementos. Até mesmo se fosse possível alocar tal espaço, seria ineficiente ou proibitivo, em tempo de máquina, executar laços sobre a matriz à procura de elementos não-nulos. Obviamente, algum tipo de esquema de armazenamento é requerido, um que armazene só elementos não-nulos da matriz, juntamente com informações auxiliares suficientes para determinar a posição dos elementos e como estes podem ser utilizados em operações matriciais. Infelizmente, não há um esquema padrão de uso geral. Neste trabalho é favorecido o esquema de armazenamento RCOCL (Representação Completa e Ordenada por Comprimento de Linha) (GUSTAVSON, 1978). A vantagem deste esquema é que o mesmo requer aproximadamente, duas vezes o número de elementos não-nulos da matriz

(outros métodos podem requerer até três ou cinco vezes) e utilizar nas operações matriciais algoritmos com baixa complexidade e bom desempenho.

O esquema RCOCL é composto de três arranjos unidimensionais:

- **ija[k]** que armazenará os índices **j** de coluna dos elementos não-nulos ordenados por linha e dentro de cada linha ordenados por colunas;
- **sa[k]** que armazenará os elementos não-nulos da matriz ordenados por linha e dentro de cada linha ordenados por colunas;
- **index[i]** que armazenará os índices que determinam o início das informações referentes as linhas de índice **i** nos arranjos **ija[k]** e **sa[k]**, com $\mathbf{i} \in \{1,2,3,...,\mathbf{n},\mathbf{n}+1\}$.

Para exemplificar o uso do esquema RCOCL, considere a matriz 5×5 dada pela Figura 4.1.

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

FIGURA 4.1 – Matriz quadrada de ordem 5×5.

Para o esquema RCOCL, tem-se o armazenamento compacto da matriz dado pela Tabela 4.1.

TABELA 4.1 - Armazenamento compacto usando o esquema RCOCL.

Índice k	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
ija[k]	1	3	2	2	3	4	5	4	5
sa[k]	3	1	4	7	5	9	2	6	5
Índice i	**1**	**2**	**3**	**4**	**5**	**6**			
Index[i] = k	1	3	4	7	8	10			

Convém ressaltar que o armazenamento RCOCL por natureza estrutural, ou seja, por indexar as linhas da matriz, torna-se mais eficiente, em termos de armazenamento, quanto maior for o número de colunas em relação às de linhas, pois o arranjo **index** diminui. Por exemplo, para uma matriz esparsa de 1×7000, tem-se o tamanho de **index** igual a 2 e o tamanho de **sa** e **ija** dependentes do grau de esparsidade.

4.2.1 – VARIÁVEIS AUXILIARES PARA O ESQUEMA RCOCL

Como os arranjos **ija**, **sa** e **index** não contêm nenhuma informação das dimensões da matriz, são necessárias variáveis adicionais para conter tais informações:

- **numlin** que armazenará o número de linhas;
- **numcol** que armazenará o número de colunas.

Durante o processamento dos dados e as operações matriciais muitos elementos assumirão valores absolutos muito próximos de zero (exemplo: $1{,}3333333 \cdot 10^{-25}$). Estes valores sendo diferente de zero passam a ser armazenados e processados nas operações, ocasionando, progressivamente, um mau desempenho. Com o objetivo de eliminar o armazenamento e processamento de valores muito próximos de zero defini-se mais uma variável.

- **limiar** que armazenará um valor real positivo. Se o valor absoluto do elemento da matriz ou resultante de uma operação for menor que **limiar** este valor é considerado nulo.

O esquema RCOCL e as variáveis auxiliares montam a seguinte estrutura de dados para matrizes esparsas de ordem $m \times n$:

- O valor de **numlin** é **m**;
- O valor de **numcol** é **n**;
- Nenhum elemento de valor absoluto menor que **limiar** é armazenado e processado.
- Os elementos não-nulos da linha **i** estão em **sa[k]** onde **index[i]** $\leq$ **k** $\leq$ **index[i+1]-1**;
- O comprimento do arranjo **index** é **numlin+1**;
- O comprimento dos arranjos **ija** e **sa** são iguais a **index[numlin+1]-1**.

4.3 – ARMAZENAMENTO PARA VETORES ESPARSOS: RCOCV (REPRESENTAÇÃO COMPLETA E ORDENADA POR COMPRIMENTO DE VETOR)

A multiplicação e a solução de sistemas lineares executam operações lineares sobre suas linhas. Estas operações lineares geralmente são executadas por sub-rotinas cujos argumentos são as linhas esparsas da matriz. Para tanto, pode-se usar a orientação a objeto para sobrecarregar as sub-rotinas de operações lineares nos operadores

matemáticos e dar origem a um objeto para vetores esparsos cujos elementos podem ser os elementos de uma linha da matriz esparsa. No entanto o objeto para vetores esparsos deve dispor de um modo de armazenamento. Este modo de armazenamento é uma extensão do modo RCOCL que aqui se denomina de RCOCV (Representação Completa e Ordenada por Comprimento do Vetor).

O esquema RCOCV monta dois arranjos unidimensionais e duas variáveis auxiliares:

- **ija[k]** que armazenará em ordem crescente os índices **j** dos elementos não-nulos.
- **sa[k]** que armazenará os elementos não-nulos do vetor na ordem crescente de seus índices **j**.
- **kov** que armazenará o índice que determina o início das informações referentes ao elementos do vetor.
- **kfv** que armazenará o índice que determina o fim das informações referentes ao elementos do vetor.

Observe que o modo de armazenamento para vetores esparsos é extremamente semelhante ao modo de armazenamento por linha indexada para matrizes esparsas. Isto facilita na elaboração dos algoritmos de transferência de elementos de uma linha de uma matriz com armazenamento RCOCL para um vetor esparso com armazenamento RCOCV.

Como exemplo, considere a matriz **B** de ordem 4×7, dada pela Figura 4.2.

$$\mathbf{B} = \begin{bmatrix} 3 & 0 & 1 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 1 \\ 0 & 7 & 5 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

FIGURA 4.2 – Matriz de ordem 4×7.

Para o esquema RCOCL, tem-se a Tabela 4.2 para o armazenamento compacto da matriz **B**.

TABELA 4.2 – Armazenamento compacto da matriz B utilizando o esquema RCOCL.

Índice k	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
ija[k]	1	3	6	2	7	2	3	4	7
sa[k]	3	1	3	4	1	7	5	9	2
Índice i	**1**	**2**	**3**	**4**	**5**				
index[i] = k	1	4	6	9	10				

Portanto o armazenamento compacto da segunda linha da matriz **B** no esquema RCOCV tem-se a Tabela 4.3, onde **kov** = 1 e **kfv** = **index[**2+1**]** - **index[**2**]**.

TABELA 4.3 – Armazenamento compacto da segunda linha da matriz B, utilizando o esquema RCOLV.

Índice k	**1**	**2**
ija[k]	2	7
sa[k]	4	1

A principal vantagem do esquema RCOCV é a possibilidade de utilizar os endereços de memória (ponteiros) do esquema RCOCL no momento em que as operações lineares da multiplicação e solução de sistemas lineares são executadas. A Tabela 4.4 ilustra, o armazenamento RCOCV usando (através de ponteiros) os arranjos **ija** e **sa** do esquema RCOCL.

TABELA 4.4 – Armazenamento compacto da segunda linha da matriz B, utilizando o esquema RCOLV.

Índices	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
ija[k]	1	3	6	2	7	2	3	4	7
sa[k]	3	1	3	4	1	7	5	9	2
Kov	**index[i]**								
Kof	**index[i**+1**]**-1								

4.3.1 – VARIÁVEIS AUXILIARES PARA O ESQUEMA RCOCV

Como os arranjos **ija**, **sa** e as variáveis **kof e kfv** não contêm nenhuma informação da dimensão do vetor, defini-se a variável:

- **tamanho** que armazenará a dimensão do vetor.

Com o objetivo de eliminar o armazenamento e processamento de valores muito próximos de zero defini-se mais uma variável:

- **limiar** que armazenará um valor real positivo. Se o valor absoluto do elemento do vetor resultante de uma operação vetorial for menor que **limiar** este valor é considerado nulo.

4.4 – ALOCAÇÃO DINÂMICA DE MEMÓRIA PARA O ESQUEMA RCOCL E RCOCV.

Os algoritmos para as operações matriciais com o esquema RCOCL necessitam alocar memória dinamicamente, já que a quantidade de elementos das matrizes esparsas depende da entrada de dados fornecida pelo usuário (ou sistema) e do resultado do processamento (multiplicação, soma, transposição, etc.). A linguagem de implementação deve possuir operadores de alocação de memória dinâmica. Em C++, por exemplo, **new** e **delete** são os operadores de alocação de memória. O operador **new** recebe como argumento o tipo do objeto que está sendo alocado dinamicamente e retorna um ponteiro para um objeto do mesmo tipo. O operador **new** executa o construtor do objeto e aloca memória. O operador **delete** recebe como argumento o ponteiro retornado por **new** e executa o destruidor do objeto para desalocar a memória alocada (a memória é retornada ao sistema de forma que a memória possa ser realocada no futuro).

O modo de armazenamento RCOCL tal como foi apresentado, ou seja, dispondo os dados seqüencialmente nos arranjos, sugere o uso de *array* na alocação de memória dinâmica. Um *array* consiste em um grupo de posições de memória consecutivas, todas de mesmo nome e mesmo tipo. Para fazer referência a uma posição particular ou elemento no *array*, especifica-se o nome do *array* e o número da posição (índice) do elemento no *array.*

A principal desvantagem apresentada pela alocação dinâmica para *array* é o fato da quantidade de memória alocada não diminuir ou aumentar quando se exclui ou insere, respectivamente, elementos no *array*. Então, quando os algoritmos de armazenamento recebem como argumentos matrizes esparsas cuja quantidade de elementos são conhecidas pode-se alocar dinamicamente para o tipo *array* a quantidade exata de memória para conter os valores dos elementos das matrizes esparsas. Entretanto, inserir ou excluir elementos não-nulos no espaço alocado é um processo que envolve uma nova alocação de memória e transferência de dados do antigo para o novo local de memória implicando na perda de desempenho. Já algumas operações matriciais (multiplicação, adição, etc.) que são executadas por algoritmos cujos armazenamentos já se encontram na forma do esquema RCOCL e, portanto com alocação de memória exata à quantidade de elementos, necessitam alocar memória temporária para conter o resultado de suas operações. Esta

memória temporária teria alocação ideal se a quantidade exata de elementos da matriz resposta fosse conhecida, entretanto, isto não é possível, pois operações como a adição e a multiplicação podem ter a sua matriz resposta com diversas quantidades de elementos. Logo, a solução consiste em determinar a quantidade máxima de elementos da matriz resposta e alocar memória para esta quantidade. O número máximo de elementos da matriz resposta é calculado de acordo com cada tipo de operação matricial.

O esquema RCOCV é extremamente semelhante ao esquema RCOCL logo todos os procedimentos para alocação dinâmica de memória também são semelhantes.

4.5 – DEFINIÇÃO DAS CLASSES SMATRIZ PARA MATRIZES ESPARSAS E SVETOR PARA VETORES ESPARSOS.

A partir do esquema de armazenamento RCOCL e de suas variáveis auxiliares já definidas tem-se a seguinte definição de classe.

```
class SMatriz {
public:
        //Construtor
        SMatriz(int QuantElementos );
        //Destrutor
        ~SMatriz();
private:        //Membros dados da Classe SMatriz usando o esquema RCOCL

        complex< double > *sa;      // Declara um ponteiro para o arranjo sa
        int *ija                    // Declara um ponteiro para o arranjo ija
        int *index;                 // Declara um ponteiro para o arranjo index

        int numlin;                 // Declara a variável auxiliar numlin
        int numcol;                 // Declara a variável auxiliar numcol
        double limiar;              // Declara a variável auxiliar limiar
};
```

Observe que os membros dados da Classe SMatriz são as variáveis de ponteiros para os arranjos **sa**, **ija** e **index**, e as variáveis auxiliares **numlin**, **numcol** e **limiar**.

Para o esquema de RCOCV e suas variáveis auxiliares tem-se:

```
class SVetor {
public:
        //Construtor
        SVetor(int QuantElementos );
        //Destrutor
        ~SVetor();
private:        //Membros de dado da Classe SVetor usando o esquema RCOCV
        complex< double > *sa;      // Declara um ponteiro para o arranjo sa
        int *ija                    // Declara um ponteiro para o arranjo ija
        int kov;                    //Declara um ponteiro para o arranjo kov
        int kfv;                    //Declara um ponteiro para o arranjo kfv

        int tamanho;                // Declara a variável auxiliar tamanho
        double limiar;              // Declara a variável auxiliar numlin
};
```

4.6 – OPERAÇÕES COM VETORES ESPARSOS PARA O ESQUEMA RCOCV

As duas principais operações com vetores esparsos no esquema RCOCV são a soma de vetores esparsos e o produto de um número complexo por um vetor esparso. A partir destes operadores pode-se efetuar as principais operações lineares.

4.6.1 – SOMA DE VETORES ESPARSOS

A Figura 4.3 mostra o fluxograma da operação de soma de dois objetos do tipo SVetor, **Va** e **Vb**. Para este algoritmo destaca-se o cálculo da quantidade de memória a ser alocada para conter o resultado da operação e a descrição do algoritmo.

4.6.1.1 – ALOCAÇÃO DE MEMÓRIA

Dados **m** objetos $\mathbf{V}1$, $\mathbf{V}2$, ... **Vm** da classe SVetor e com dimensões $\mathbf{V}1.\mathbf{tamanho} = \mathbf{V}2.\mathbf{tamanho} = ... = \mathbf{Vm}.\mathbf{tamanho} = \mathbf{n}$ tem-se que a quantidade máxima $\mathbf{Qs}_{\mathbf{máx}}$ de elementos não-nulos que o objeto **Vs** da classe SVetor dado por $\mathbf{Vs} = \mathbf{V}1 + \mathbf{V}2 + ... + \mathbf{Vm}$ pode ter é :

$$\mathbf{Qs_{máx}} = \begin{cases} \mathbf{Q}1 + \mathbf{Q}2 + ... + \mathbf{Qm} & \textbf{se} \quad \mathbf{Q}1 + \mathbf{Q}2 + ... + \mathbf{Qm} \leq \mathbf{n} \\ \mathbf{n} & \textbf{se} \quad \mathbf{Q}1 + \mathbf{Q}2 + ... + \mathbf{Qm} > \mathbf{n} \end{cases} \tag{4.1}$$

em que

$\mathbf{Q}1$ é quantidade de elementos de $\mathbf{V}1$ definida por $\mathbf{Q}1 = \mathbf{V}1.\mathbf{kfv} - \mathbf{V}1.\mathbf{kov} + 1$;

$\mathbf{Q}2$ é quantidade de elementos de $\mathbf{V}2$ definida por $\mathbf{Q}2 = \mathbf{V}2.\mathbf{kfv} - \mathbf{V}2.\mathbf{kov} + 1$;

...

$\mathbf{Qm}$ é quantidade de elementos de $\mathbf{Vm}$ definida por $\mathbf{Qm} = \mathbf{Vm}.\mathbf{kfv} - \mathbf{Vm}.\mathbf{kov} + 1$.

4.6.1.2 – DESCRIÇÃO DO ALGORITMO

Para se obter o objeto **Vs** deve-se ler os elementos **Va**.**sa**[**ka**] e **Va**.**ija**[**ka**], onde $\mathbf{Va}.\mathbf{kov} \leq \mathbf{ka} \leq \mathbf{Va}.\mathbf{kfv}$, em conjunto com os elementos **Vb**.**sa**[**kb**] e **Vb**.**ija**[**kb**], onde $\mathbf{Vb}.\mathbf{kov} \leq \mathbf{kb} \leq \mathbf{Vb}.\mathbf{kfv}$; e obter, através de condicionais, a soma dos elementos com mesmo índice $\mathbf{j} = \mathbf{Va}.\mathbf{ija}[\mathbf{ka}] = \mathbf{Vb}.\mathbf{ija}[\mathbf{kb}]$ de coluna. Estas condicionais são definidas de acordo com as possíveis situações descritas a seguir.

- Há pelo menos um elemento a ser adicionado no arranjo **Vs**.**sa**, ou seja, $(\mathbf{Va}.\mathbf{kov} \leq \mathbf{Va}.\mathbf{kfv})\mathbf{ou}(\mathbf{Vb}.\mathbf{kov} \leq \mathbf{Vb}.\mathbf{kfv})$.
    - Tanto o arranjo **Va**.**sa** quanto o **Vb**.**sa** apresentam elementos a serem operados.
        - A primeira e a segunda parcela estão presentes nos arranjos **Va**.**sa** e **Vb**.**sa**, respectivamente.
            - Se $\mathbf{Va}.\mathbf{sa}[\mathbf{ka}] + \mathbf{Vb}.\mathbf{sa}[\mathbf{kb}] \leq \mathbf{Vs.limiar}$ nenhum armazenamento é efetuado.
            - Se $\mathbf{Va}.\mathbf{sa}[\mathbf{ka}] + \mathbf{Vb}.\mathbf{sa}[\mathbf{kb}] > \mathbf{Vs.limiar}$ o armazenamento é efetuado. Então $\mathbf{Vs}.\mathbf{sa}[\mathbf{ks}] \leftarrow \mathbf{Va}.\mathbf{sa}[\mathbf{ka}] + \mathbf{Vb}.\mathbf{sa}[\mathbf{kb}]$ e $\mathbf{Vs}.\mathbf{ija}[\mathbf{ks}] \leftarrow \mathbf{Va}.\mathbf{ija}[\mathbf{ka}]$ ou $\mathbf{Vs}.\mathbf{ija}[\mathbf{ks}] \leftarrow \mathbf{Vb}.\mathbf{ija}[\mathbf{kb}]$.
        - A primeira parcela está presente no arranjo **Va**.**sa** enquanto que a segunda parcela não está presente no arranjo **Vb**.**sa**, pois a mesma é nula. Então $\mathbf{Vs}.\mathbf{sa}[\mathbf{ks}] \leftarrow \mathbf{Va}.\mathbf{sa}[\mathbf{ka}]$ e $\mathbf{Vs}.\mathbf{ija}[\mathbf{ks}] \leftarrow \mathbf{Va}.\mathbf{ija}[\mathbf{ka}]$;

        - A segunda parcela está presente no arranjo **Vb**.**sa** enquanto que a primeira parcela não está presente no arranjo **Va**.**sa**, pois a mesma é nula. Então **Vs**.**sa**[**ks**] ← **Vb**.**sa**[**kb**] e **Vs**.**ija**[**ks**] ← **Vb**.**ija**[**kb**];
    - O arranjo **Va**.**sa** apresenta elementos a serem operados enquanto que o arranjo **Vb**.**sa** não apresenta. Então **Vs**.**sa**[**ks**] ← **Va**.**sa**[**ka**] e **Vs**.**ija**[**ks**] ← **Va**.**ija**[**ka**];
    - O arranjo **Vb**.**sa** apresenta elementos a serem operados enquanto que o arranjo **Va**.**sa** não apresenta. Então **Vs**.**sa**[**ks**] ← **Vb**.**sa**[**kb**] e **Vs**.**ija**[**ks**] ← **Vb**.**ija**[**kb**];

- Não há elementos a serem adicionados no arranjo **Vs**.**sa** e, portanto, não há necessidade de operar a soma.

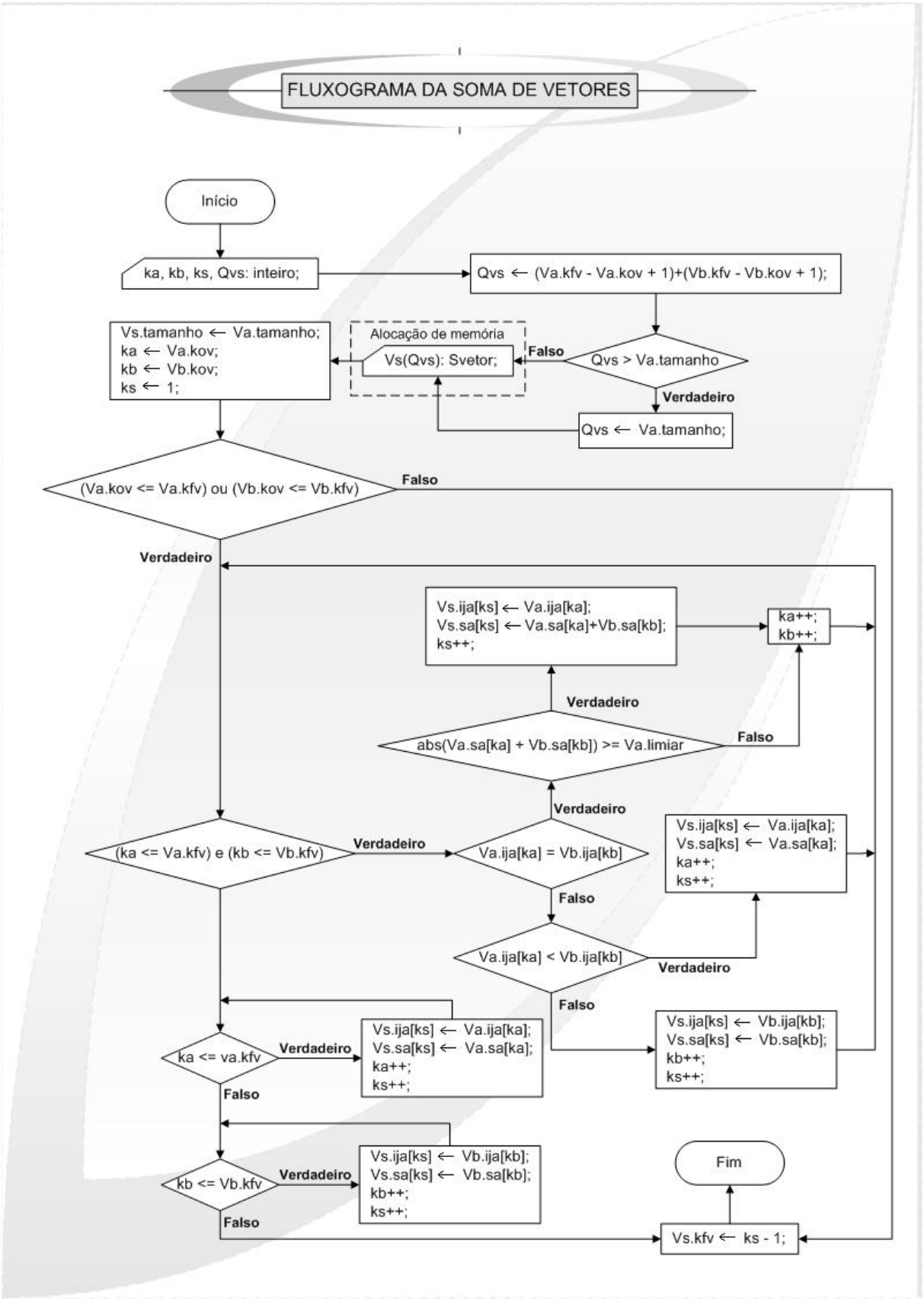

FIGURA 4.3 – Fluxograma do algoritmo da soma de dois objetos do tipo SVetor.

4.6.2 – PRODUTO DE UM NÚMERO COMPLEXO POR UM VETOR

A Figura 4.4 mostra o fluxograma da operação do produto de um objeto **x** do tipo complex<double> (número complexo) e um objeto **Va** do tipo SVetor. Para este algoritmo destaca-se o cálculo da quantidade de memória a ser alocada para conter o resultado da operação e a descrição do algoritmo.

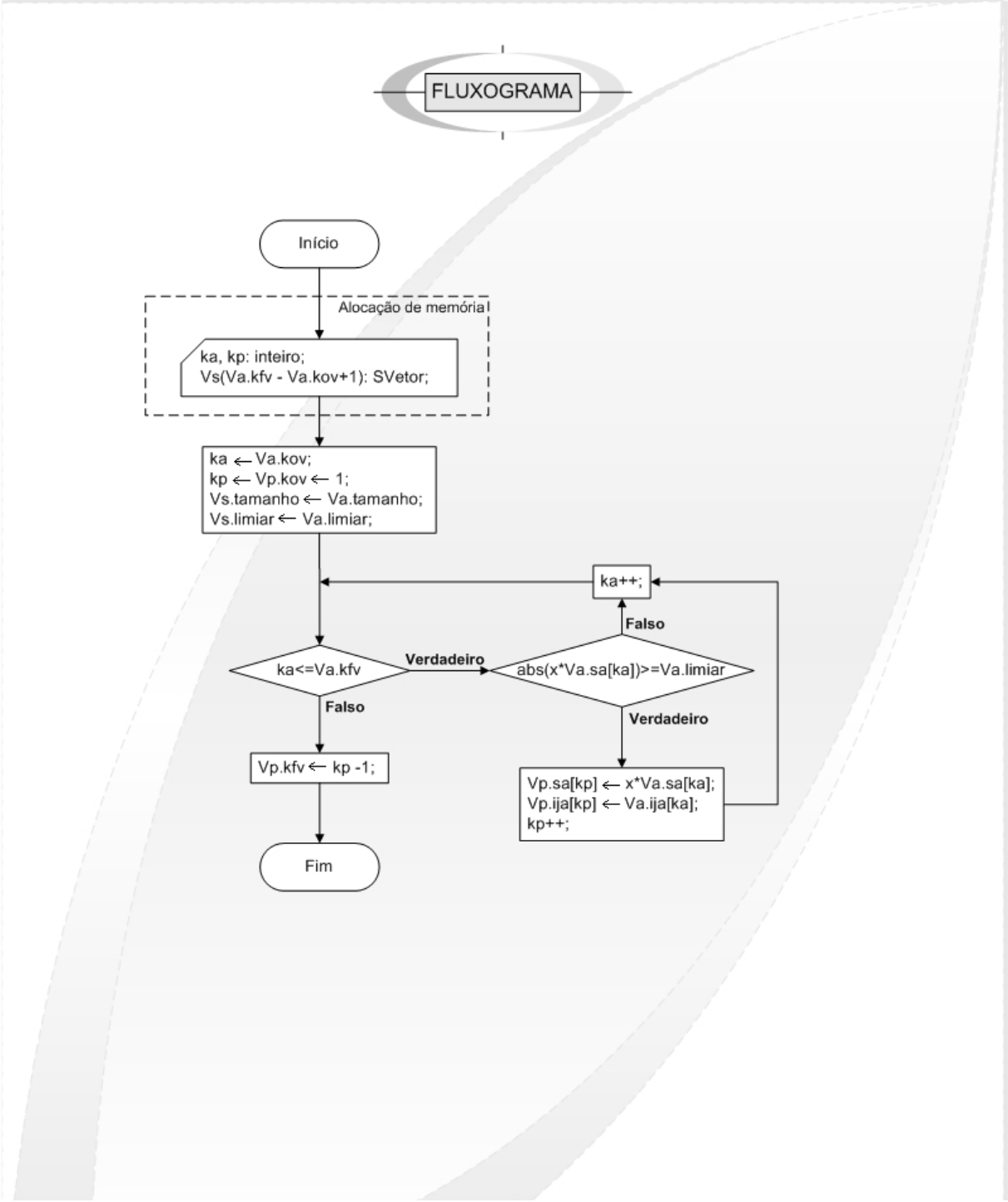

FIGURA 4.4 – Fluxograma da operação do produto de um objeto x do tipo complex<double> e um objeto Va do tipo SVetor.

4.6.2.1 – ALOCAÇÃO DE MEMÓRIA

Dados um objeto **x** do tipo complex<double> e um objeto **Va** do tipo SVetor, com $\mathbf{Va.tamanho} = \mathbf{n}$, tem-se que a quantidade máxima de elementos não-nulos $\mathbf{Qp_{máx}}$ que o vetor produto $\mathbf{Vp} = \mathbf{x} \cdot \mathbf{Va}$ pode ter é dada por:

$$\mathbf{Qp_{máx}} = \mathbf{Va.kfv} - \mathbf{Va.kov} + 1 \tag{4.2}$$

Demonstração:

$$\mathbf{Vp.sa[k]} = \begin{cases} 0 & \textbf{se } \mathbf{x} \cdot \mathbf{Va.sa[k]} \leq \mathbf{Va.limiar} \\ \mathbf{x} \cdot \mathbf{Va.sa[k]} & \textbf{se } \mathbf{x} \cdot \mathbf{Va.sa[k]} > \mathbf{Va.limiar} \end{cases}$$

$$\Rightarrow \quad \mathbf{Vp.kfv} - \mathbf{Vp.kov} + 1 \leq \mathbf{Va.kfv} - \mathbf{Va.kov} + 1$$

$$\therefore \quad \mathbf{Qp_{máx}} = \mathbf{Va.kfv} - \mathbf{Va.kov} + 1$$

Portanto **Vp** deve dispor de espaço de memória para $\mathbf{Qp_{máx}}$ elementos.

4.6.2.2 – DESCRIÇÃO DO ALGORITMO

O produto $\mathbf{Vp} = \mathbf{x} \cdot \mathbf{Va}$ é efetuado executando-se uma única varredura sobre os elementos de **Va.sa[ka]** e **Va.ija[ka]**. Durante a leitura de cada elemento dos arranjos é efetuada a operação $\mathbf{Vp.sa[kp]} \leftarrow \mathbf{x} \cdot \mathbf{Va.sa[ka]}$ e $\mathbf{Vp.ija[kp]} \leftarrow \mathbf{Va.ija[ka]}$ se $\mathbf{x} \cdot \mathbf{va.sa[ka]} \geq \mathbf{Va.limiar}$, conforme a Figura 4.4.

4.6.3 – DECLARAÇÃO DAS PRINCIPAIS FUNÇÕES DA CLASSE SVETOR

Neste tópico é descrito um exemplo de como são declaradas as funções responsáveis pela adição de vetores esparsos e produto de um número complexo por um vetor esparso. A declaração destas operações utiliza a sobrecarga de operadores que permite o programa cliente utilizar a notação matemática usuais de operações matriciais.

A definição da Classe SVetor dada a seguir mostra as declarações das funções membro e de uma função *friend*. Observe que a função "*operator**" declarada como *friend* faz uma chamada a função membro "*operator**".

```
class SVetor {

friend SVetor &operator*(complex<double>  z, SVetor &Va) { return Va*z; }

public:
        //Construtor
        SVetor(int QuantElementos );
        //Destrutor
        ~SVetor();
        //Funções membro da Classe
        SVetor &operator+(SVetor &Vb);
        SVetor &operator*(complex<double>  z);

private:        //Membros de dado da Classe SVetor usando o esquema RCOCV

        complex< double > *sa;      // Declara um ponteiro para o arranjo sa
        int *ija                    // Declara um ponteiro para o arranjo ija
        int kov;                    // Declara a variável kov
        int kfv;                    // Declara a variável kfv
        int tamanho;                // Declara a variável auxiliar tamanho
        double limiar;              // Declara a variável auxiliar numlin
};
```

4.7 – OPERAÇÕES COM MATRIZES ESPARSAS PARA O ESQUEMA RCOCL

Para que os algoritmos de operações com matrizes esparsas atinjam um bom desempenho é de fundamental importância que interajam cordialmente com o modo de armazenamento utilizado e façam uso adequado dos recursos disponíveis na linguagem de implementação. Para um entendimento melhor do que seria uma boa interação entre o algoritmo e o modo de armazenamento considere a multiplicação de duas matrizes esparsas de grandes dimensões e que o tipo de armazenamento seja o RCOCL. Para que o algoritmo apresente bom desempenho é preciso que o mesmo acesse o espaço de armazenamento por linha (e não por coluna), processe os dados por linha (e não por coluna), e armazene os resultados por linha (e não por coluna).

Das exigências mencionadas e pela indisponibilidade de algoritmos para operações matriciais esparsas com armazenamento RCOCL, foram desenvolvidas neste trabalho novas abordagens para o desenvolvimento destes algoritmos.

4.7.1 – TRANSPOSIÇÃO DE MATRIZES ESPARSAS

A Figura 4.7 mostra o fluxograma da operação de transposição de um objeto **A** do tipo SMatriz. Para este algoritmo destaca-se o cálculo da quantidade de memória a ser alocada para conter o resultado da operação e a descrição do algoritmo.

4.7.1.1 – ALOCAÇÃO DE MEMÓRIA

Considere o objeto **A** da classe SMatriz com $\mathbf{A.numlin} = \mathbf{m}$ e $\mathbf{A.numcol} = \mathbf{n}$ com $\mathbf{A.index}[\mathbf{m}+1]-1$ elementos. Tem-se que o objeto **T** da classe SMatriz contendo o resultado da transposição de **A** apresenta $\mathbf{A.numlin} = \mathbf{m}$, $\mathbf{A.numcol} = \mathbf{n}$ e $\mathbf{T.index}[\mathbf{n}+1]-1 = \mathbf{A.index}[\mathbf{m}+1]-1$ elementos. Isto porque o processo de transposição não exclui ou acrescenta elementos na matriz **T** e, portanto, **T** deve ter espaço em memória para $\mathbf{A.index}[\mathbf{m}+1]-1$ elementos.

4.7.1.2 – DESCRIÇÃO DO ALGORITMO

Transpor a matriz **A** de ordem $\mathbf{m} \times \mathbf{n}$ cujo armazenamento é o esquema RCOCL é o equivalente a imaginar a matriz **A** armazenada pelo esquema RCOCC (Representação Completa e Ordenada por Comprimento de Coluna). Para exemplificar a idéia exposta aqui, segue o exemplo da Figura 4.5, onde tem-se uma matriz **A** cujo seu armazenamento compacto se encontra na Tabela 4.5 para o esquema RCOCL. O armazenamento da transposta da matriz **A**, mostrada na Figura 4.6 é descrito na Tabela 4.6, onde equivaleria a armazenar os elementos da matriz **A** coluna por coluna.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} 1 & 2 \end{matrix} \\ \begin{bmatrix} 1 & 0 \\ 6 & 2 \\ 3 & 4 \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

FIGURA 4.5 – Matriz A para o armazenamento RCOCL

TABELA 4.5 – Armazenamento compacto para o esquema RCOCL da matriz A.

Índice k	**1**	**2**	**3**	**4**	**5**
ija[k]	1	1	2	1	2
sa[k]	1	6	2	3	4
Índice i	**1**	**2**	**3**	**4**	
Index[i] = k	1	2	4	6	

$$\mathbf{A^t} = \begin{matrix} 1 & 2 & 3 \end{matrix} \begin{bmatrix} 1 & 6 & 3 \\ 0 & 2 & 4 \end{bmatrix} \begin{matrix} 1 \\ 2 \end{matrix}$$

FIGURA 4.6 – Transposta da matriz A para o armazenamento RCOCL

TABELA 4.6 – Armazenamento compacto para o esquema RCOCL da transposta da matriz A.

Índice k	**1**	**2**	**3**	**4**	**5**
ija[k]	1	2	3	2	3
as[k]	1	6	3	2	4
Índice i	**1**	**2**	**3**		
Index[i] = k	1	4	6		

A diferença deste tipo de armazenamento é que os arranjos de **ija** e **sa** apresentam seus elementos ordenados por coluna e dentro de cada coluna, ordenados por linha, e os valores inteiros no arranjo **index** passam a indicar os índices correspondentes ao início das informações em **ija** e **sa** referentes às colunas da matriz esparsa. Os valores das variáveis **numlin** e **numcol** são permutados, enquanto o valor da variável limiar é preservado.

Com esta idéia em mente, a determinação do objeto **T** do tipo SMatriz contendo a transposta do objeto **A** do tipo SMatriz consiste primeiramente nas seguintes atribuições:

T.**numlin** ← **A**.**numcol**
T.**numcol** ← **A**.**numlin**
T.**limiar** ← **A**.**limiar**

Ao se instanciar o objeto **T** do tipo SMatriz, **T**.**index** terá todos os seus valores definidos como 1, ou seja, $\mathbf{T.index[k]} = 1$, onde $\mathbf{k} \in \{1,2,...,\mathbf{T.numlin}+1\}$. Isto porque a matriz em seu primeiro estado não contém nenhum elemento. Para a determinação dos valores de **T**.**index** necessita-se, previamente, saber a quantidade de elementos contidos

em cada coluna de **A**. A contagem de elementos de cada coluna é realizada pela expressão $\mathbf{T.index}[\mathbf{A.ija}[\mathbf{ka}]+1]++$, ou seja, $\mathbf{T.index}[\mathbf{A.ija}[\mathbf{ka}]+1] \leftarrow \mathbf{T.index}[\mathbf{A.ija}[\mathbf{ka}]+1]+1$, executada durante um laço na variável **ka** sobre o arranjo **A.ija**, onde $\mathbf{ka} \in \{1,2,3,...,\mathbf{A.index}[\mathbf{A.numlin}+1]-1\}$. Feito isto, a quantidades de elementos de cada coluna $\mathbf{i}-1$ é dada por $\mathbf{T.index}[\mathbf{i}]-1$, onde $\mathbf{i} \in \{2,3,...,\mathbf{T.numlin}+1\}$. Após isto, os elementos definitivos $\mathbf{T.index}[\mathbf{k}]$ são calculados pela expressão $\mathbf{T.index}[\mathbf{i}] \leftarrow \mathbf{T.inde}[\mathbf{i}-1]+\mathbf{T.index}[\mathbf{i}]-1$, durante a execução de um laço na variável **i** sobre **T.index**. Em outras palavras, durante a execução de um laço na variável **i** sobre **T.index**, cada elemento de $\mathbf{T.index}[\mathbf{i}]$**,** a partir do segundo, é igual ao anterior mais a quantidades de elementos da coluna $\mathbf{i}-1$ de **A**. Observe que para obter-se o arranjo **T.index** usa-se apenas dois laços separados. O número de repetições do primeiro laço é **A**.**index**[**A**.**numlin** + 1] − 1, e do segundo, **A.numcol**.

A partir do arranjo **T.index** armazena-se os elementos em **T.ija** e **T.sa**. As variáveis **T**.**index**[**i** − 1], onde **i-1** é a linha da matriz **T**, por fornecerem os índices de **T.ija** e **T.sa** referentes a posição das colunas da matriz **A** em **T.ija** e **T.sa**. Estas variáveis podem ser usadas como contadores de posição na inserção dos elementos de **A.ija** e **A.sa** para **T.ija** e **T.sa**, respectivamente. Isto é feito durante uma varredura simultânea nos arranjos **A.ija** e **A.sa**. Como os valores de **T**.**index**[**i** − 1] não podem ser alterados define-se o arranjo **pos** cujas características são iguais as do **T.index**. As atribuições dos elementos do **pos** é feita simultaneamente aos elementos de **T.index**, pela atribuição **pos**[**i**] ← **T**.**index**[**i**]. Os valores de **T.ija** e **T.sa** são obtidos por dois laços em cascata. O primeiro laço é executado na variável **j** de 1 a **A**.**numlin** e tem como finalidade determinar o índice **ka** e **kaf** que identificam, respectivamente, as posições inicial e final da linha **j** de **A**. O segundo laço varia de **ka** a **kaf** e executa três operações inerentes a transposição:

1. $\mathbf{kt} \leftarrow \mathbf{pos}[\mathbf{A.ija}[\mathbf{ka}]]++$: **A**.**ija**[**ka**] fornece a coluna do elemento **A**.**sa**[**ka**] da linha **j** da matriz **A**. $\mathbf{kt} \leftarrow \mathbf{pos}[\mathbf{A.ija}[\mathbf{ka}]]$ identifica o índice **kt** da posição do elemento **A**.**sa**[**ka**] e do índice **j** nos arranjos **T.sa** e **T.ija**, respectivamente, para uma linha **A**.**ija**[**ka**] da matriz **T**. **pos**[**A.ija**[**ka**]] + + incrementa $\mathbf{pos}[\mathbf{A.ija}[\mathbf{ka}]]$ de uma unidade para uma futura operação;
2. **T**.**sa**[**kt**] ← **A**.**sa**[**ka**]: Atribui **A**.**sa**[**ka**] a **T**.**sa**[**kt**];
3. **T.ija**[**kt**] ← **j**: Atribui **j** a **T.ija**[**kt**];

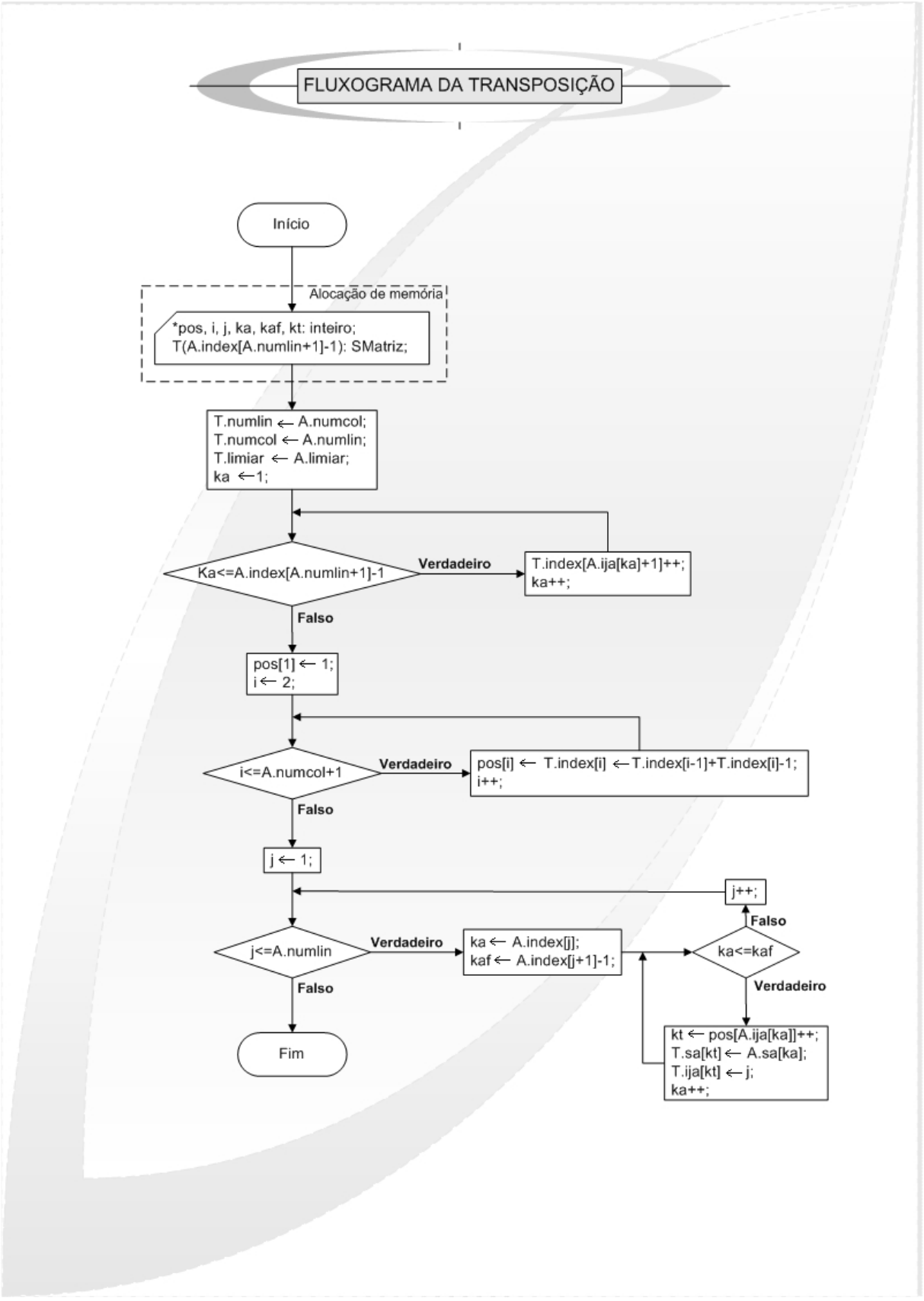

FIGURA 4.7 – Fluxograma da operação de transposição de um objeto A do tipo SMatriz.

4.7.1.3 – DESEMPENHO DO ALGORITMO DE TRANSPOSIÇÃO

A Figura 4.8 mostra a comparação de desempenho entre a operação de transposição com o esquema RCOCL e a operação transposição com o esquema convencional para matrizes de ordem quadrada com grau de esparsidade 96%. Como pode ser observado o uso do esquema RCOCL em conjunto com a linguagem C++ proporciona um bom desempenho. Para um grau de esparsidade menor a necessidade do uso do esquema RCOCL torna-se mais acentuada.

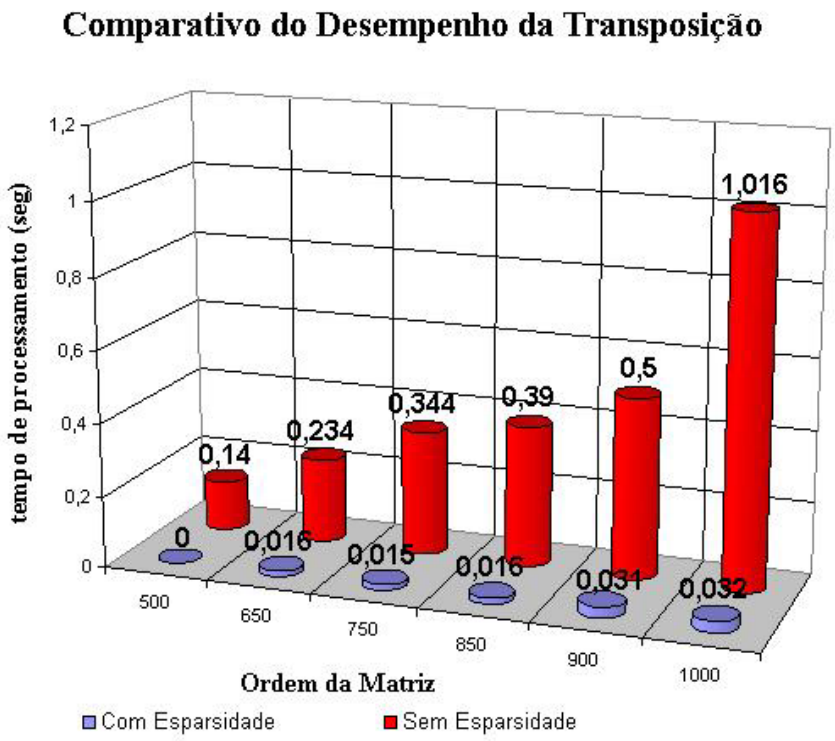

FIGURA 4.8 – Comparação do desempenho entre a operação de transposição com o esquema RCOCL e a operação transposição com o esquema convencional para matrizes de ordem quadrada com grau de esparsidade 96%.

4.7.2 – MULTIPLICAÇÃO DE MATRIZES ESPARSAS

A Figura 4.9 mostra o fluxograma da operação de multiplicação de **A** e **B** do tipo SMatriz. Para este algoritmo destaca-se a formulação matemática do processo, o cálculo da quantidade de memória a ser alocada para conter o resultado da operação e a descrição do algoritmo.

4.7.2.1 – FORMULAÇÃO MATEMÁTICA

Preliminarmente será mostrada a interpretação usual da operação de multiplicação e posteriormente o modo de interpretação da mesma operação por meio de leitura e armazenamento por linha.

Dada a matriz **A**, de ordem $\mathbf{n}\times\mathbf{p}$, e a matriz **B**, de ordem $\mathbf{p}\times\mathbf{m}$, define-se a matriz produto **C** de ordem $n\times m$ como:

$$\mathbf{A}\cdot\mathbf{B}=\mathbf{C} \Leftrightarrow \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{np} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{bmatrix}$$

onde

$$c_{ij}=\sum_{k=1}^{p} a_{ik}\cdot b_{kj} \tag{4.3}$$

O elemento c_{ij} é, usualmente, interpretado como o produto da linha **i** de **A** pela coluna **j** de **B**. Esta interpretação não é satisfatória para a operação com o esquema RCOCL, visto que a operação de leitura da coluna **j** de **B** não é compatível com o armazenamento. Daí a necessidade de uma outra interpretação.

Será definida a linha **i** de **C** como

$$\mathbf{C_i}=\left[c_{i1}\ c_{i2}\ \cdots c_{im}\right] \tag{4.4}$$

e a linha **k** de **B** como

$$\mathbf{B_k}=\left[b_{k1}\ b_{k2}\cdots b_{km}\right] \tag{4.5}$$

Aplicando (4.3) em (4.4), tem-se:

$$\mathbf{C_i}=\left[\sum_{k=1}^{p} a_{ik}\cdot b_{k1} \quad \sum_{k=1}^{p} a_{ik}\cdot b_{k2} \quad \cdots \quad \sum_{k=1}^{p} a_{ik}\cdot b_{km}\right]$$

$$\mathbf{C_i} = \sum_{\mathbf{k}=1}^{\mathbf{p}} \left[\mathbf{a_{ik}} \cdot \mathbf{b_{k1}} \quad \mathbf{a_{ik}} \cdot \mathbf{b_{k2}} \quad \cdots \quad \mathbf{a_{ik}} \cdot \mathbf{b_{km}} \right]$$

$$\mathbf{C_i} = \sum_{\mathbf{k}=1}^{\mathbf{p}} \mathbf{a_{ik}} \cdot \left[\mathbf{b_{k1}} \quad \mathbf{b_{k2}} \quad \cdots \quad \mathbf{b_{km}} \right]$$

Substituindo, $\left[\mathbf{b_{k1}} \quad \mathbf{b_{k2}} \quad \cdots \quad \mathbf{b_{km}} \right]$ por (4.5) tem-se:

$$\mathbf{C_i} = \sum_{\mathbf{k}=1}^{\mathbf{p}} \mathbf{a_{ik}} \cdot \mathbf{B_k} \tag{4.6}$$

Nesta expressão a linha **i** de **C**, $\mathbf{C_i}$, é a somatória do produto das linhas **k** de **B**, $\mathbf{B_k}$, pelos elementos da linha **i** de **A**, $\mathbf{a_{ik}}$. Observe que as operações e o resultado são por linha e, portanto, ótimas para o esquema RCOCL.

Definindo **C[i]** e **B[k]** como objetos da classe SVetores contendo os elementos não-nulos, respectivamente, de $\mathbf{C_i}$ e $\mathbf{B_k}$ e considerando **A** como objeto da classe SMatriz tem-se:

$$\begin{cases} \mathbf{a_{ik}} = \mathbf{A.sa[ka]} \\ \mathbf{k} = \mathbf{A.ija[ka]} \end{cases} \tag{4.7}$$

onde $\mathbf{A.index[i]} \le \mathbf{ka} \le \mathbf{A.index[i+1]} - 1$

Substituindo (4.7) em (4.6) tem-se:

$$\mathbf{C[i]} = \sum_{\mathbf{ka}=\mathbf{A.index[i]}}^{\mathbf{A.index[i+1]}-1} \mathbf{A.sa[ka]} \cdot \mathbf{B[A.ija[ka]]} \tag{4.8}$$

Esta expressão é a essência do desempenho e legibilidade do algoritmo de multiplicação de matrizes esparsas com esquema RCOCL.

4.7.2.2 – ALOCAÇÃO DE MEMÓRIA

Dados **A**, **B** e **C** objetos do tipo SMatriz e considerando que $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ tem-se que a quantidade máxima de elementos não-nulos $\mathbf{Q_{Cmáx}}$ que **C** pode ter é dada por:

$$\mathbf{Q_{Cmáx}} = \sum_{\mathbf{i}=1}^{\mathbf{A.numlin}} \begin{cases} \mathbf{Q(i)} & \textbf{se} \quad \mathbf{Q(i)} \le \mathbf{B.numcol} \\ \mathbf{B.numcol} & \textbf{se} \quad \mathbf{Q(i)} > \mathbf{B.numcol} \end{cases} \tag{4.9}$$

onde $\mathbf{Q(i)} = \sum_{\mathbf{ka}=\mathbf{A.index[i]}}^{\mathbf{A.index[i+1]}-1} \{\mathbf{B.index}[\mathbf{A.ija}[\mathbf{ka}]+1] - \mathbf{B.inde}[\mathbf{A.ija}[\mathbf{ka}]]\}$

Demonstração

Na equação (4.8) tem-se que $\mathbf{C}[\mathbf{i}]$ é um somatórios de objetos do tipo SVetor e, portanto, pode-se aplicar a equação (4.1) para cada linha **i** de **A** e obter a quantidade máxima $\mathbf{Qi_{máx}}$ para cada operação da linha **i** de **A**.

$$\mathbf{Qi_{máx}} = \begin{cases} \mathbf{Q(i)} & \textbf{se} \quad \mathbf{Q(i)} \le \mathbf{B.numcol} \\ \mathbf{B.numcol} & \textbf{se} \quad \mathbf{Q(i)} > \mathbf{B.numcol} \end{cases}$$

Contabilizando todas as linhas de **A** tem-se:

$$\mathbf{Q_{Cmáx}} = \sum_{\mathbf{i}=1}^{\mathbf{A.numlin}} \begin{cases} \mathbf{Q(i)} & \textbf{se} \quad \mathbf{Q(i)} \le \mathbf{B.numcol} \\ \mathbf{B.numcol} & \textbf{se} \quad \mathbf{Q(i)} > \mathbf{B.numcol} \end{cases} \tag{4.10}$$

4.7.2.3 – DESCRIÇÃO DO ALGORITMO DE MULTIPLICAÇÃO

O algoritmo de multiplicação da Figura 4.9 é dividido em duas etapas. Na primeira é efetuado o cálculo da máxima quantidade de elementos da matriz produto **C** e, consecutivamente, a alocação de memória para essa quantidade de elementos. O cálculo de memória é efetuado por um laço independente e dois laços em cascata para atender a expressão (4.9). O laço independente calcula a quantidade de elementos em cada linha. O primeiro e o segundo laço em cascata calculam a somatória externa e interna, respectivamente, da expressão (4.9). A condicional dentro do segundo laço em cascata verifica qual dos dois argumentos da somatória externa da equação (4.9) será aceito. Na segunda etapa é efetuada a multiplicação com o armazenamento RCOCL. Para tanto são executados dois laços em cascata para atender a expressão (4.8). O primeiro laço é executado de 1 a $\mathbf{A}.\mathbf{numlin}$ e tem como finalidade determinar os índices $\mathbf{ka} = \mathbf{A}.\mathbf{index}[\mathbf{i}]$ e $\mathbf{kaf} = \mathbf{A}.\mathbf{index}[\mathbf{A}.\mathbf{numlin} + 1] - 1$ que identificam, respectivamente, as posições inicial e final da linha **i** de **A**. O segundo laço varre **A.sa** e **A.ija** na variável **ka**, com $\mathbf{A.index[i]} \le \mathbf{ka} \le \mathbf{A.index[i} + 1\mathbf{]} - 1$, para obter $\mathbf{vo} \leftarrow \mathbf{vo} + \mathbf{A}.\mathbf{sa}[\mathbf{ka}] \cdot \mathbf{B}\left[\mathbf{A}.\mathbf{ija}[\mathbf{ka}]\right]$ a cada interação.

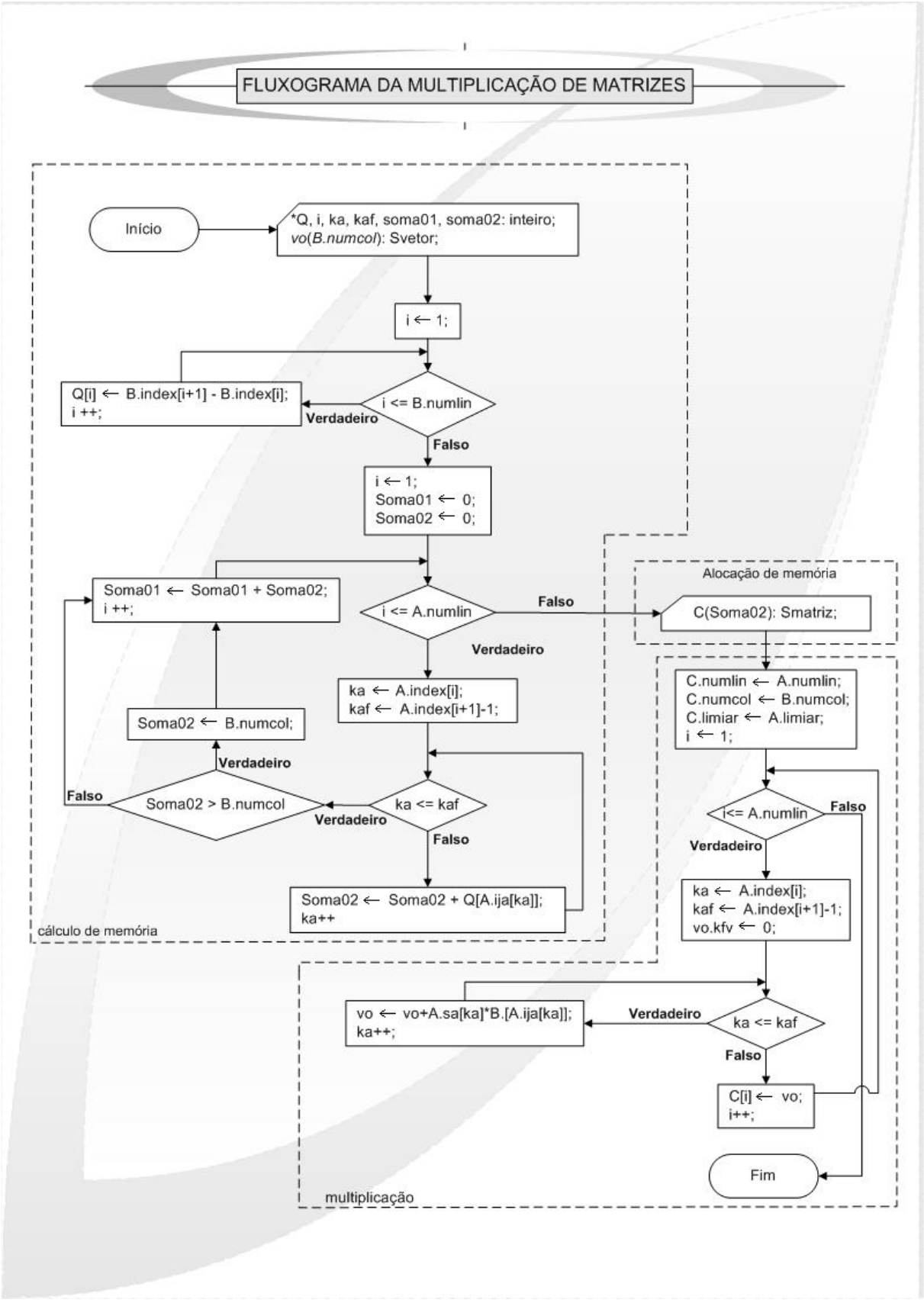

FIGURA 4.9 – Fluxograma da operação multiplicação dos objetos A e B do tipo SMatriz.

4.7.2.4 – DESEMPENHO DO ALGORITMO DE MULTIPLICAÇÃO

A Figura 4.10 mostra a comparação de desempenho entre a operação de multiplicação com o esquema RCOCL e a operação de multiplicação com o esquema convencional para matrizes de ordem quadrada com grau de esparsidade 96%. Como pode ser observado o uso do esquema RCOCL em conjunto com a linguagem C++ proporciona um bom desempenho. Este desempenho se torna mais visível quanto maior for a ordem das matrizes operadas. É evidente que para um grau de esparsidade menor a necessidade do uso do esquema RCOCL torna-se mais acentuada.

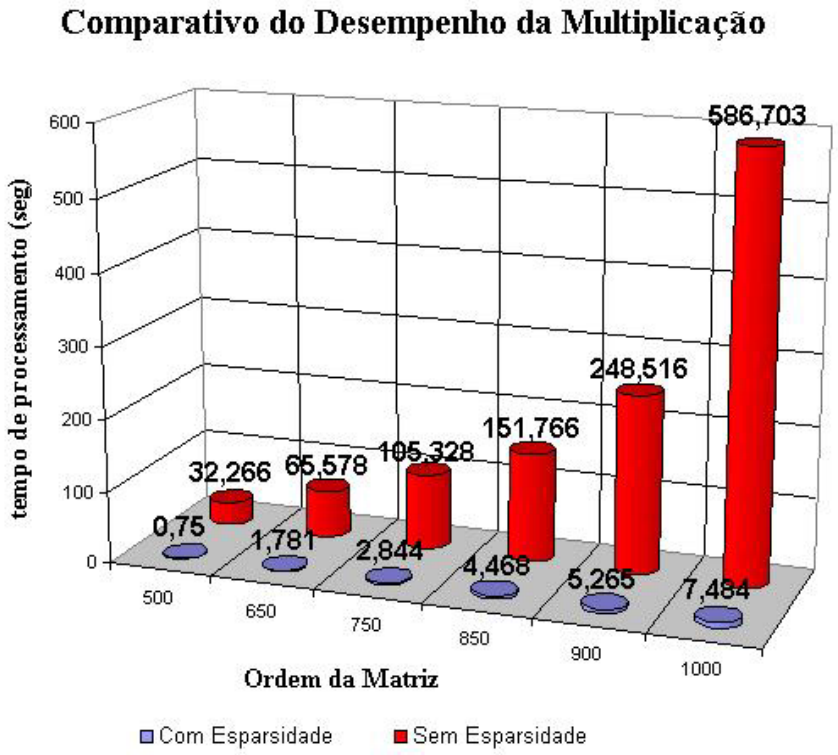

FIGURA 4.10 – Comparação do desempenho entre a operação de multiplicação com o esquema RCOCL e a operação de multiplicação com o esquema convencional para matrizes de ordem quadrada com grau de esparsidade 96%.

4.7.3 – ADIÇÃO DE MATRIZES ESPARSAS

A Figura 4.11 mostra o fluxograma da operação de soma dos objetos **A** e **B** do tipo SMatriz. Para este algoritmo destaca-se o cálculo da quantidade de memória a ser alocada para conter o resultado da operação e a descrição do algoritmo.

4.7.3.1 – ALOCAÇÃO DE MEMÓRIA

Definindo **A**, **B** e **C** como objetos do tipo SMatriz, com as mesmas dimensões e considerando que $\mathbf{C} = \mathbf{A} + \mathbf{B}$, tem-se que a quantidade máxima de elementos não-nulos $\mathbf{Q_{Cmáx}}$ que **C** pode ter é dada por:

$$\mathbf{Qs_{máx}} = \begin{cases} \mathbf{Qa} + \mathbf{Qb} & \textbf{se} \quad \mathbf{Qa} + \mathbf{Qb} \le \mathbf{A.numlin} \cdot \mathbf{A.numcol} \\ \mathbf{A.numlin} \cdot \mathbf{A.numcol} & \textbf{se} \quad \mathbf{Qa} + \mathbf{Qb} > \mathbf{A.numlin} \cdot \mathbf{A.numco} \end{cases} \tag{4.11}$$

onde

Qa é quantidade de elementos de **A** definida por $\mathbf{Qa} = \mathbf{A.index}\left[\mathbf{A.numlin} + 1\right] - 1$;

Qb é quantidade de elementos de **B** definida por $\mathbf{Qb} = \mathbf{B.index}\left[\mathbf{B.numlin} + 1\right] - 1$.

4.7.3.2 – DESCRIÇÃO DO ALGORITMO

O algoritmo da soma da Figura 4.11 consiste através de laços varrer as linhas de **A** e **B** de mesmo índice **i**; e obter, através de condicionais, a soma dos elementos com o mesmo índice **j** de coluna, ou seja, $\mathbf{c_{ij}} = \mathbf{a_{ij}} + \mathbf{b_{ij}}$.

Durante o acesso ao armazenamento das linhas de **A** e **B** podem ocorrer as seguintes situações:

- Há pelo menos um elemento não-nulo a ser adicionado a linha **i** de **C**.
    - Tanto a linha **i** de **A** quanto à linha **i** de **B apresentam** parcelas não nulas a serem operadas.
        - As parcelas , $\mathbf{a_{ij}}$ e $\mathbf{b_{ij}}$, estão presentes na linha **i** de **A** e de **B**, respectivamente.
            - Se $\mathbf{A.sa[ka]} + \mathbf{B.sa[kb]} \le \mathbf{C.limiar}$ nenhum armazenamento é efetuado.
            - Se $\mathbf{A.sa[ka]} + \mathbf{B.sa[kb]} > \mathbf{C.limiar}$ o armazenamento é efetuado. Então $\mathbf{C.sa[kc]} \leftarrow \mathbf{A.sa[ka]} + \mathbf{B.sa[kb]}$ e $\mathbf{C.ija[kc]} \leftarrow \mathbf{A.ija[ka]}$ ou $\mathbf{C.ija[kc]} \leftarrow \mathbf{B.ija[kb]}$.

- A parcela $\mathbf{a_{ij}}$ está presente na linha **i** de **A** enquanto que a parcela $\mathbf{b_{ij}}$ não está presente na linha **i** de **B**, pois a mesma é nula. Então $\mathbf{C.sa[kc]} \leftarrow \mathbf{A.sa[ka]}$ e $\mathbf{C.ija[kc]} \leftarrow \mathbf{A.ija[ka]}$.
- A parcela $\mathbf{b_{ij}}$ está presente na linha **i** de **B** enquanto que a parcela $\mathbf{a_{ij}}$ não está presente na linha **i** de **A**, pois a mesma é nula. Então $\mathbf{C.sa[kc]} \leftarrow \mathbf{B.sa[kb]}$ e $\mathbf{C.ija[kc]} \leftarrow \mathbf{B.ija[kb]}$.

- A linha **i** de **A apresenta** parcelas não nulas a serem operadas enquanto que a linha **i** de **B não apresenta**. Então $\mathbf{C.sa[kc]} \leftarrow \mathbf{A.sa[ka]}$ e $\mathbf{C.ija[kc]} \leftarrow \mathbf{A.ija[ka]}$.
- A linha **i** de **A não apresenta** parcelas não nulas a serem operadas enquanto que a linha **i** de **B apresenta.** Então $\mathbf{C.sa[kc]} \leftarrow \mathbf{B.sa[kb]}$ e $\mathbf{C.ija[kc]} \leftarrow \mathbf{B.ija[kb]}$.

- Não há elementos a serem adicionados a linha **i** de **C** e, portanto, não há necessidade de operar sobre estas linhas.

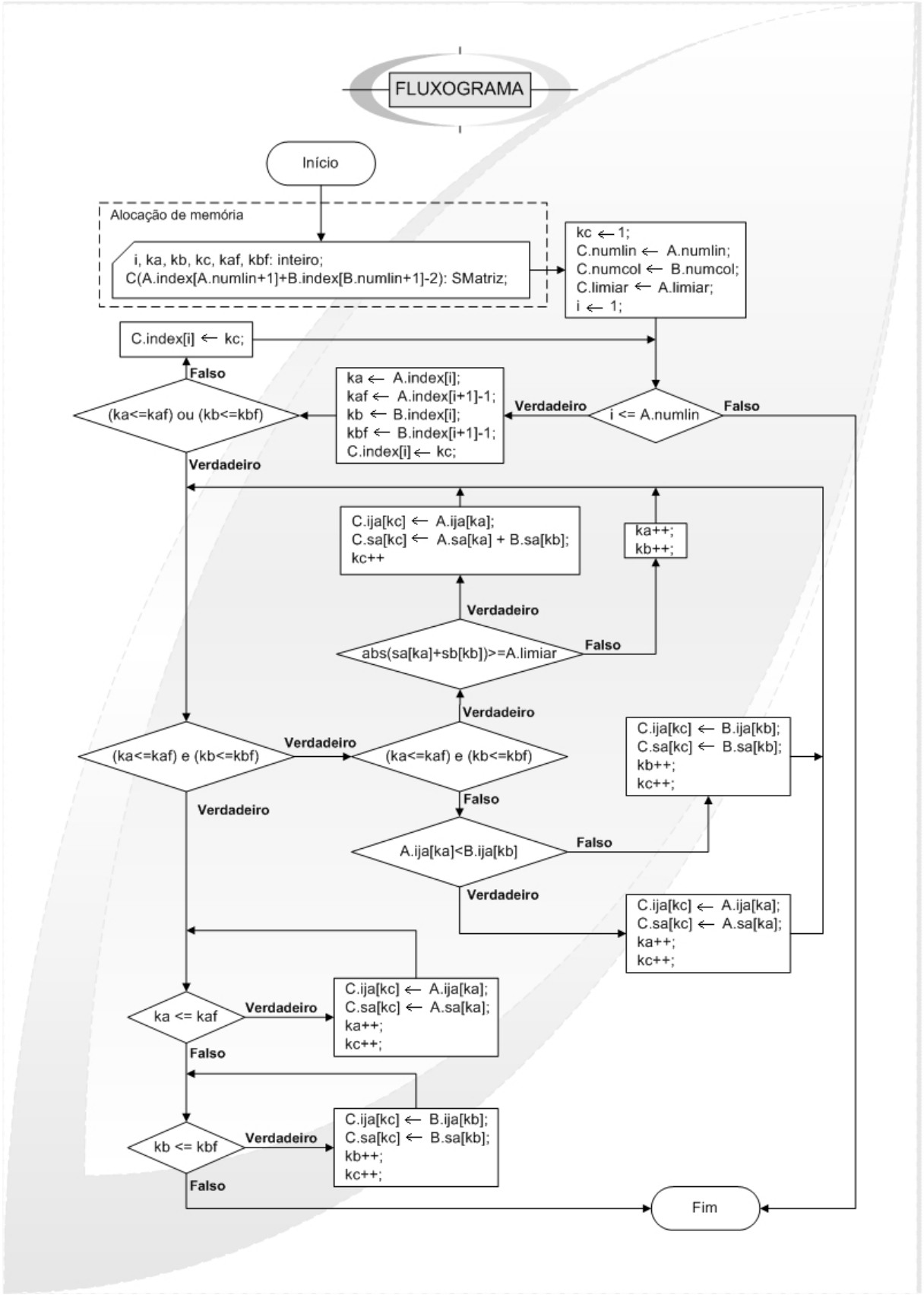

FIGURA 4.11 – Fluxograma da operação de soma dos objetos A e B do tipo SMatriz.

4.7.3.3 – DESEMPENHO DO ALGORITMO DE SOMA

A Figura 4.12 mostra a comparação de desempenho entre a operação de soma com o esquema RCOCL e a operação de soma com o esquema convencional para matrizes de ordem quadrada com grau de esparsidade 96%. Como pode ser observado o uso do esquema RCOCL em conjunto com a linguagem C++ proporciona um bom desempenho.

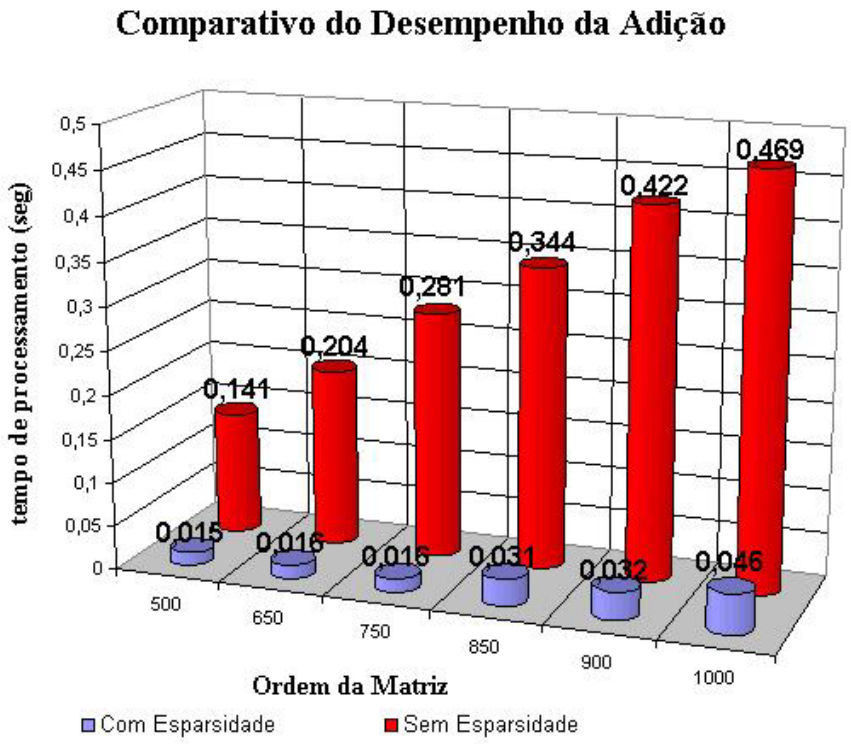

FIGURA 4.12 – Comparação do desempenho entre a operação de soma com o esquema RCOCL e a operação de soma com o esquema convencional para matrizes de ordem quadrada com grau de esparsidade 96%.

4.7.4 – SOLUÇÃO DE SISTEMAS LINEARES

As Figuras 4.13, 4.14, 4.15, 4.16, 4.17 e 4.18 mostram por partes o fluxograma da operação de solução de sistemas lineares dados um objeto **Y** do tipo SMatriz e um objeto **b** do tipo **Vetor**. Para este algoritmo destaca-se o cálculo da quantidade de memória a ser alocada para conter os resultados das operações inerentes ao algoritmo, a ordenação de **Y**, a eliminação de Gauss e a extração do vetor resposta **x**.

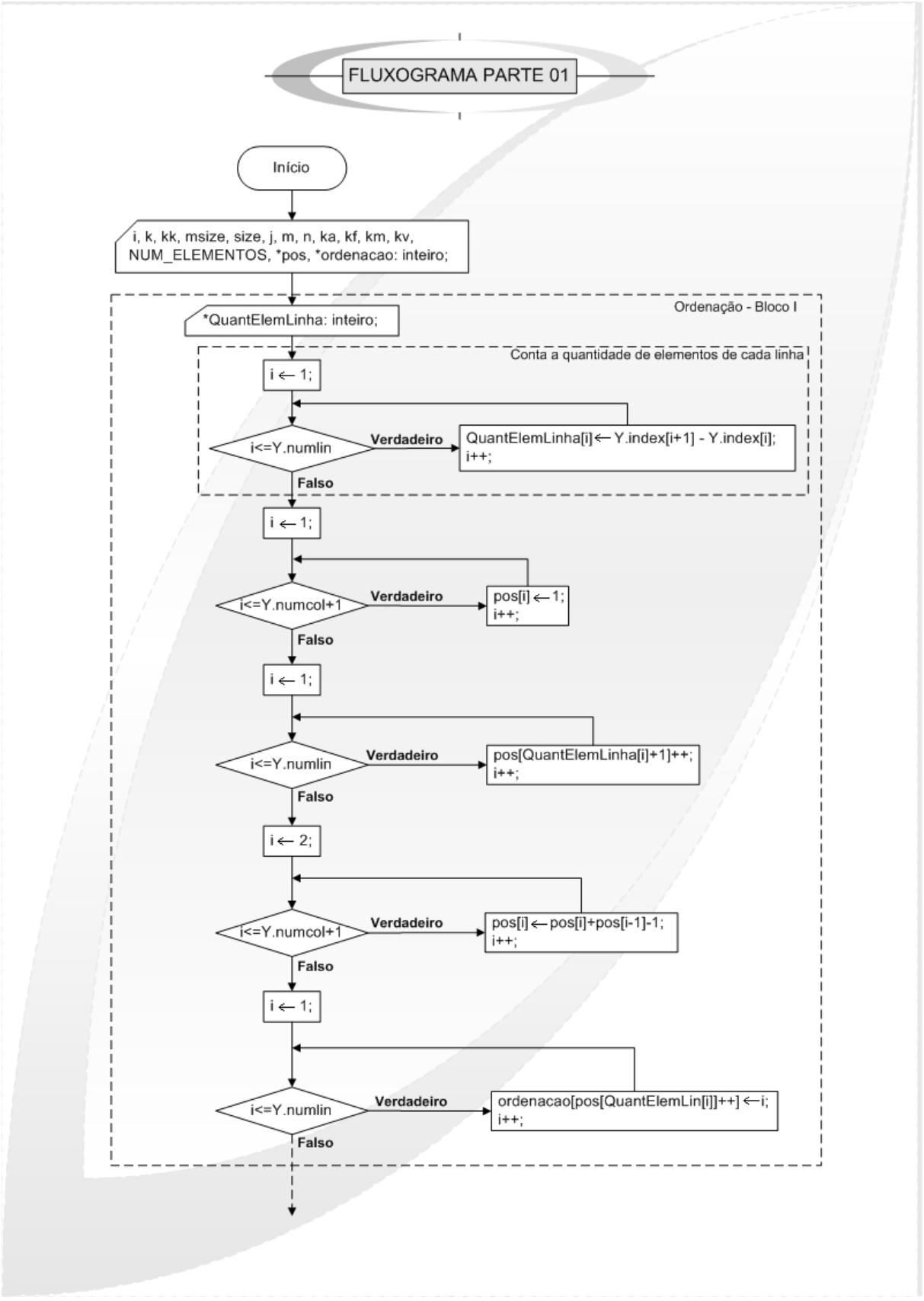

FIGURA 4.13 – Parte 01 do algoritmo de solução de sistemas lineares.

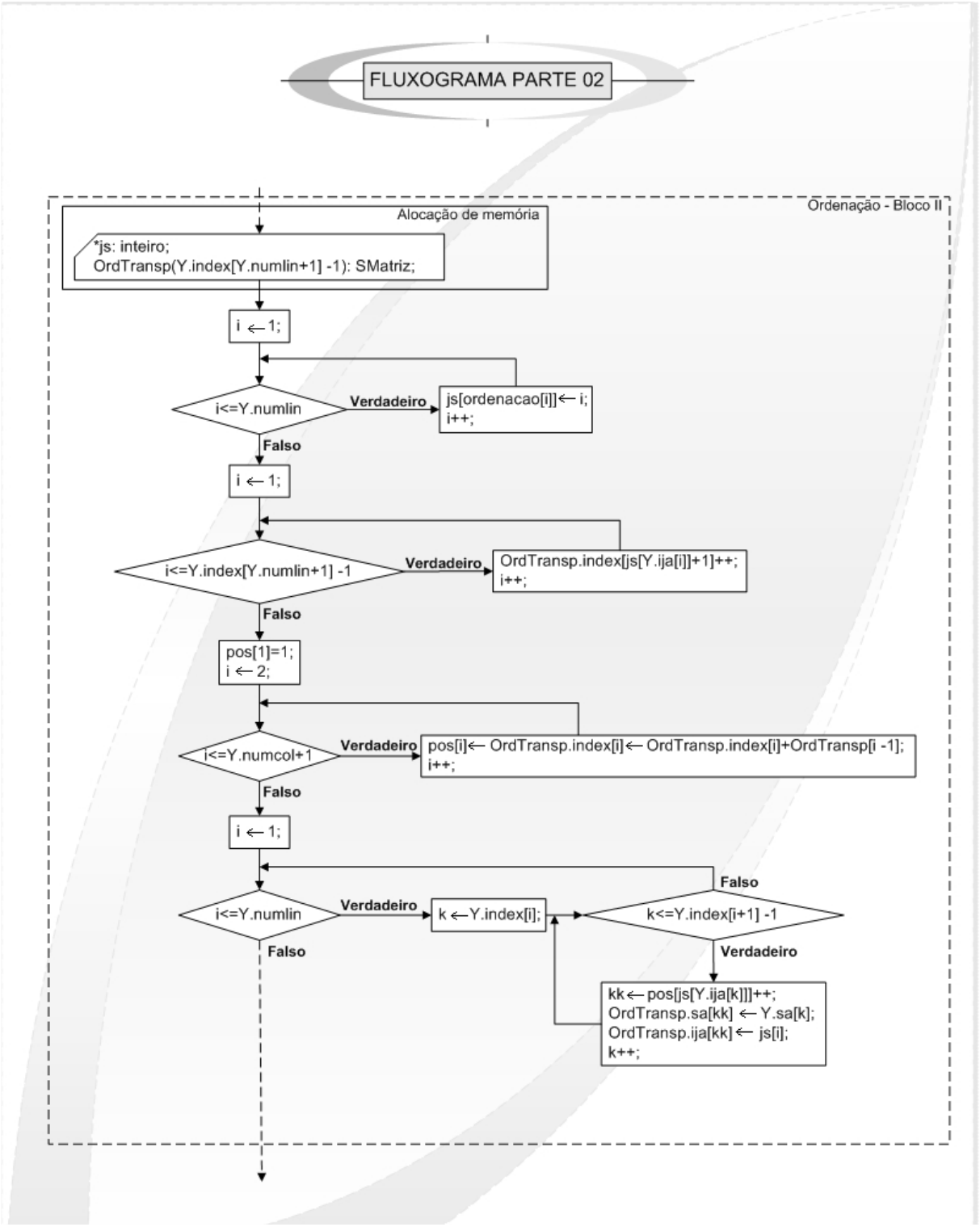

FIGURA 4.14 – Parte do algoritmo de solução de sistemas lineares.

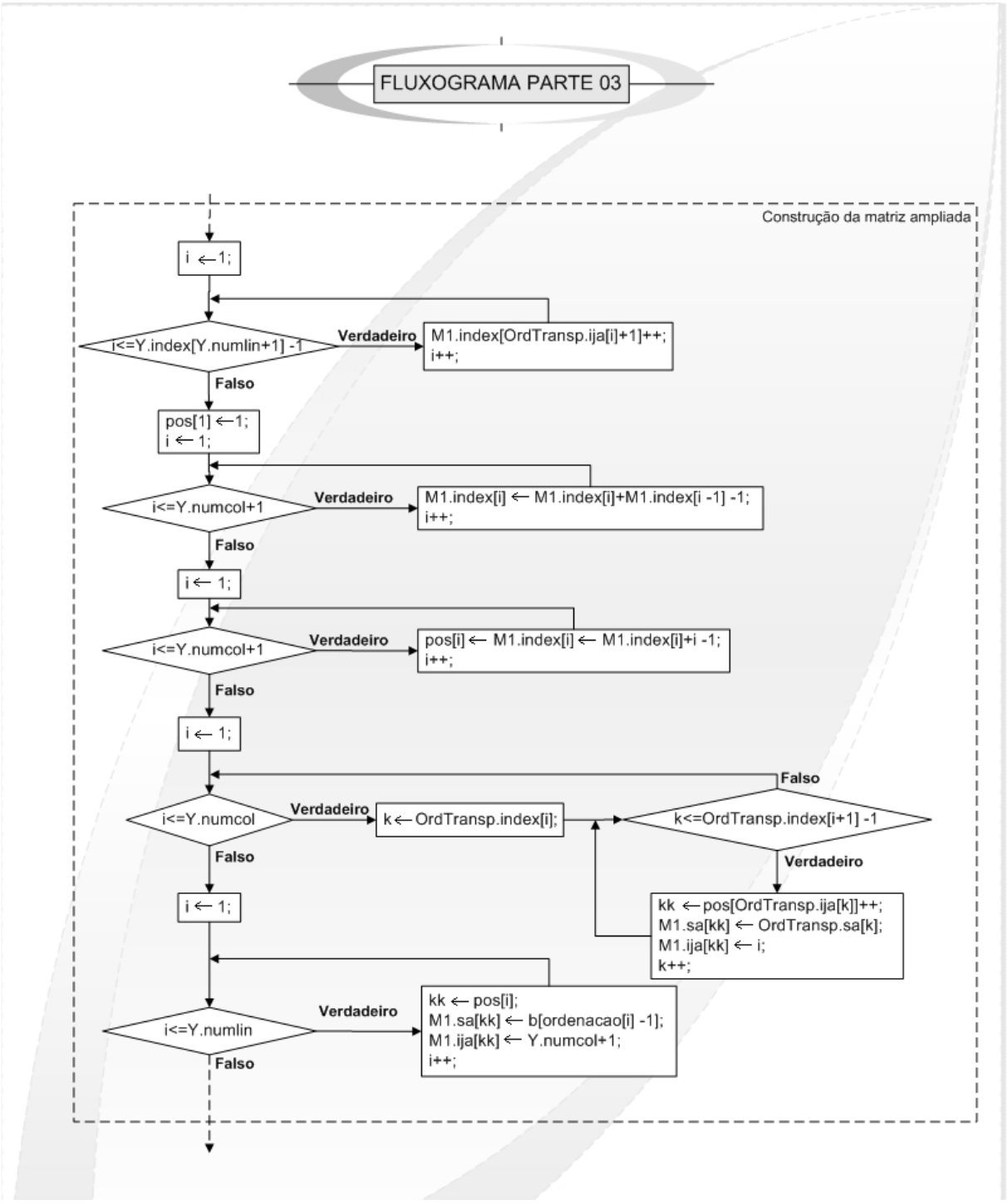

FIGURA 4.15 – Parte 03 do algoritmo de solução de sistemas.

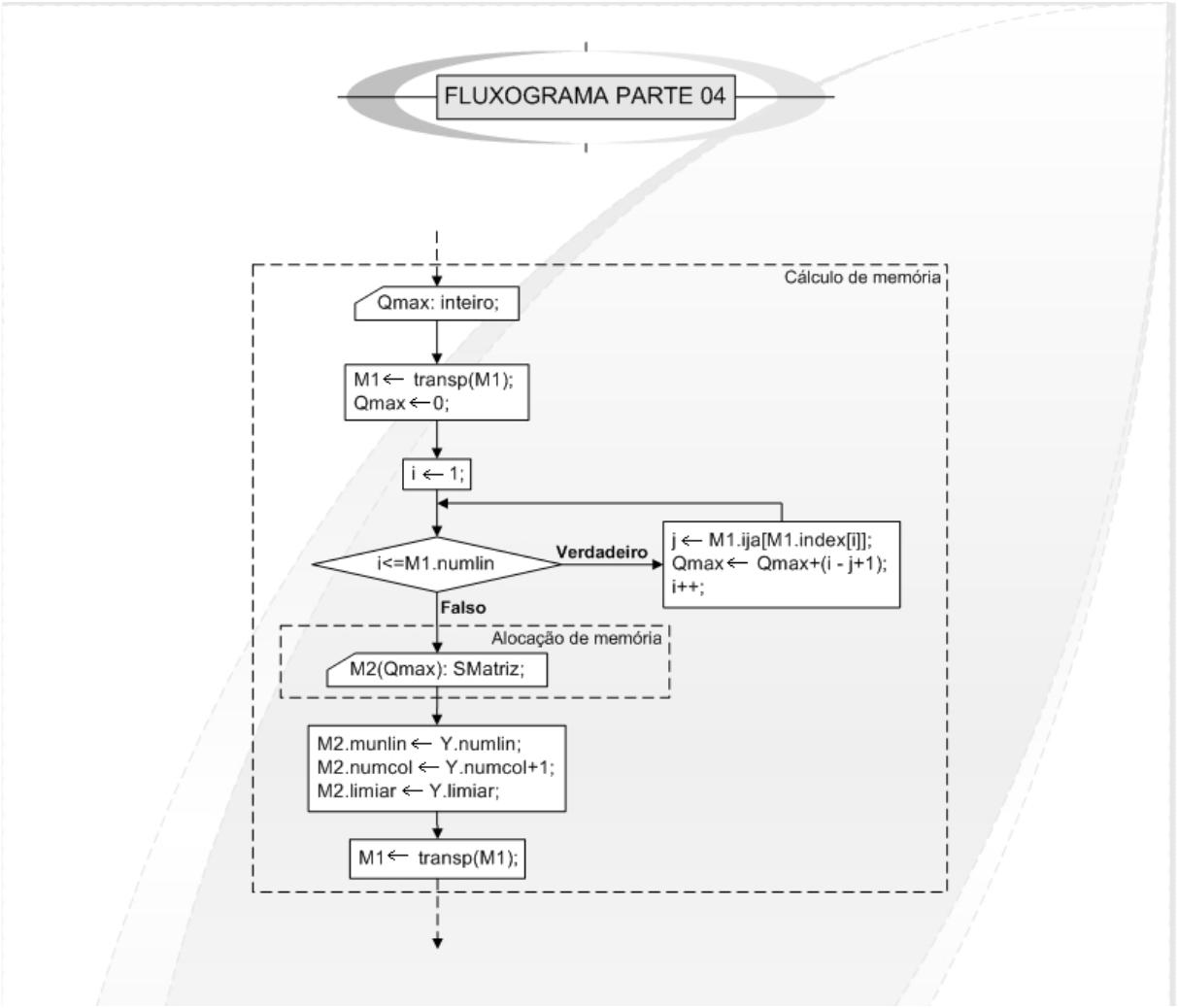

FIGURA 4.16 – Parte 04 do algoritmo de solução de sistemas.

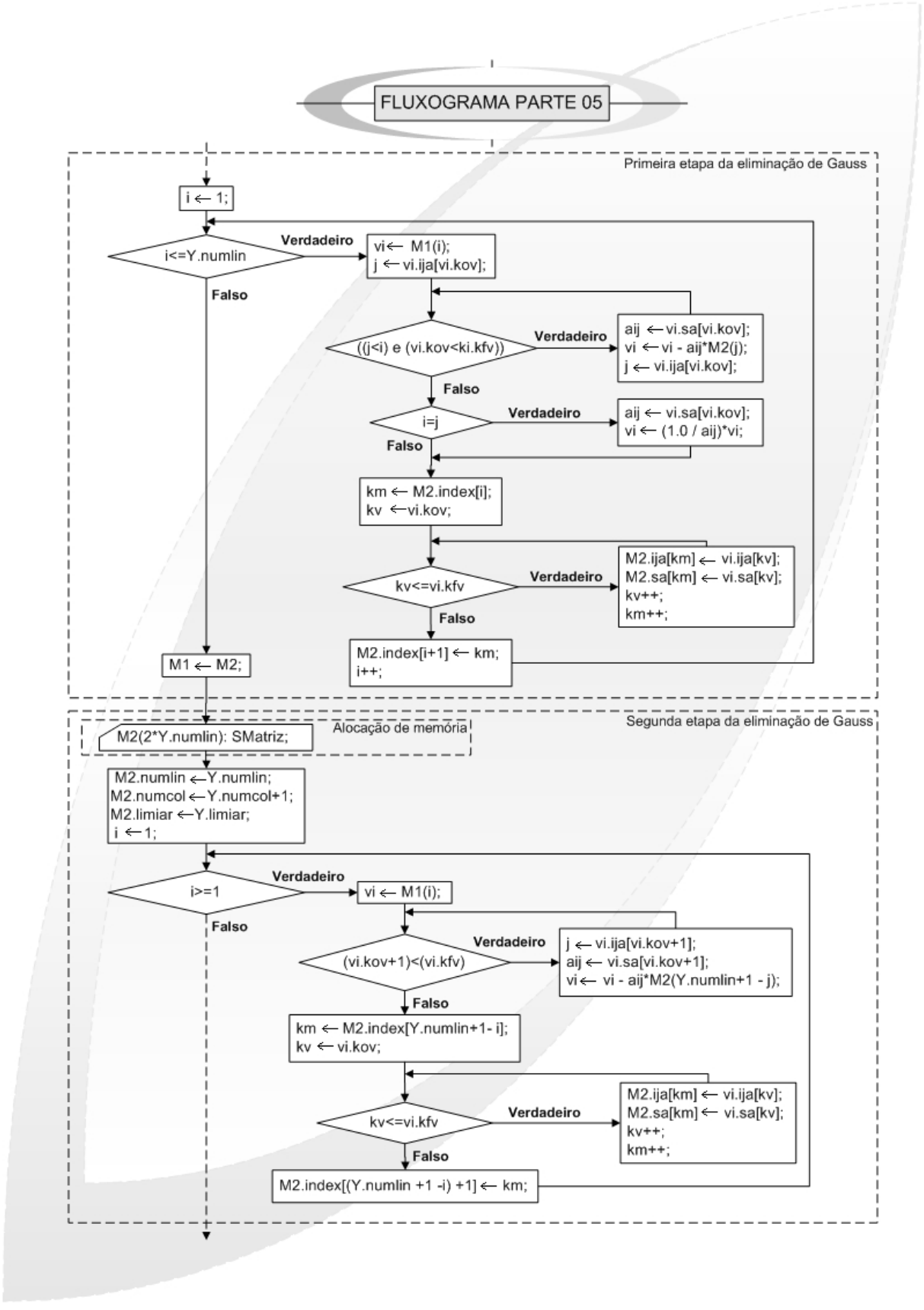

FIGURA 4.17 – Parte 05 do algoritmo de solução de sistemas.

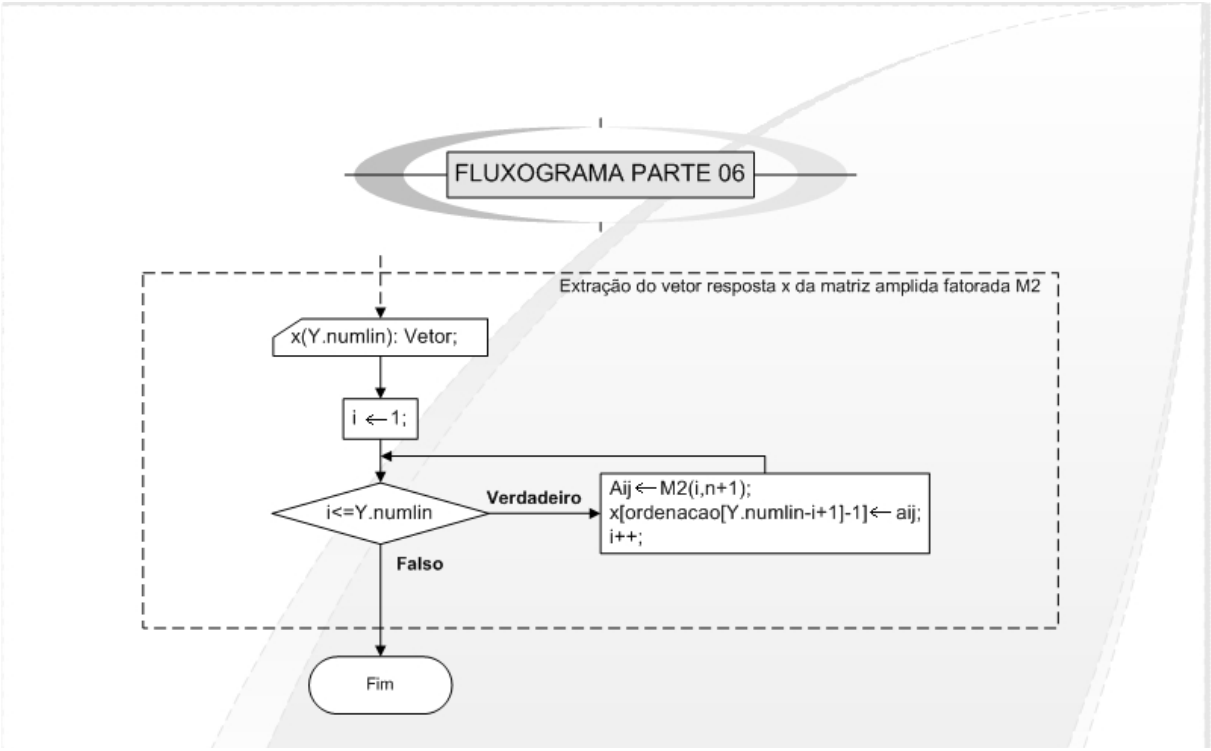

FIGURA 4.18 – Parte 06 do algoritmo de solução de sistemas.

4.7.4.1 – ELIMINAÇÃO DE GAUSS

Os sistemas lineares podem ser representados de uma forma geral por

$$\mathbf{Y} \cdot \mathbf{x} = \mathbf{b} \tag{4.12}$$

onde, **Y** é a matriz $\mathbf{n} \times \mathbf{n}$ dos coeficientes, **x** é o vetor dependente $\mathbf{n} \times 1$ e **b** é o vetor independente $\mathbf{n} \times 1$.

Uma maneira de se resolver o sistema seria pela obtenção da matriz $\mathbf{Y}^{-1}$ explicitamente. Isto, no entanto, além de ser computacionalmente pouco eficiente, seria impraticável para matrizes **Y** com dimensões elevadas. A razão é que, apesar de **Y** ser esparsa nos problemas mencionados anteriormente, sua inversa $\mathbf{Y}^{-1}$ em geral é cheia.

O sistema algébrico linear da equação (4.12) pode ser reescrito da seguinte forma:

$$\begin{bmatrix} \mathbf{y}_{11} & \mathbf{y}_{12} & \mathbf{y}_{13} & \cdots & \mathbf{y}_{1n} \\ \mathbf{y}_{21} & \mathbf{y}_{22} & \mathbf{y}_{23} & \cdots & \mathbf{y}_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{y}_{n1} & \mathbf{y}_{n2} & \mathbf{y}_{n3} & \cdots & \mathbf{y}_{nn} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_n \end{bmatrix} \tag{4.13}$$

A solução desse sistema é efetuada, em duas etapas, por combinações lineares das equações que constituem o sistema e cujo método é ligeiramente diferente do método da eliminação de Gauss: na primeira etapa, as linhas são operadas uma por vez, de cima para baixo, sendo os elementos de cada linha zerados da esquerda para a direita até o elemento precedente ao elemento diagonal, e o elemento diagonal, operado por último, para o valor 1; e, na segunda etapa, as linhas são operadas uma por vez, de baixo para cima, sendo os elementos de cada linha (exceto o elemento diagonal) zerados da esquerda para a direita até o último elemento. Estes procedimentos garantem que os zeros obtidos nos passos iniciais não sejam, destruídos nos passos subseqüentes do processo de eliminação. Os sistemas resultantes em cada uma das duas etapas descritas anteriormente são mostrados a seguir:

$$\begin{bmatrix} 1 & \mathbf{a'}_{12} & \mathbf{a'}_{13} & \cdots & \mathbf{a'}_{1n} \\ 0 & 1 & \mathbf{a'}_{23} & \cdots & \mathbf{a'}_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x_n} \end{bmatrix} = \begin{bmatrix} \mathbf{b'}_1 \\ \mathbf{b'}_2 \\ \vdots \\ \mathbf{b'_n} \end{bmatrix} \tag{4.14}$$

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x_n} \end{bmatrix} = \begin{bmatrix} \mathbf{b''}_1 \\ \mathbf{b''}_2 \\ \vdots \\ \mathbf{b''_n} \end{bmatrix} \tag{4.15}$$

A ligeira modificação no método da eliminação de Gauss tem como objetivo diminuir o número de divisões nas operações de combinações lineares e, portanto, aumentar o desempenho do algoritmo de solução de sistemas lineares.

4.7.4.2 – ORDENAÇÃO

A Eliminação Gaussiana tende a destruir a esparsidade. A Figura 4.19 mostra a estrutura da matriz jacobiana para o caso IEEE 300 barras. As Figuras 4.20 e 4.21 mostram a estrutura da fatoração LU para a matriz jacobiana do caso IEEE 300 barras. Observa-se pela decomposição LU que a introdução de *fill-in's* é elevada e, portanto, o desempenho se torna inaceitável para as aplicações propostas neste trabalho. Faz-se, então, necessário a utilização de um procedimento conhecido como ordenação para minimizar esta influência destrutiva da Eliminação Gaussiana. A ordenação seria a escolha adequada de uma determinada seqüência de eliminações, de modo a criar o menor número possível de elementos não-nulos após as operações pertinentes, ou seja, conservar a esparsidade da

matriz. Salienta-se que esse procedimento também diminui o número de operações aritméticas envolvidas, diminuindo os erros de arredondamento.

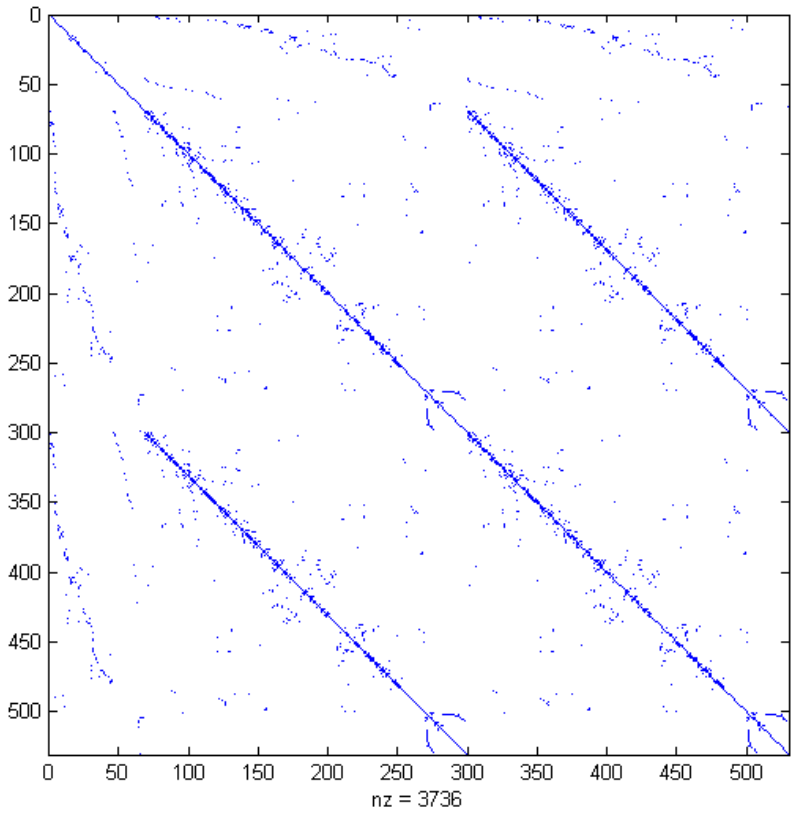

FIGURA 4.19 – Estrutura da matriz jacobiana para o caso IEEE 300 barras .

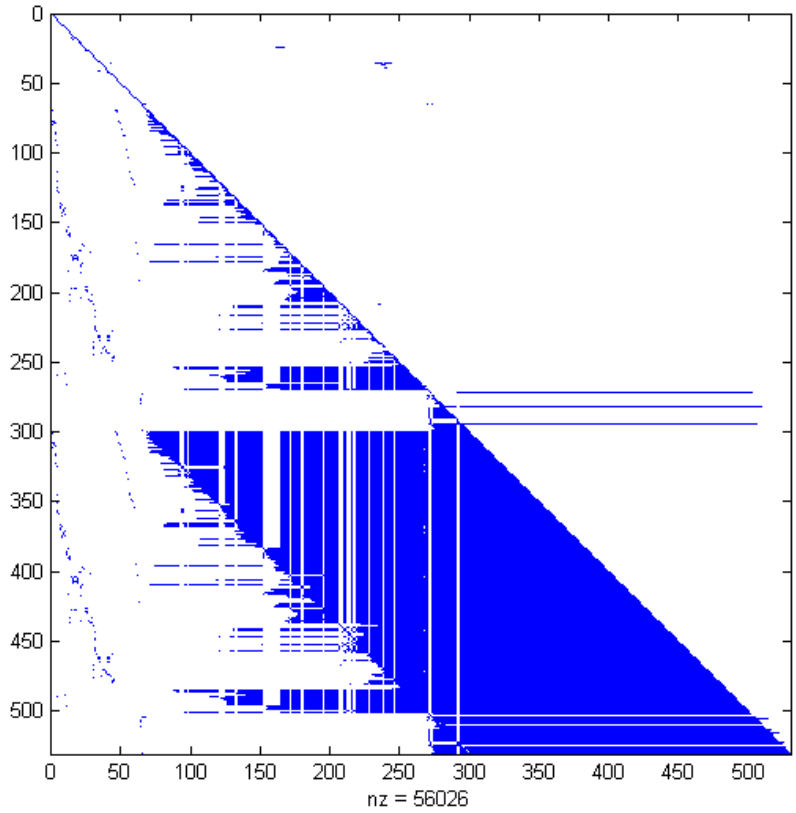

FIGURA 4.20 – Estrutura da fatoração L (LU) da matriz jacobiana para o caso IEEE 300 barras.

Os esquemas de ordenação de Tinney, abordados no Capítulo II, propõem a fatoração da matriz por coluna em ordem das colunas com menor quantidade de elementos. É importante ressaltar que para estas fatorações nenhuma ordenação matricial de fato é efetuada. Entretanto como o esquema utilizado é o RCOCL se torna inconveniente efetuar fatorações por coluna. Porém, pode-se ordenar a matriz na ordem das linhas com menos elementos e efetuar a fatoração por linha pela eliminação de Gauss. Após o processo de eliminação de Gauss ordena-se a matriz para seu estado original. Agora resta escolher qual dos três esquemas de Tinney é o mais adequado para a presente proposta.

O primeiro esquema de ordenação apesar de ser estático é a melhor escolha pois não define uma nova ordem de fatoração durante a eliminação de Gauss. O segundo esquema efetua para cada operação elementar da eliminação de Gauss uma nova ordem de fatoração. Entretanto, para o esquema RCOCL ter-se-ia que ordenar a matriz para cada operação elementar da eliminação de Gauss o que provocaria uma perda de desempenho significativa. O esquema três consegue prever qual linha introduzirá menos elementos no processo. No entanto, exigiria várias simulações de ordenação da matriz o que o torna inviável.

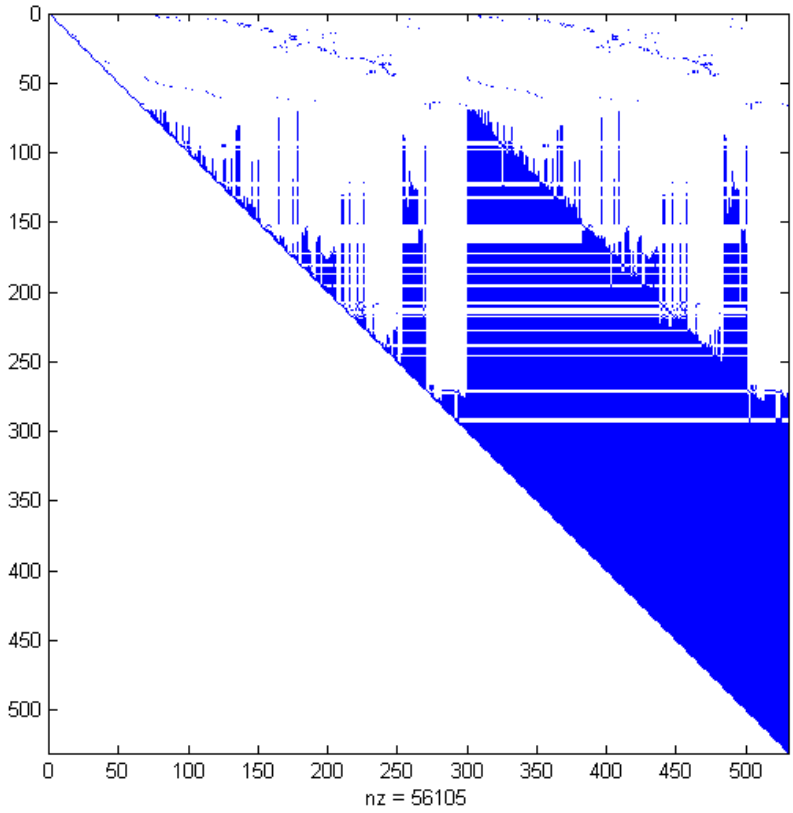

FIGURA 4.21 – Estrutura da fatoração U (LU) da matriz jacobiana para o caso IEEE 300 barras.

4.7.4.3 – ALOCAÇÃO DE MEMÓRIA

As três principais alocações de memória correspondem ao resultado da operação de ordenação, a primeira e a segunda etapa da fatoração matricial.

Para o processo de ordenação a quantidade de memória a ser alocada é a mesma da matriz a ser ordenada.

A quantidade de memória necessária para armazenar o resultado expresso pela equação (4.14) consiste em determinar a quantidade máxima de elementos não-nulos nesta expressão. Para tanto, deve-se considerar o pior caso de geração de *fill-in's*. Isto pode ser conseguido considerando que o primeiro elemento não-nulo, contando de cima para baixo, da coluna **j** da matriz ampliada, gere a quantidade máxima de *fill-in's*. Entretanto, seria necessário laços de varredura para obtenção destes elementos visto que o esquema de armazenamento é o RCOCL. Porém, a transposição da matriz ampliada permite que estes elementos sejam localizados imediatamente. Portanto, tem-se que a quantidade máxima $\mathbf{Q_i}$ de elementos não-nulos da linha **i** da matriz ampliada transposta é dada pela Equação (4.16).

$$\mathbf{Q_i} = \mathbf{i} - \mathbf{j} + 1 \qquad (4.16)$$

Portanto, a quantidade máxima $\mathbf{Q_{máx}}$ de elementos não-nulos da Equação (4.14) é dada por:

$$\mathbf{Q_{máx}} = \sum_{\mathbf{i}=1}^{\mathbf{n}} \mathbf{Q_i} \tag{4.17}$$

4.7.4.4 – DESCRIÇÃO DO ALGORITMO

O algoritmo da Solução de Sistemas Lineares mostrado pelas Figuras 4.13, 4.14, 4.15, 4.16, 4.17 e 4.18 é dividido em cinco blocos funcionais:

- Ordenação;
- Construção da matriz ampliada;
- Cálculo de memória para a primeira etapa da eliminação de Gauss;
- Eliminação de Gauss;
- Extração do vetor resposta **x** da matriz ampliada.

4.7.4.4.1 – CÁLCULO DE MEMÓRIA

A Figura 4.16 mostra o algoritmo responsável pelo cálculo de memória a ser alocada para a primeira etapa da Eliminação de Gauss. Este algoritmo determina a quantidade de elementos não-nulos pela aplicação da Equação (4.17). Para tanto, efetua a transposição da matriz ampliada dada como argumento e posteriormente aplica um laço que identifica a cada iteração **i** o índice de coluna **j** do primeiro elemento da linha **i** da transposta da matriz ampliada calculando a parcela **i** da somatória da Equação (4.17). Ao fim de todas as iterações do laço é feita novamente a transposição da matriz com o intuito de restaurar o estado inicial.

4.7.4.4.2 – ORDENAÇÃO

Este algoritmo consiste no Bloco I, Bloco II e no algoritmo de construção da matriz ampliada mostrados, respectivamente, nas Figuras 4.13, 4.14 e 4.15.

Bloco I: Consiste na definição da seqüência de índices de linhas para o processo de ordenação da matriz argumento. O primeiro passo consiste em varrer, por meio de um único laço, o arranjo **index** da matriz argumento e fazer uma contagem a cada iteração da quantidade de elementos de cada linha. A ordenação dos índices de linha, na seqüência da quantidade de seus elementos, utiliza o princípio do algoritmo de transposição que durante sua execução também efetua um processo de ordenação. Para tanto, faz-se uso de quatro laços independentes, cada um com um único argumento, e realizando aproximadamente **numlin** iterações.

Bloco II: Como mencionado anteriormente o processo utilizado pelo algoritmo de transposição efetua um processo de ordenação. Daí o uso de um algoritmo semelhante ao algoritmo de transposição. Os arranjos **ija** e **sa** do esquema RCOCL apresentam seus elementos não-nulos ordenados por linha e dentro de cada linha ordenados por colunas. Entretanto, o uso do princípio do algoritmo de transposição em conjunto com a seqüência de ordenação distorce ligeiramente o esquema RCOCL. Os arranjos **ija** e **sa** passam a apresentar seus elementos não-nulos ordenados por linha e dentro de cada linha não necessariamente ordenados por colunas. A correção desta distorção é feita pela aplicação do processo de transposição no algoritmo de construção da matriz ampliada. A Figura 4.22 mostra a estrutura da matriz jacobiana ordenada para o caso IEEE 300 barras. Observa-se que a matriz aumenta suas características de dominância diagonal. As Figuras 4.23 e 4.24 mostram a estrutura da fatoração LU para a matriz jacobiana ordenada do caso IEEE 300 barras. Comparando estas figuras com as Figuras 4.20 e 4.21 observa-se que a introdução de *fill-in's* é significativamente minimizada quando utiliza-se o processo de ordenação e, portanto, o desempenho se torna satisfatório para as aplicações propostas neste trabalho.

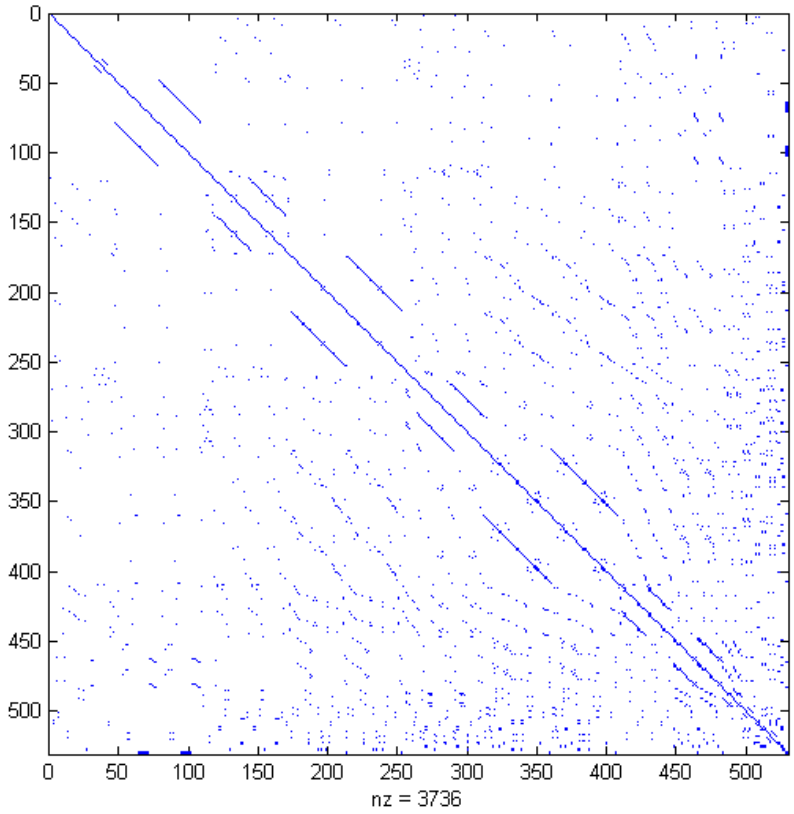

FIGURA 4.22 – Estrutura da matriz jacobiana ordenada para o caso IEEE 300 barras.

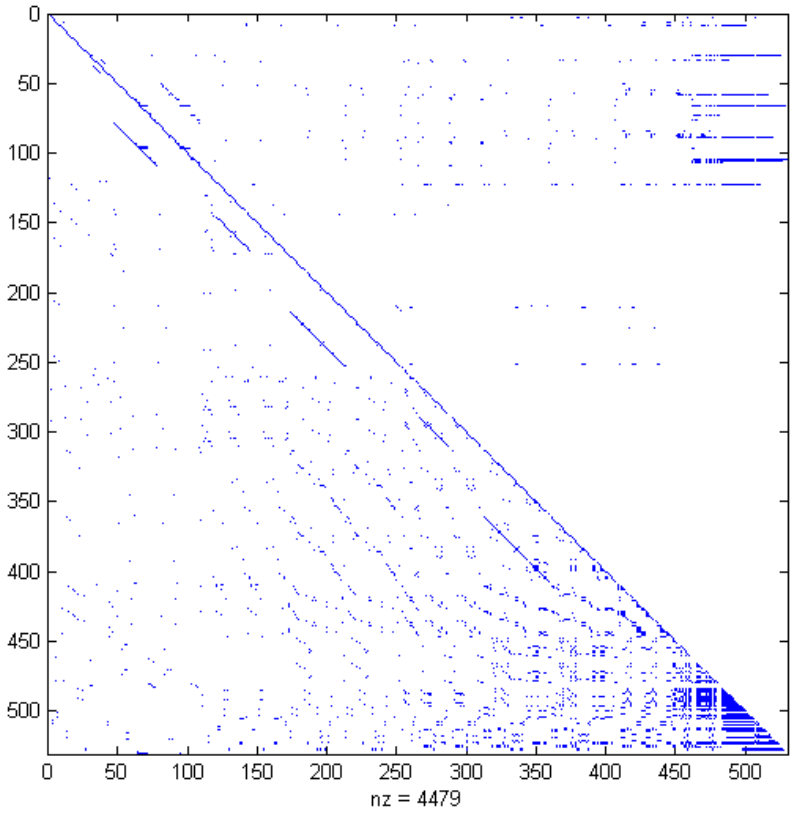

FIGURA 4.23 – Fatoração L (LU) para a matriz jacobiana ordenada do caso IEEE 300.

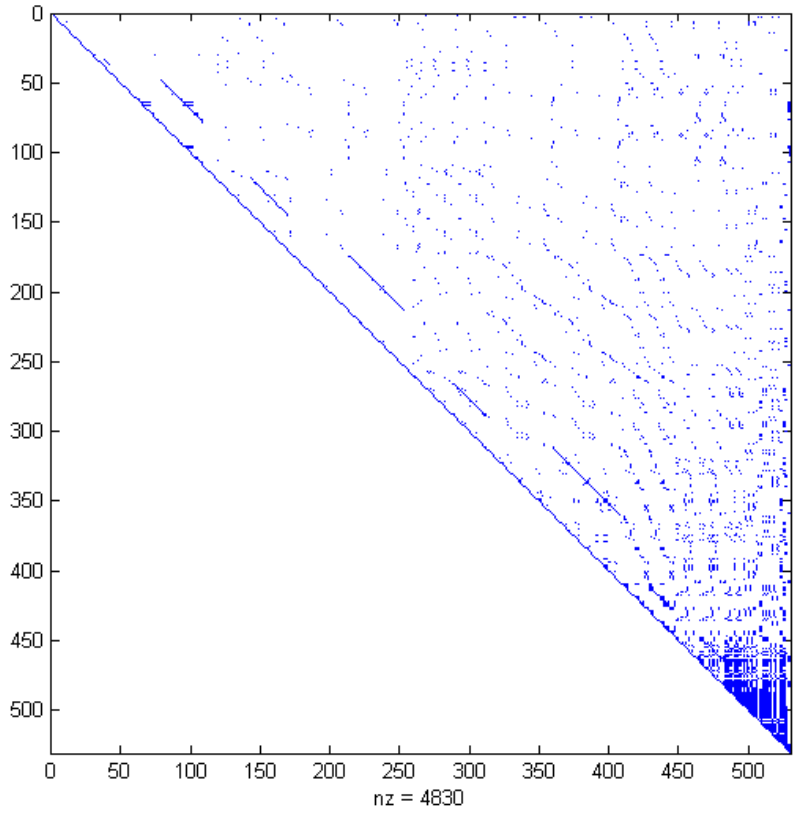

FIGURA 4.24 – Fatoração U (LU) para a matriz jacobiana ordenada do caso IEEE 300

4.7.4.4.3 – CONSTRUÇÃO DA MATRIZ AMPLIADA

A Figura 4.15 mostra o algoritmo que realiza a construção da matriz ampliada do sistema constituído pela matriz e o vetor dados como argumentos. A matriz argumento para este algoritmo é a matriz resultante do processo de ordenação. Portanto o processo de construção da matriz ampliada consiste de um processo de transposição seguido da inserção dos elementos que constituem o vetor argumento. A inserção dos elementos do vetor argumento na matriz argumento é efetuado por um único laço que varre simultaneamente o vetor argumento e as linha da matriz argumento preenchendo em cada iteração o último espaço de memória destinada ao elemento a ser inserido.

4.7.4.4.4 – ELIMINAÇÃO DE GAUSS

A Figura 4.17 mostra o algoritmo responsável pela primeira e segunda etapa da Eliminação de Gauss. A matriz argumento da primeira etapa da eliminação de Gauss é a matriz ampliada. A matriz argumento da segunda etapa é a matriz resultado da primeira etapa. Este algoritmo é dividido em dois blocos funcionais:

i. **Primeira etapa da eliminação de Gauss**: O algoritmo varre por meio de um laço cada linha **i** da matriz argumento. Esta varredura é efetuada de cima para baixo. Para cada iteração deste laço tem-se três tarefas distintas. A primeira tarefa consiste em fazer $\mathbf{aij} = 0$ para $\mathbf{j} < \mathbf{i}$ através de operações lineares com a classe SVetor e está condicionada a um laço cujas a iterações só deixam de ser executadas quando $\mathbf{j} \geq \mathbf{i}$ ou quando não existirem mais elementos na linha **i**. A segunda tarefa consiste em fazer $\mathbf{aij} = 1$ para $\mathbf{j} = \mathbf{i}$ através de uma operação linear com a classe SVetor. A terceira tarefa é a inserção do vetor resultado das operações lineares das duas tarefas anteriores na matriz resultado. Um único laço é responsável por esta tarefa e tem como quantidade de iterações a quantidade de elementos do vetor resultado das operações lineares. Como o vetor resultado é esparso a quantidade iterações é pequena.

ii. **Segunda etapa da eliminação de Gauss**: O algoritmo varre por meio de um laço cada linha **i** da matriz argumento. Esta varredura é efetuada de baixo para cima. Para cada iteração deste laço tem-se duas tarefas distintas. A primeira tarefa consiste em fazer $\mathbf{aij} = 0$ para $\mathbf{j} > \mathbf{i}$ através de operações lineares com a classe SVetor e está condicionada a um laço cujas a iterações só deixam de ser executadas quando existirem apenas dois elementos na linha **i**. A segunda tarefa é a inserção do vetor resultado das operações lineares da tarefa precedente na matriz resultado. Um único laço é responsável por esta tarefa e executa apenas duas iterações.

4.8 – DECLARAÇÃO DAS PRINCIPAIS FUNÇÕES DA CLASSE SMATRIZ

Neste tópico é descrito um exemplo de como são declaradas as funções responsáveis pela adição, multiplicação, transposição de matrizes esparsas e da solução de sistemas lineares esparsos. A declaração destas operações utiliza a sobrecarga de operadores que permite o programa cliente utilizar a notações e idéias usuais de operações matriciais.

A definição da Classe SMatriz dada a seguir em C++ mostra as declarações das funções membros e das funções *friend*.

```
class SMatriz {

    friend Vetor &operator/( Vetor &b, SMatriz &A );      //Solução de Sistemas Lineares
    friend SMatriz &transp(SMatriz &A);                   //Transpõe a matriz A

public:
        //Construtor
        SMatriz(int QuantElementos );
        //Destrutor
        ~SMatriz();
        //Funções membro da Classe
        SMatriz &operator+(SMatriz &B);          //Soma de matrizes esparsas
        SMatriz &operator*(SMatriz &B);          //Multiplicação de matrizes esparsas

private:        //Membros de dado da Classe SMatriz usando o esquema RCOCL

        complex< double > *sa;      //Declara um ponteiro para  o arranjo sa
        int *ija;                   //Declara um ponteiro para o arranjo ija
        int *index;                 //Declara um ponteiro para o arranjo index
        int numlin;                 //Declara a variável auxiliar numlin
        int numcol;                 //Declara a variável auxiliar numcol
        double limiar;              //Declara a variável auxiliar limiar
};
```

4.9 – CONCLUSÃO

Neste capítulo foi apresentado o esquema de armazenamento compacto RCOCL (Representação Completa e Ordenada por Comprimento de Linha) e a implementação de suas principais operações matriciais na Programação Orientada a Objeto. Os algoritmos desenvolvidos apresentam bom desempenho, boa alocação de memória, baixa complexidade e boa legibilidade para uso em aplicações que evolvam matrizes de grande dimensões com grau de esparsidade elevado.

A Programação Orientada a Objeto permitiu o uso dos conceitos como funções membros, dados membros, sobrecarga de operadores, entre outros, resultando nas Classes SMatriz e SVetor. Estas classes são independentes da estrutura das matrizes esparsas que irão representar e, portanto, podem ser usadas em qualquer aplicação.

CAPÍTULO 5 – APLICAÇÃO DAS TÉCNICAS DE ESPARSIDADES NO ESTUDO DE FLUXO DE CARGA

5.1 – INTRODUÇÃO

Neste capítulo, será feito um estudo comparativo do desempenho das técnicas de esparsidade na simulação de um fluxo de carga, considerando o sistema IEEE – 14, 118 e 300 barras, utilizando uma estrutura de matrizes no formato convencional e outra no formato esparso. Os testes foram desenvolvidos num microcomputador AMD Duron XP 1100MHz, com 247 MB de memória RAM.

A técnica de solução utilizada para o fluxo de carga é o Método de Newton-Raphson Completo, sendo este método mais universalmente aceito (BROWN, 1977). Geralmente a matriz admitância é relativamente esparsa sendo esta esparsidade conservada na matriz Jacobiana, o que facilita a utilização de técnicas de esparsidade na solução do sistema.

Neste Capítulo, são descritas algumas características do programa computacional desenvolvido para análise de fluxo de carga, assim como uma breve descrição teórica do método de Newton-Raphson.

5.2 – MÉTODO DE NEWTON-RAPHSON COMPLETO PARA SOLUÇÃO DE FLUXO DE CARGA

5.2.1 – RESOLUÇÃO DE SISTEMAS ALGÉBRICOS PELO MÉTODO DE NEWTON-RAPHSON

O método de Newton-Raphson generalizado é um algoritmo iterativo para resolver um conjunto de equações não lineares simultâneas dentro de um número igual de variáveis, sendo que, em cada iteração, o problema é aproximado pela equação matricial linear. A aproximação linearizada pode ser melhor visualizada no caso de um problema para um sistema unidimensional.

Considere a equação algébrica não-linear dada pela Equação 5.1.

$$g(x) = 0 \tag{5.1}$$

Em termos geométricos a solução da Equação 5.1 corresponde ao ponto x_s em que a curva $g(x)$ corta o eixo horizontal x, conforme mostra a Figura 5.1.

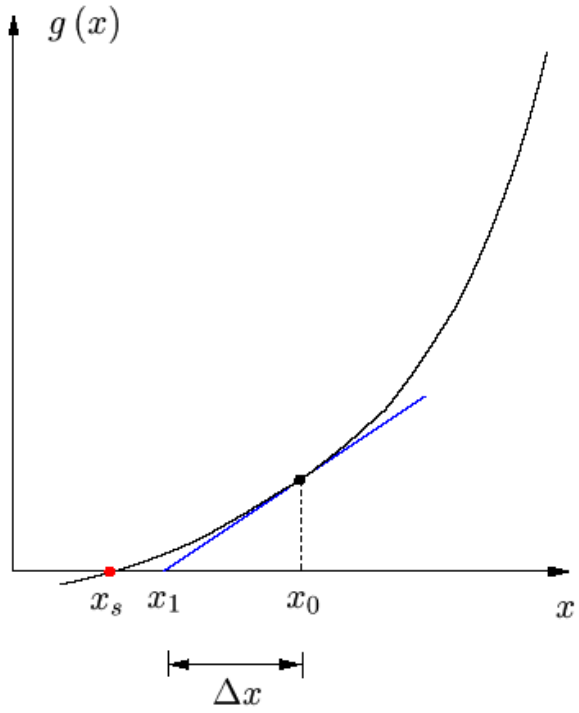

FIGURA 5.1 – Aproximação linear de uma única variável

A resolução da Equação 5.1 pelo método de Newton, resulta em um processo iterativo que segue os seguintes passos:

i) Inicializar o contador de iterações $v = 0$, escolhendo um ponto inicial $x = x^{(v)} = x^{(0)}$.

ii) Calcular o valor da função $g(x)$ no ponto $x = x^{(v)}$, ou seja, $g(x^{(v)})$.

iii) Comparar o valor calculado $g(x^{(v)})$ com uma tolerância especificada ε. Se $\left|g(x^{(v)})\right| \leq \varepsilon$, então $x = x^{(v)}$ corresponderá à solução procurada dentro da faixa de tolerância $\pm\varepsilon$. Caso contrário prosseguir.

iv) Linearizar a função $g(x)$ em torno de um ponto $(x^{(v)}, g(x^{(v)}))$ por intermédio da série de Taylor desprezando os termos de ordem igual e superior a dois. Tal processo resume-se ao cálculo da derivada $g'(x^{(v)})$.

$$g(x^{(v)} + \Delta x^{(v)}) \cong g(x^{(v)}) + g'(x^{(v)}) \cdot \Delta x^{(v)} \qquad (5.2)$$

v) Resolver o problema linearizado, ou seja, encontrar $\Delta x^{(v)}$ conforme a Equação 5.4. Em seguida estimar um novo valor de x conforme a Equação 5.5.

$$g(x^{(v)}) + g'(x^{(v)}) \cdot \Delta x^{(v)} = 0 \tag{5.3}$$

$$\Delta x^{(v)} = -\frac{g(x^{(v)})}{g'(x^{(v)})} \tag{5.4}$$

$$x^{(v+1)} = x^{(v)} + \Delta x^{(v)} \tag{5.5}$$

vi) Fazer $v \leftarrow v+1$ e voltar para o passo ii.

As etapas do processo podem ser melhor exemplificadas e visualizadas através da Figura 5.2, onde $x^{(3)}$ está suficientemente próximo de x_s ($g(x^{(3)})$ dentro da faixa de tolerância $\pm\varepsilon$) a ponto de ser considerado como a solução da equação $g(x) = 0$.

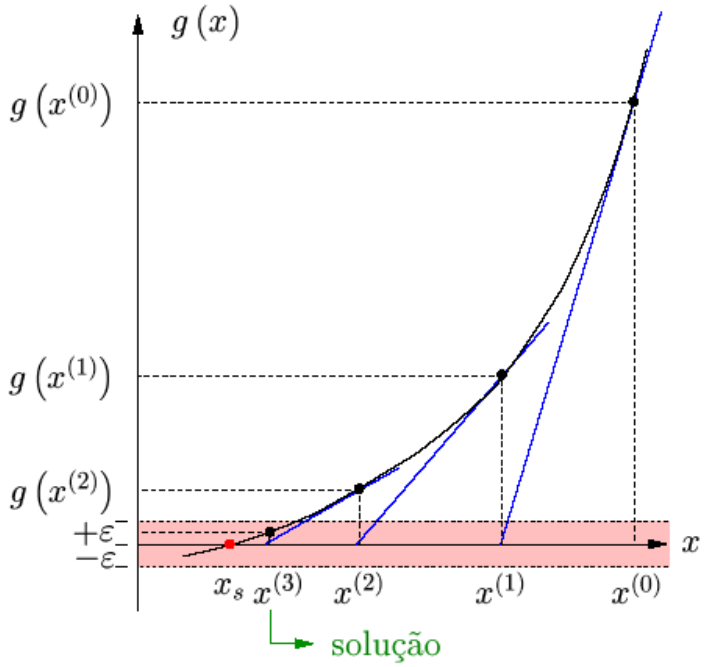

FIGURA 5.2 – Processo iterativo de Newton para um sistema unidimensional

Considere-se agora o caso de um sistema n-dimensional de equações algébricas não-lineares dadas pela Equação 5.6.

$$\underline{g}(\underline{x}) = 0 \tag{5.6}$$

Sendo $\underline{g}$ e $\underline{x}$ são vetores de dimensão (n×1) correspondentes respectivamente às funções e incógnitas dadas pelas Equações 5.7 e 5.8.

$$\underline{g}(\underline{x}) = [g_1(\underline{x}), g_2(\underline{x}), \cdots, g_n(\underline{x})]^T \tag{5.7}$$

$$\underline{x} = [x_1, x_2, \cdots, x_n]^T \tag{5.8}$$

Os passos do algoritmo de solução para o caso n-dimensional são basicamente os mesmos do caso unidimensional.

A diferença está no passo iv em que se realiza a linearização de $\underline{g}(\underline{x})$, onde tal linearização se dá em torno de $x = x^{(v)}$ e é dada pela Equação 5.9.

$$\underline{g}(\underline{x}^{(v)} + \Delta\underline{x}^{(v)}) \cong \underline{g}(\underline{x}^{(v)}) + J(\underline{x}^{(v)}) \cdot \Delta\underline{x}^{(v)} \tag{5.9}$$

Sendo que J é chamada de matriz Jacobiana e é dada pela Equação 5.10.

$$J = \frac{\partial}{\partial \underline{x}}\underline{g} = \begin{bmatrix} \frac{\partial}{\partial x_1} g_1 & \frac{\partial}{\partial x_2} g_1 & \cdots & \frac{\partial}{\partial x_n} g_1 \\ \frac{\partial}{\partial x_1} g_2 & \frac{\partial}{\partial x_2} g_2 & \cdots & \frac{\partial}{\partial x_n} g_2 \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} g_n & \frac{\partial}{\partial x_2} g_n & \cdots & \frac{\partial}{\partial x_n} g_n \end{bmatrix} \tag{5.10}$$

O vetor de correção é calculado impondo-se as Equações 5.11 e 5.12.

$$\underline{g}(\underline{x}^{(v)}) + J(\underline{x}^{(v)}) \cdot \Delta\underline{x}^{(v)} \tag{5.11}$$

$$\Delta\underline{x}^{(v)} = -\left[J(\underline{x}^{(v)})\right]^{-1} \cdot \underline{g}(\underline{x}^{(v)}) \tag{5.12}$$

O algoritmo para resolução do sistema de equações $\underline{g}(\underline{x}) = 0$ pelo método de Newton é dado pelos seguintes passos:

i) Inicializar o contador de iterações $v = 0$, escolhendo um ponto inicial $\underline{x} = \underline{x}^{(v)} = \underline{x}^{(0)}$.

ii) Calcular o valor da função $\underline{g}(\underline{x})$ no ponto $\underline{x} = \underline{x}^{(v)}$, ou seja, $\underline{g}(\underline{x}^{(v)})$.

iii) Teste de convergência. Se $\left|g_i(\underline{x}^{(v)})\right| \leq \varepsilon$ para $i = 1 \ldots n$, então $\underline{x} = \underline{x}^{(v)}$ será a solução procurada dentro da faixa de tolerância $\pm\varepsilon$ e o processo convergiu. Caso contrário, prosseguir.

iv) Calcular a matriz Jacobiana $J(\underline{x}^{(v)})$.

v) Determinar o novo ponto $\underline{x}^{(v+1)}$ partindo das Equações 5.13 e 5.14.

$$\Delta \underline{x}^{(v)} = -\left[J(\underline{x}^{(v)})\right]^{-1} \cdot \underline{g}(\underline{x}^{(v)}) \tag{5.13}$$

$$\underline{x}^{(v+1)} = \underline{x}^{(v)} + \Delta \underline{x}^{(v)} \tag{5.14}$$

vi) Fazer $v \leftarrow v + 1$ e voltar para o passo ii.

5.2.2 – AVALIAÇÃO DO MÉTODO DE NEWTON-RAPHSON

- Converge para vários casos em que outros métodos (por exemplo, Gauss-Seidel) divergem, sendo mais confiável.
- O número de iterações necessárias para a convergência independe da dimensão do problema.
- Requer mais espaço de memória para armazenamento, devido à matriz Jacobiana.
- O tempo computacional por iteração é maior, pois deve-se inverter a matriz Jacobiana e multiplica-la por um vetor.
- As técnicas de armazenamento compacto e de fatoração reduziriam de maneira significativa o espaço de memória necessário e o esforço computacional.
- Apresenta convergência quadrática.
- Não é sensível à escolha da barra de referência.
- É sensível à escolha do ponto inicial.

5.2.3 – RESOLUÇÃO DO PROBLEMA DE FLUXO DE CARGA PELO MÉTODO DE NEWTON-RAPHSON

A idéia básica do Fluxo de Carga é a obtenção das condições de operação (tensões, fluxos de potência) de uma rede elétrica em função de sua topologia e dos níveis de demanda e geração de potência.

Nos tópicos anteriores foi descrita a formulação básica do problema e apresentado a solução de sistemas de equações algébricas não-lineares pelo método de Newton. Em meados da década de 60, técnicas de armazenamento compacto e ordenamento da fatoração (TINNEY & WALKER, 1967) tornaram o método de Newton muito mais rápido, exigindo pequeno espaço de memória, mantendo as características de ótima convergência, passando a ser adotado pela maioria das empresas de energia elétrica.

A aplicação do método de Newton na solução do fluxo de carga parte-se do balanço de potência injetado nas barras (soma das potências distribuídas pelos ramos conectados

em cada barra), onde aplicando-se a lei das correntes de Kirchhoff (LCK) para todas as NB barras da rede elétrica tem-se as Equações 5.15 e 5.16.

$$P_k = V_k \sum_{m \in \kappa} V_m (G_{km} \cos\theta_{km} + B_{km} \operatorname{sen}\theta_{km}) \quad k = 1, \ldots, NB \tag{5.15}$$

$$Q_k = V_k \sum_{m \in \kappa} V_m (G_{km} \operatorname{sen}\theta_{km} - B_{km} \cos\theta_{km}) \quad k = 1, \ldots, NB \tag{5.16}$$

Onde:

- κ – conjunto das barras vizinhas da barra k (diretamente conectadas à barra k) mais a própria barra k.
- V_k, V_m – magnitude das tensões nas barras k e m.
- θ_k, θ_m – ângulos de fase das tensões nas barras k e m.

Na formulação do problema, a cada barra da rede são associadas quatro variáveis, sendo que duas delas entram no problema como dados e duas como incógnitas:

V_K – magnitude da tensão nodal (barra k);

θ_k – ângulo da tensão nodal;

P_k – injeção líquida (geração menos carga) de potência ativa;

Q_k – injeção líquida de potência reativa.

Dependendo de quais variáveis nodais entram como dados e quais são consideradas como incógnitas, define-se três tipos de barras conforme a Tabela 5.1.

TABELA 5.1 – Tipos de barras da rede elétrica.

Tipo de Barra	Variáveis Dadas	Variáveis Calculadas
PQ	P_K e Q_k	V_k e θ_k
PV	P_k e V_k	Q_k e θ_k
REFERÊNCIA	V_k e θ_k	P_k e Q_k

Logo, a rede tem $(PV + PQ + 1)$ barras formando $2 \times (PQ + PV + 1)$ dados e incógnitas.

Em função da existência de dois tipos de incógnitas,o problema de fluxo de carga pode ser decomposto em dois subsistemas de equações algébricas:

Subsistema 1 de dimensão $2\times PQ+PV$

Neste subproblema são dados P_k e Q_k nas barras PQ, e P_k e V_k nas barras PV, então é calculado V_k e θ_k nas barras PQ, e θ_k nas barras PV, resultando num sistema com um total de $(2\times PQ+PV)$ incógnitas.

Para cada potência dada, pode-se escrever as Equações 5.17 e 5.18, resultando em um sistema de $(2\times PQ+PV)$ equações e mesmo número de incógnitas (sistema determinado).

$$P_k^{esp}-P_k=0, \quad \text{para barras PQ e PV} \tag{5.17}$$

$$Q_k^{esp}-Q_k=0, \quad \text{para barras PQ} \tag{5.18}$$

Subsistema 2 de dimensão $PV+2$

Após resolvido o Subsistema 1, e já com os novos valores de V_k e θ_k para todas as barras, parte-se para a determinação das potências nodais desconhecidas.

As incógnitas restantes são P para a barra de referência (uma incógnita) e Q para as barras PV e a barra de referência ($PV+1$ incógnitas), o que resulta num total de $PV+2$ incógnitas a serem determinadas. Como o estado da rede é conhecido, basta aplicar diretamente as Equações 5.15 e 5.16 das potências nodais para as respectivas barras.

As incógnitas do subsistema 1 podem ser escritas como:

$$\underline{x}=\begin{bmatrix}\underline{\theta}\\ \underline{V}\end{bmatrix}\begin{matrix}\}\ PV+PQ\\ \}\ PQ\end{matrix} \tag{5.19}$$

Em que $\underline{\theta}$ é o vetor dos ângulos das tensões das barras PQ e PV e $\underline{V}$ é o vetor das magnitudes das tensões das barras PQ. As Equações (5.17) e (5.18) para o subsistema 1 podem ser reescritas conforme as Equações (5.20) e (5.21).

$$\Delta\underline{P}=\underline{P}^{esp}-\underline{P}(\underline{V},\underline{\theta}) \quad \text{para barras PQ e PV} \tag{5.20}$$

$$\Delta\underline{Q}=\underline{Q}^{esp}-\underline{P}(\underline{V},\underline{\theta}) \quad \text{para barras PQ} \tag{5.21}$$

Onde $\underline{P}$ é o vetor das injeções de potência ativa nas barras PQ e PV e $\underline{Q}$, o das injeções de potência reativa nas barras PQ, podendo ser calculados através das Equações (5.15) e (5.16). Os vetores $\Delta\underline{P}$ e $\Delta\underline{Q}$ são chamados de *mismatches* (ou resíduos, ou erros) de potência ativa e reativa.

Considere a função vetorial dada pela Equação (5.22).

$$\underline{g}(\underline{x}) = \begin{bmatrix} \Delta\underline{P} \\ \Delta\underline{Q} \end{bmatrix} \tag{5.22}$$

Como a solução do subsistema 1 é obtida quando os *mismatches* são iguais a zero, as equações do subsistema 1 podem ser colocadas na forma da Equação (5.23).

$$\underline{g}(\underline{x}) = \begin{bmatrix} \Delta\underline{P} \\ \Delta\underline{Q} \end{bmatrix} = 0 \tag{5.23}$$

A equação (5.23) é formada por um sistema de equações algébricas não-lineares e pode ser resolvida pelo método de Newton, abordado neste tópico.

O ponto central da resolução do sistema dado pela Equação (5.23) pelo método de Newton consiste na determinação do vetor de correção do estado $\Delta\underline{x}$ a cada iteração.

Para uma certa iteração v, $\Delta\underline{x}$ é obtido através da Equação (5.24).

$$\underline{g}(\underline{x}^{(v)}) = -\underline{J}(\underline{x}^{(v)}) \cdot \Delta\underline{x}^{(v)} \tag{5.24}$$

Para o subsistema 1 (determinação das variáveis de estado desconhecidas) tem-se as Equações (5.25), (5.26) e (5.27).

$$\underline{g}(\underline{x}^{(v)}) = \begin{bmatrix} \Delta\underline{P}^{(v)} \\ \Delta\underline{Q}^{(v)} \end{bmatrix} \tag{5.25}$$

$$\Delta\underline{x}^{(v)} = \begin{bmatrix} \Delta\underline{\theta}^{(v)} \\ \Delta\underline{Q}^{(v)} \end{bmatrix} \tag{5.26}$$

$$\underline{J}(\underline{x}^{(v)}) = \begin{bmatrix} \dfrac{\partial(\Delta\underline{P})}{\partial\underline{\theta}} & \dfrac{\partial(\Delta\underline{P})}{\partial\underline{V}} \\ \dfrac{\partial(\Delta\underline{Q})}{\partial\underline{\theta}} & \dfrac{\partial(\Delta\underline{Q})}{\partial\underline{V}} \end{bmatrix} \tag{5.27}$$

Lembrando das Equações (5.20) e (5.21) (cujas derivadas aparecem na matriz Jacobiana) e de que os valores especificados das potências são constantes, pode-se escrever a Equação (5.28), resultando na matriz Jacobiana dada pela Equação (5.29).

$$\frac{\partial(\underline{P})}{\partial\underline{\theta}} = \frac{\partial(\underline{P}^{esp} - \underline{P}(\underline{V},\underline{\theta}))}{\partial\underline{\theta}} = -\frac{\partial(\underline{P}(\underline{V},\underline{\theta}))}{\partial\underline{\theta}} \tag{5.28}$$

$$J(\underline{x}^{(v)}) = -\begin{bmatrix} \dfrac{\partial(\underline{P})}{\partial\underline{\theta}} & \dfrac{\partial(\underline{P})}{\partial\underline{V}} \\ \dfrac{\partial(\underline{Q})}{\partial\underline{\theta}} & \dfrac{\partial(\underline{Q})}{\partial\underline{V}} \end{bmatrix} \tag{5.29}$$

As submatrizes que compõem a matriz Jacobiana são geralemtne representadas pelas Equações (5.30), (5.31), (5.32) e (5.33), onde finalmente as equações do sistema podem ser colocadas na forma da equação (5.34).

$$H = \frac{\partial(\underline{P})}{\partial\underline{\theta}} \tag{5.30}$$

$$N = \frac{\partial(\underline{P})}{\partial\underline{V}} \tag{5.31}$$

$$M = \frac{\partial(\underline{Q})}{\partial\underline{\theta}} \tag{5.32}$$

$$L = \frac{\partial(\underline{Q})}{\partial\underline{V}} \tag{5.33}$$

$$\begin{bmatrix} \Delta\underline{P}^{(v)} \\ \Delta\underline{Q}^{(v)} \end{bmatrix} = \begin{bmatrix} H & N \\ M & L \end{bmatrix}^{(v)} \cdot \begin{bmatrix} \Delta\underline{\theta}^{(v)} \\ \Delta\underline{V}^{(v)} \end{bmatrix} \tag{5.34}$$

As expressões para os elementos das matrizes H, M, N e L são obtidas a partir das Equações (5.15) e (5.16) das potências nodais e definidas pelas equações (5.35), (5.36), (5.37) e (5.38).

$$H\begin{cases} H_{km} = \partial P_k / \partial\theta_m = V_k V_m (G_{km}\,\mathrm{sen}\,\theta_{km} - B_{km}\cos\theta_{km}) \\ H_{kk} = \partial P_k / \partial\theta_k = -V_k^2 B_{kk} - V_k \sum_{m\in\kappa} V_m (G_{km}\,\mathrm{sen}\,\theta_{km} - B_{km}\cos\theta_{km}) \end{cases} \tag{5.35}$$

$$N\begin{cases} N_{km} = \partial P_k / \partial V_m = V_k (G_{km}\cos\theta_{km} + B_{km}\,\mathrm{sen}\,\theta_{km}) \\ N_{kk} = \partial P_k / \partial V_k = V_k G_{kk} + \sum_{m\in\kappa} V_m (G_{km}\cos\theta_{km} + B_{km}\,\mathrm{sen}\,\theta_{km}) \end{cases} \quad (5.36)$$

$$M\begin{cases} M_{km} = \partial Q_k / \partial \theta_m = -V_k V_m (G_{km}\cos\theta_{km} + B_{km}\,\mathrm{sen}\,\theta_{km}) \\ M_{kk} = \partial Q_k / \partial \theta_K = -V_k^2 G_{kk} + V_k \sum_{m\in\kappa} V_m (G_{km}\cos\theta_{km} + B_{km}\,\mathrm{sen}\,\theta_{km}) \end{cases} \quad (5.37)$$

$$L\begin{cases} L_{km} = \partial Q_k / \partial V_m = V_k (G_{km}\,\mathrm{sen}\,\theta_{km} - B_{km}\cos\theta_{km}) \\ L_{kk} = \partial Q_k / \partial V_k = -V_k G_{kk} + \sum_{m\in\kappa} V_m (G_{km}\,\mathrm{sen}\,\theta_{km} - B_{km}\cos\theta_{km}) \end{cases} \quad (5.38)$$

5.3 – DESCRIÇÃO DO PROGRAMA DE FLUXO DE CARGA

Neste tópico, são descritas algumas características do programa computacional desenvolvido para a análise de fluxo de carga em sistemas de energia elétrica, transmissão, subtransmissão e distribuição.

A interfase do programa do fluxo de carga se dá através de um conjunto de arquivos externos com dados de entrada e os resultados obtidos pelo programa são automaticamente gravados em um arquivo externo com dados de saída, onde todos esses arquivos estão no formato de texto ASCII. Cada campo dentro dos arquivos está em conformidade com a formatação especificada nos tópicos seguintes.

Fazendo-se uso de um editor de textos, o conteúdo dos arquivos pode ser facilmente impresso ou exibido no monitor de vídeo.

O conjunto de arquivos externo define as características da rede elétrica e são divididos em 4 arquivos:

- **Dados de Barra (dadosbarra.dat)** – Este arquivo contém dados relativos aos parâmetros de barra da rede.
- **Dados de Ramo (dadosramos.dat)** – Este arquivo contém dados relativos aos parâmetros de linha da rede.
- **Dados Gerais (dadosgerais.dat)** – Este arquivo contém dados relativos aos parâmetros que serão utilizados para o cálculo do fluxo de carga.
- **Dados para Geração (dadosgeradores.dat)** – Este arquivo contém dados relativos aos parâmetros das barras de geração.

  A descrição do formato dos dados em cada arquivo está descrita no Anexo II.

  O software é composto por três funções básicas que são:

- **void mconexão**(**int** f[ ], **int** pnb, **int** pnl) – Constrói a matriz de conexão para os vetores de barra *from* e *to*;
- **void makeYbus**(**int** pF_BUS[], **int** pT_BUS[], **int** pnb, **int** pnl, **double** &pbaseMVA, VetorComp &pBR_STATUS, VetorComp &pBR_RX, VetorComp &pBR_B, VetorComp &pTAP, VetorComp &pGS_BS, Sparse &pYbus, Sparse &pYf, Sparse &pYt) – Constrói a matriz de admitância nodal Ybus;
- **void dSbus_dV**(Sparse &pYbus, VetorComp &pV, Sparse &dSbus_dVm, Sparse &dSbus_dVa) – Calcula as matrizes dadas pelas derivadas parciais das potências aparentes injetadas nas barras da rede em relação à magnitude e fase das tensões, onde estas serão utilizadas na construção da matriz Jacobiana (ver anexo I).

O programa foi desenvolvido, utilizando recursos de programação orientada a objeto, pois instancia variáveis que são objetos de classes para uma estrutura de dados de vetor de números complexos e outra para uma matriz genérica, podendo ser tanto convencional ou desenvolvida com recursos de esparsidade. Diante disso, as operações matemáticas matriciais e vetoriais estão encapsuladas na forma de sobrecarga de operadores, que são funções cujos algoritmos desenvolvem determinadas operações matemáticas, onde o nome dessas funções está encapsulado no caractere que simboliza determinado operador matemático.

5.3.1 – TESTE DAS ROTINAS DESENVOLVIDAS

Neste tópico é apresentado um teste particular para três casos de rede-teste padrão, ou seja, os padrões “IEEE 14 barras”, “IEEE 118 barras” e “IEEE 300 barras”.

Utilizando o software de fluxo de carga citado anteriormente, foram utilizados dois tipos de estruturas de dados para os teste. Um dos testes foi realizado com uma classe para matrizes sem recursos de esparsidade, ou seja, sua implementação obedecia à álgebra convencional de matrizes. O outro teste foi realizado com a classe de esparsidade desenvolvida nesse trabalho.

Como resultado extraiu-se o número de iterações e seus respectivos *mismatch* de potência para os casos IEEE – 118 e 300 barras. Os resultados estão expostos na Tabela 5.1 com seus respectivos gráficos na Figura 5.3 e Figura 5.4.

TABELA 5.2 – Convergência do Fluxo de Carga

	IEEE-118 (esparso)	IEEE-118 (cheio)	IEEE-300 (esparso)	IEEE-300 (cheio)
Iterações	**Mismatch**	**Mismatch**	**Mismatch**	**Mismatch**
1	1,74270000	1,743000	10,19710000	10,200000
2	0,55866200	0,558700	2,90904000	2,409000
3	0,00443065	0,004431	0,28689400	0,286700
4	5,5200E-07	5,52E-07	0,00347326	0,003468
5	–	–	9,55E-07	9,51E-07

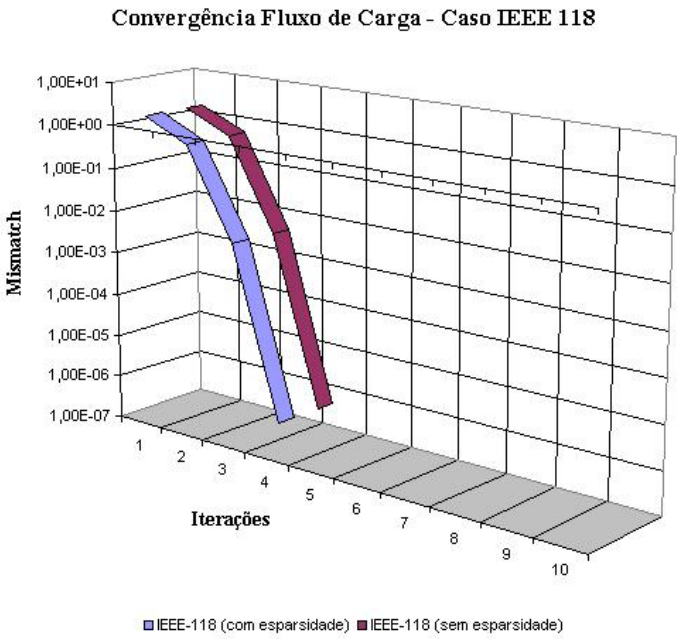

FIGURA 5.3 – Convergência do Fluxo de Carga para o caso IEEE 118

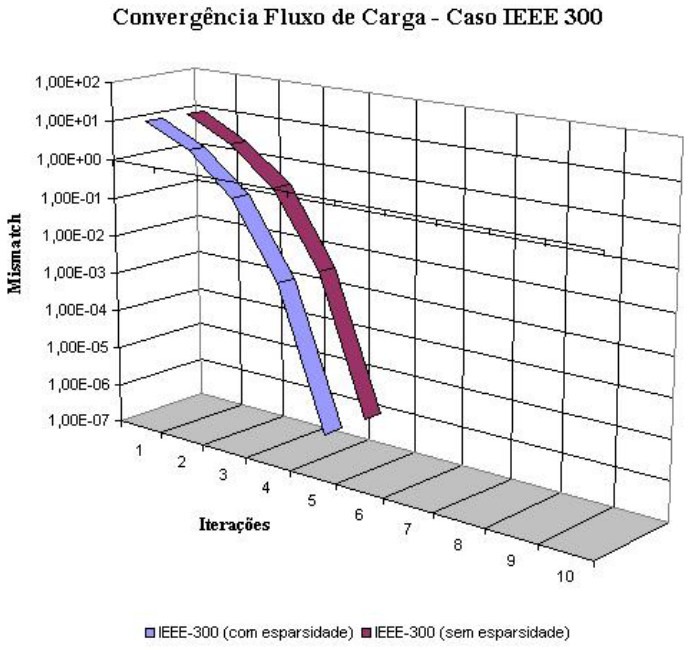

FIGURA 5.4 – Convergência do Fluxo de Carga para o caso IEEE 300

Nota-se pelos gráficos das Figuras 5.3 e 5.4 a convergência quadrática do fluxo de carga através dos *mismatch* de potência, comportamento este previsto para o método de Newton completo. Nota-se também através desses gráficos e pela Tabela 5.1 que os valores numéricos, em termos de comparação, tanto para os casos com esparsidade assim como sem esparsidade são aproximadamente iguais.

Utilizando-se os mesmos procedimentos citados anteriormente fez-se um teste comparativo para validação numérica do resultado das operações matriciais presentes no fluxo de carga.

Os teste comparativos foram feitos para o caso IEEE 14 barras, obtendo-se como resultado do fluxo de carga os valores dos módulos das tensões e ângulo de fase para cada barra. Tais resultados são mostrados graficamente na Figura 5.5 e Figura 5.6.

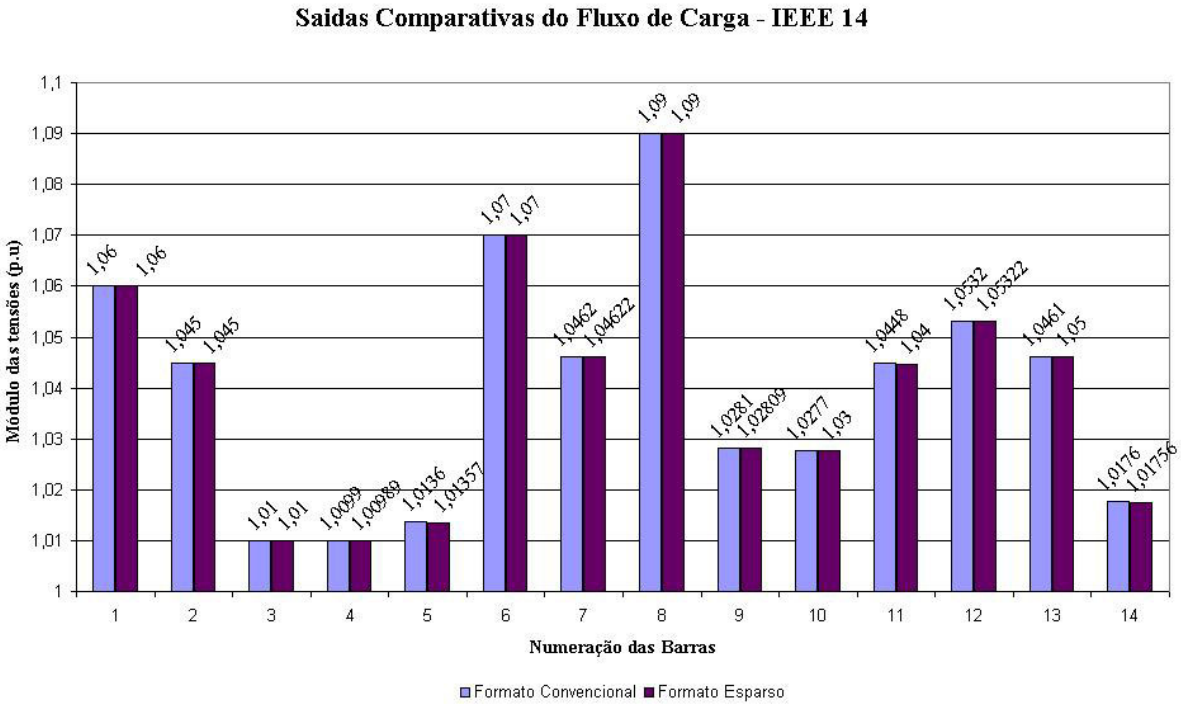

FIGURA 5.5 – Valores comparativos dos módulos das tensões para o caso IEEE 14 barras

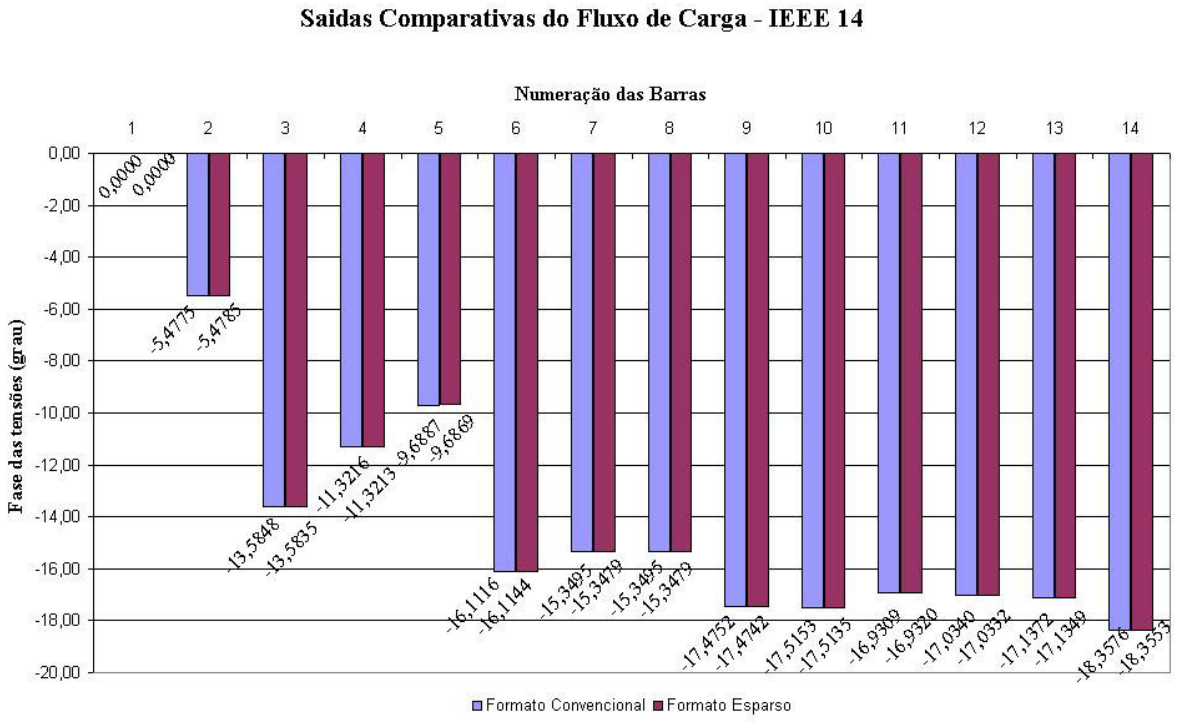

FIGURA 5.6 – Valores comparativos dos ângulos de fase das tensões para o caso IEEE 14 barras

Mais uma vez nota-se a aproximação numérica entre os valores para os caso simulados com esparsidade e sem esparsidade, o que valida os algoritmos desenvolvidos para esparsidade.

Outro tipo de teste realizado foi analisar o desempenho quanto ao tempo de processamento dos algoritmos. Os teste comparativos foram feitos para os casos IEEE 118 e 300 barras, obtendo-se como resultado os valores de tempo de processamento total do Fluxo de Carga e o tempo de solução do sistema linear para a matriz jacobiana, em cada iteração do Fluxo de Carga.

Os resultados de medição de tempo estão expostos nas Figuras 5.7, 5.8 e 5.9.

O procedimento de medição se deu durante a execução do programa, usando a seguinte linha de código em C++:

```
#include <time.h>

void void( void )
{
  clock_t T_inicio, T_fim;
  T_inicio = clock();
  // Intervalo de código do programa a ser mensurado
  ...

  // Intervalo de código do programa a ser mensurado
  T_fim = clock();
  cout << " O tempo de medição foi de " << ( double ) ( T_fim – T_inicio ) / CLOCKS_PER_SEC << " seg" << endl;
}
```

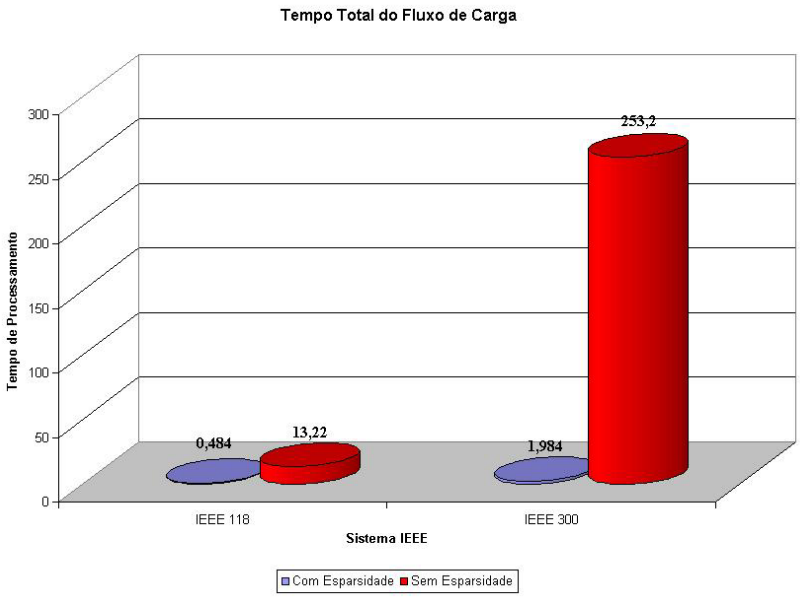

FIGURA 5.7 – Tempo total de execução do programa de Fluxo de Carga

Observando o gráfico da Figuras 5.7, nota-se uma diferença bastante significativa do tempo de processamento total do Fluxo de Carga para os casos IEEE 118 e 300 barras, onde para o Fluxo de Carga com as rotinas de esparsidade o tempo é bem menor em relação às sem esparsidade.

Para o algoritmo sem esparsidade observa-se que a diferença de tempo entre o caso IEEE 300 e 118 barras é cerca de cem vezes maior, o que reforça a aplicação de técnicas de esparsidade para minimizar este crescimento bastante elevado de tempo de processamento.

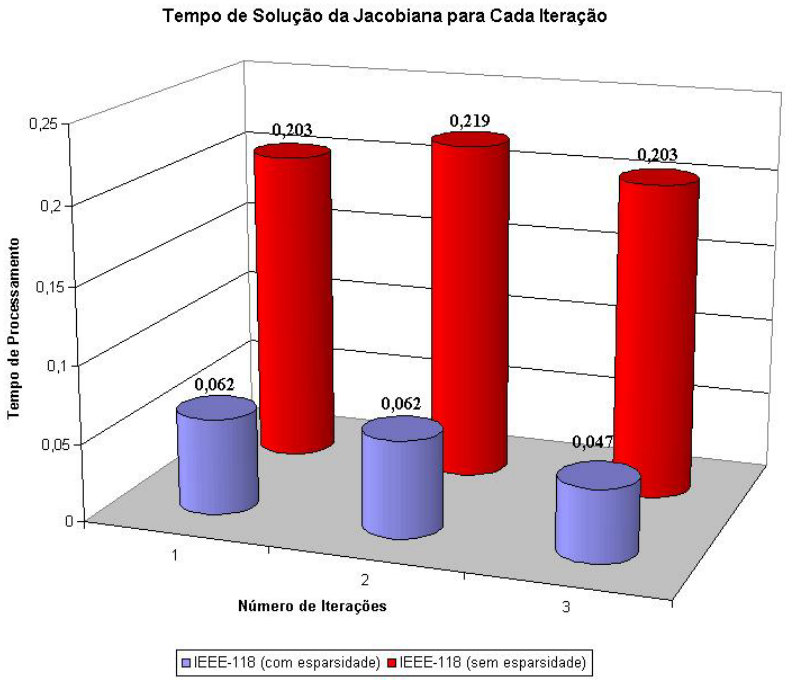

FIGURA 5.8 – Tempo de processamento da solução do sistema linear da jacobiana para cada iteração, caso IEEE 118 barras – com e sem esparsidade.

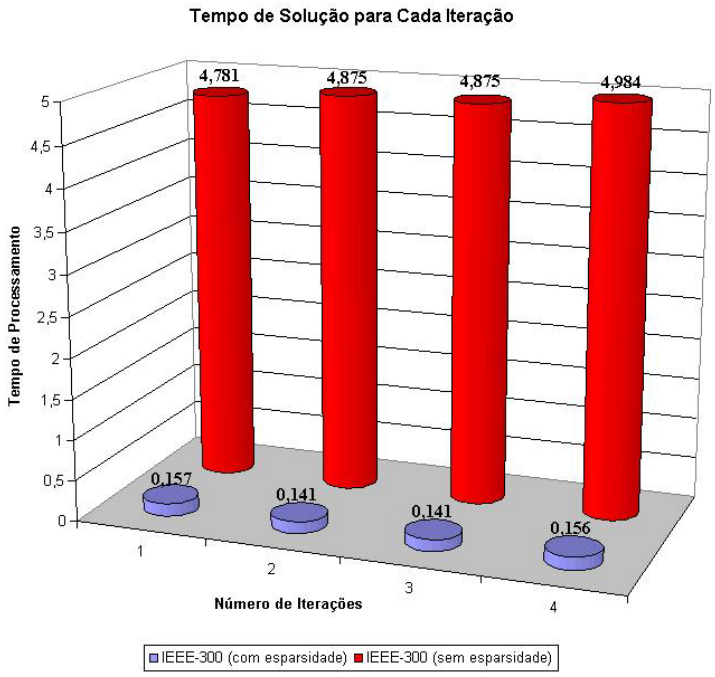

FIGURA 5.9 – Tempo de processamento da solução do sistema linear da jacobiana para cada iteração, caso IEEE 300 barras – com e sem esparsidade.

Observando os gráficos das Figuras 5.8 e 5.9, nota-se que os valores de tempo de processamento para todas as iterações mantêm-se aproximadamente na mesma faixa de valores, devido à estrutura da jacobiana não mudar a cada iteração em termos de grau de esparsidade e ordem de tamanho.

Comparando os gráficos das Figuras 5.8 e 5.9, nota-se que os resultados já começam a ser bastante significativos quando simulado o Fluxo de Carga para o caso IEEE 300 barras, onde se trabalha com matrizes de grau de esparsidade e ordem de grandeza maior em relação ao caso IEEE 118 barras.

Diante disso, fica clara a validade de aplicação dos métodos abordados, em um processo onde precisão e rapidez nas respostas são fatores determinantes de um bom desempenho.

5.4 – CONCLUSÃO

Este capítulo apresentou a aplicação das técnicas de esparsidade no estudo de fluxo de carga, onde foi abordada uma descrição teórica do método de Newton-Raphson, visto que tal método foi utilizado na elaboração do software de fluxo de carga utilizado juntamente com as rotinas de esparsidade.

Foram abordadas algumas características principais do software de fluxo de carga, como a sua interface com os dados da rede e algumas funções principais utilizados na manipulação dos dados dentro do algoritmo.

Diversos testes em relação às rotinas desenvolvidas foram executados e discutidos, focando o problema de fluxo de carga como caso teste.

Os resultados obtidos mostram a importância de se aplicar técnicas de esparsidade em problemas de Sistemas de Potência de grande porte, além de mostrar a robustez e validação dos algoritmos de esparsidade desenvolvidos.

CAPÍTULO 6 – CONCLUSÃO

Neste trabalho foram desenvolvidos estudos e aplicações com técnicas de esparsidade na Programação Orientada a Objeto, enfocando-se o esquema de armazenamento compacto intitulado Representação Completa e Ordenada por Comprimento de Linha (RCOCL), bem como sua viabilidade prática de utilização em programas de Fluxo de Carga. Os resultados desta pesquisa culminam no desenvolvimento das classes SMatriz e SVetor que proporcionam bom desempenho, boa alocação de memória, baixa complexidade e boa legibilidade para aplicações envolvendo operações com matrizes com grau de esparsidade significativo. O bom desempenho proporcionado ao programa de análise de Fluxo de Carga para os casos IEEE118 e IEEE300, confirmam a viabilidade prática da proposta do presente trabalho.

A Programação Orientada a Objeto agrega aos algoritmos de operações com matrizes esparsas as suas duas principais contribuições que consistem na reutilização de software e na tendência de produzir software de maneira mais compreensível, mais bem organizado e mais fácil de manter, modificar e depurar. Isto pode ser significativo, pois se estima que até 80% dos custos de software não estão associados com os esforços originais de desenvolvimento do software, mas sim com a evolução continuada e manutenção deste software ao longo de sua vida útil.

Pode-se citar ainda como ponto forte da ferramenta utilizada no desenvolvimento das rotinas (Linguagem Orientada a Objeto C++) a sobrecarga de operadores, bastante usada nas rotinas de esparsidade, encapsulando muitas das operações matriciais através de seus respectivos operadores matemáticos, tornando os códigos mais legíveis e fáceis de depurar tanto no desenvolvimento das classes SMatriz e SVetor bem como no programa cliente de Fluxo de Carga.

Por último destaca-se que o aproveitamento da esparsidade de matrizes que descrevem um sistema de potência se constituiu um dos principais objetivos da aplicação das rotinas desenvolvidas no estudo destes sistemas. Apesar do foco dado aos sistemas de potência as rotinas são também aplicáveis a qualquer tipo de estrutura de matriz esparsa, podendo ser usadas em outros campos de pesquisa que envolvam operações com matrizes esparsas.

ANEXO I – DESCRIÇÃO DO ALGORITMO PARA CRIAÇÃO DA MATRIZ JACOBIANA

I.1 – INTRODUÇÃO

Para a montagem da matriz Jacobiana foi definida uma seqüência de operações matriciais e vetoriais, onde se descartava o esforço computacional em se calcular cada elemento da matriz, o que contrariava as técnicas de esparsidade. Tais operações foram utilizadas no software para calcular o fluxo de carga, onde ao se extrair os dados da rede, montou-se uma seqüência de vetores que seriam utilizados em tais operações.

Essas operações matriciais e vetoriais são definidas através de deduções matemáticas descritas neste anexo.

I.2 – CÁLCULO DOS TERMOS DA MATRIZ JACOBIANA

Para a montagem da matriz jacobiana houve a necessidade de se criar duas matrizes, onde uma era formada por termos dados pela derivada parcial da potencia aparente injetada nas barras em relação à magnitude das tensões e a outra era formada por termos dados pela derivada parcial da potencia aparente injetada nas barras em relação ao ângulo de fase das tensões.

Para a dedução dessas duas matrizes definem-se as seguintes equações:

$$\bar{I} = Y_{buss} * \overline{E} \tag{I.1}$$

$$\overline{S} = diag(\overline{E}) * conj(\bar{I}) = diag(conj(\bar{I})) * \overline{E} \tag{I.2}$$

onde:

$\bar{I}$ – Vetor de correntes injetadas na rede;

Y_{buss} – Matriz de admitância nodal;

$\overline{E}$ – Vetor das tensões nodais complexas da rede;

$\overline{S}$ – Vetor das potências complexas injetadas na rede.

Na equação (I.2) a expressão $diag(\overline{E})$ retorna uma matriz diagonal cujos termos são formados pelo vetor $\overline{E}$, enquanto que a expressão $conj(\bar{I})$ retorna o conjugado do

vetor complexo $\bar{I}$. Tais operadores são utilizados constantemente pelas equações subseqüentes.

Baseando-se na expressão de Euler para os números complexos, em cada termo do vetor $\overline{E}$ tem-se:

$$\dot{E}_k = V_k e^{j\theta_k} \tag{I.3}$$

onde:

V_k – Módulo da tensão na barra k;

θ_k – Ângulo de fase da tensão na barra k.

Em seguida, define-se o seguinte operador linear sobre os vetores $\overline{E}$ e $\overline{\theta}$:

$$\frac{\partial \overline{E}}{\partial \overline{\theta}} = \begin{bmatrix} \frac{\partial \dot{E}_1}{\partial \theta_1} & \frac{\partial \dot{E}_1}{\partial \theta_2} & \cdots & \frac{\partial \dot{E}_1}{\partial \theta_n} \\ \frac{\partial \dot{E}_2}{\partial \theta_1} & \frac{\partial \dot{E}_2}{\partial \theta_2} & \cdots & \frac{\partial \dot{E}_2}{\partial \theta_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \dot{E}_n}{\partial \theta_1} & \frac{\partial \dot{E}_n}{\partial \theta_2} & \cdots & \frac{\partial \dot{E}_n}{\partial \theta_n} \end{bmatrix} = \begin{bmatrix} j\dot{E}_1 & 0 & \cdots & 0 \\ 0 & j\dot{E}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & j\dot{E}_n \end{bmatrix} = j \cdot diag(\overline{E}) \tag{I.4}$$

Similarmente têm-se:

$$\frac{\partial \overline{E}}{\partial \overline{V}} = \begin{bmatrix} \frac{\partial \dot{E}_1}{\partial V_1} & \frac{\partial \dot{E}_1}{\partial V_2} & \cdots & \frac{\partial \dot{E}_1}{\partial V_n} \\ \frac{\partial \dot{E}_2}{\partial V_1} & \frac{\partial \dot{E}_2}{\partial V_2} & \cdots & \frac{\partial \dot{E}_2}{\partial V_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \dot{E}_n}{\partial V_1} & \frac{\partial \dot{E}_n}{\partial V_2} & \cdots & \frac{\partial \dot{E}_n}{\partial V_n} \end{bmatrix} = \begin{bmatrix} \frac{\dot{E}_1}{V_1} & 0 & \cdots & 0 \\ 0 & \frac{\dot{E}_2}{V_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\dot{E}_n}{V_n} \end{bmatrix} = diag(\overline{E}./\overline{V}) \tag{I.5}$$

Aplicando-se os mesmos operadores lineares sobre o vetor $\bar{I}$ tem-se:

$$\frac{\partial \bar{I}}{\partial \overline{\theta}} = \frac{\partial [Y_{buss} * \overline{E}]}{\partial \overline{\theta}} = Y_{buss} * \frac{\partial \overline{E}}{\partial \overline{\theta}} = Y_{buss} * j * diag(\overline{E}) \tag{I.6}$$

$$\frac{\partial \overline{I}}{\partial \overline{V}} = \frac{\partial [Y_{buss} * \overline{E}]}{\partial \overline{V}} = Y_{buss} * \frac{\partial \overline{E}}{\partial \overline{V}} = Y_{buss} * diag(\overline{E}./\overline{V}) \tag{I.7}$$

Com os mesmos operadores citados anteriormente montou-se as matrizes cujos termos eram dados pela derivada parcial da potencia aparente injetada nas barras em relação à magnitude das tensões e pela derivada parcial da potencia aparente injetada nas barras em relação ao ângulo de fase das tensões. Utilizando-se as equações (I.4), (I.5), (I.6) e (I.7) tem-se:

$$\begin{aligned}
\frac{\partial \overline{S}}{\partial \overline{\theta}} &= diag(\overline{E}) * \frac{\partial [conj(\overline{I})]}{\partial \overline{\theta}} + diag(conj(I)) * \frac{\partial \overline{E}}{\partial \overline{\theta}} \\
&= diag(\overline{E}) * conj(Y_{buss} * j * diag(\overline{E})) + diag(conj(\overline{I})) * j * diag(\overline{E}) \\
&= diag(\overline{E}) * [-j * conj(Y_{buss} * diag(\overline{E})) + j * conj(diag(\overline{I}))] \\
&= j * diag(\overline{E}) * conj(diag(\overline{I}) - Y_{buss} * diag(\overline{E}))
\end{aligned} \tag{I.8}$$

$$\begin{aligned}
\frac{\partial \overline{S}}{\partial \overline{V}} &= diag(\overline{E}) * \frac{\partial [conj(\overline{I})]}{\partial \overline{V}} + diag(conj(I)) * \frac{\partial \overline{E}}{\partial \overline{V}} \\
&= diag(\overline{E}) * conj(Y_{buss} * diag(\overline{E}./\overline{V})) + diag(conj(\overline{I})) * diag(\overline{E}./\overline{V}) \\
&= diag(\overline{E}) * conj(Y_{buss} * diag(\overline{E})) + conj(diag(\overline{I})) * diag(\overline{E}./\overline{V})
\end{aligned} \tag{I.9}$$

As equações (I.8) e (I.9) podem ser separadas em termos de sua parte real e imaginária como descrito nas equações (I.10), (I.11) e (I.12):

$$\frac{\partial \overline{S}}{\partial \overline{\theta}} = \frac{\partial [\overline{P} + j * \overline{Q}]}{\partial \overline{\theta}} = \frac{\partial \overline{P}}{\partial \overline{\theta}} + j * \frac{\partial \overline{Q}}{\partial \overline{\theta}} \tag{I.10}$$

$$\frac{\partial \overline{S}}{\partial \overline{V}} = \frac{\partial [\overline{P} + j * \overline{Q}]}{\partial \overline{V}} = \frac{\partial \overline{P}}{\partial \overline{V}} + j * \frac{\partial \overline{Q}}{\partial \overline{V}} \tag{I.11}$$

$$\begin{cases}
real(\frac{\partial \overline{S}}{\partial \overline{\theta}}) = \frac{\partial \overline{P}}{\partial \overline{\theta}} \\
real(\frac{\partial \overline{S}}{\partial \overline{V}}) = \frac{\partial \overline{P}}{\partial \overline{V}} \\
imag(\frac{\partial \overline{S}}{\partial \overline{\theta}}) = \frac{\partial \overline{Q}}{\partial \overline{\theta}} \\
imag(\frac{\partial \overline{S}}{\partial \overline{V}}) = \frac{\partial \overline{Q}}{\partial \overline{V}}
\end{cases} \tag{I.12}$$

As dimensões das matrizes $\frac{\partial \overline{P}}{\partial \overline{\theta}}$, $\frac{\partial \overline{P}}{\partial \overline{V}}$, $\frac{\partial \overline{Q}}{\partial \overline{\theta}}$ e $\frac{\partial \overline{Q}}{\partial \overline{V}}$ são de $n \times n$, onde n é definido como o número de barras da rede.

Sabe-se que a matriz jacobiana é formada por quatro submatrizes conforme a equação (I.13).

$$Jacobiana = \begin{bmatrix} H & N \\ J & L \end{bmatrix} \tag{I.13}$$

onde:

$$H_{km} = \partial P_k \Big/ \partial \theta_m$$
$$H_{kk} = \partial P_k \Big/ \partial \theta_k \tag{I.14}$$

$$J_{km} = \partial Q_k \Big/ \partial \theta_m$$
$$J_{kk} = \partial Q_k \Big/ \partial \theta_k \tag{I.15}$$

$$N_{km} = \partial P_k \Big/ \partial V_m$$
$$N_{kk} = \partial P_k \Big/ \partial V_k \tag{I.16}$$

$$L_{km} = \partial Q_k \Big/ \partial V_m$$
$$L_{kk} = \partial Q_k \Big/ \partial V_k \tag{I.17}$$

Os índices k e m para a matriz H são definidos como os elementos da linha k e coluna m da matriz $\frac{\partial \overline{P}}{\partial \overline{\theta}}$, a mesma indexação se dá para os elementos da matriz N em relação à matriz $\frac{\partial \overline{P}}{\partial \overline{V}}$, da matriz J em relação à matriz $\frac{\partial \overline{Q}}{\partial \overline{\theta}}$ e da matriz L em relação à matriz $\frac{\partial \overline{Q}}{\partial \overline{V}}$.

ANEXO II – FORMATO DOS DADOS DE ENTRADA PARA O SOFTWARE DE FLUXO DE CARGA

Neste tópico, são descritos os dados de entrada utilizados no software de fluxo de carga, divididos em quatro arquivos cujos valores são expostos.

Dados de Barra (dadosbarra.dat):

Este arquivo contém dados relativos aos parâmetros dos barramentos da rede, distribuídos através de colunas, onde cada uma contém os seguintes vetores:

- 1ª coluna – Determina o vetor **nBar**[ ] que contém a numeração das barras;
- 2ª coluna – Determina o vetor **tipoBar**[ ] que contém o tipo de barra referenciado através da seguinte numeração (3 para barra de referência, 2 para barra PV e 1 para barra PQ);
- 3ª coluna – Determina o vetor **V**[ ] que contém a tensão complexa em cada barra, isto é, num único vetor subdivide-se em **Vm** (magnitude da tensão em p.u) e **Va** (fase da tensão em grau);
- 4ª coluna – Determina o vetor complexo **Sbus**[ ] que contém a potência aparente injetada em cada barra, isto é, um único vetor subdivide-se em **Pd** (potência ativa em MW) e **Qd** (potência reativa em MVAR);
- 5ª coluna – Determina o vetor complexo **GS_BS**[ ] que contém a condutância e a susceptância shunt em cada barra, isto é, um único vetor subdivide-se em **Gs** (condutância shunt em MW sobre V = 1 p.u) e **Bs** (susceptância shunt em MVAR sobre V = 1 p.u);
- 6ª coluna – Determina o vetor **Vmax**[ ] que contém a máxima magnitude de tensão em cada barra, expressa em p.u;
- 7ª coluna – Determina o vetor **Vmin**[ ] que contém a mínima magnitude de tensão em cada barra, expressa em p.u;

Dados de Ramo (dadosramos.dat):

Este arquivo contém dados relativos aos parâmetros de linha da rede, distribuídos através de colunas, onde cada uma contém os seguintes vetores:

- 1ª coluna – Determina o vetor **F_BUS**[ ] que contém as barras *from*;
- 2ª coluna – Determina o vetor **T_BUS**[ ] que contém as barras *to*;
- 3ª coluna – Determina o vetor complexo **BR_RX**[ ] que contém a impedância série da linha, isto é, um único vetor subdivide-se em **R** (resistência da linha em p.u) e **X** (reatância da linha em p.u);
- 4ª coluna – Determina o vetor **BR_B**[ ] que contém a susceptância shunt total da linha;
- 5ª coluna – Determina o vetor complexo **TAP**[ ] que contém os *taps* dos transformadores (para a parte real tem-se a relação de espiras **a**, e para a parte imaginária tem-se a defasagem angular em graus φ);
- 6ª coluna – Determina o vetor **BR_STATUS**[ ] que contém o *status* da linha/transformador, onde, 1 (um) equivale a ativo, e 0 (zero) equivale a inativo;

Dados Gerais (dadosgerais.dat):

Este arquivo contém dados relativos aos parâmetros que serão utilizados para o cálculo do fluxo de carga como:

- 1ª coluna – Determina a constante **baseMVA** que contém o valor da potência base usada para o cálculo do fluxo de carga;
- 2ª coluna – Determina a constante **tol** que contém o valor da tolerância;
- 3ª coluna – Determina a constante **maxit** que contém o valor da máxima iteração para o fluxo de carga;
- 4ª coluna – Determina a constante **ref** que contém a numeração da barra de referência;
- 5ª coluna – Determina a constante **nl** que contém o número de linhas da rede;
- 6ª coluna – Determina a constante **nb** que contém o número de barras da rede;
- 7ª coluna – Determina a constante **npv** que contém o número de barras PV;
- 8ª coluna – Determina a constante **npq** que contém o número de barras PQ;

Dados para Geração (dadosgeradores.dat):

Este arquivo contém dados relativos aos parâmetros das unidades geradoras, distribuídos através de colunas, onde cada uma contém os seguintes vetores:

- 1ª coluna – Determina o vetor **gbus**[ ] que contém as numerações das barras de geração;
- 2ª coluna – Determina o vetor complexo **SgPQ**[ ] que contém a potência aparente de saída nas barras, isto é, num único vetor subdivide-se em **Pg** (potência ativa de saída em MW) e **Qg** (potência reativa de saída em MVAR);
- 3ª coluna – Determina o vetor complexo **SgPQmax**[ ] que contém a máxima potência aparente de saída, isto é, num único vetor subdivide-se em **Pmax** (máxima potência ativa de saída em MW) e **Qmax** (máxima potência reativa de saída em MVAR);
- 4ª coluna – Determina o vetor **SgPQmin**[ ] que contém a mínima potência aparente de saída, isto é, num único vetor subdivide-se em **Pmin** (mínima potência ativa de saída em MW) e **Qmin** (mínima potência reativa de saída em MVAR);

ANEXO III – TABELA DE DADOS DO SISTEMA IEEE – 14, 118 E 300 BARRAS

Os dados de entrada utilizados no software de fluxo de carga são divididos em quatro arquivos cujos valores para o caso IEEE – 14 barras são expostos nas Tabelas III.1, IIII.2 e III.3. Na Figura III.1 tem-se o diagrama unifilar do sistema IEEE – 14 barras, assim como a visualização de sua matriz de admitância nodal dada pela Figura III.2, com grau de esparsidade de 72,45%.

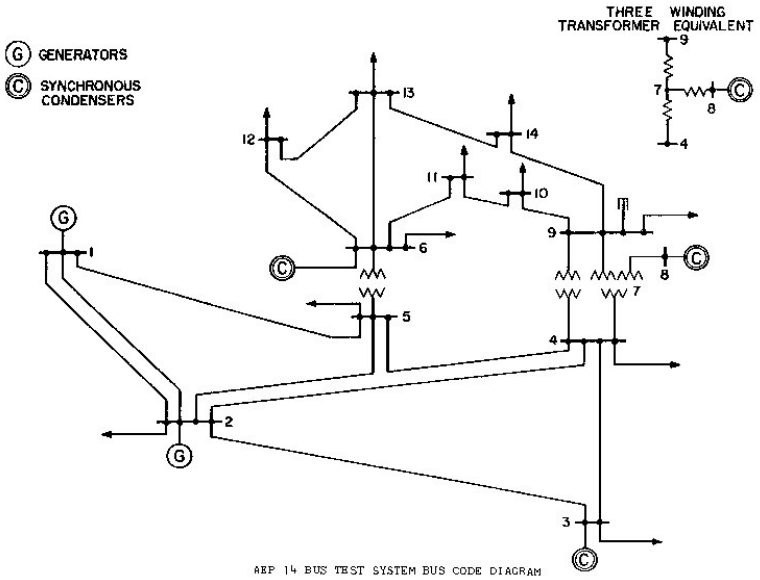

FIGURA III.1 – Diagrama unifilar do sistema IEEE – 14 barras

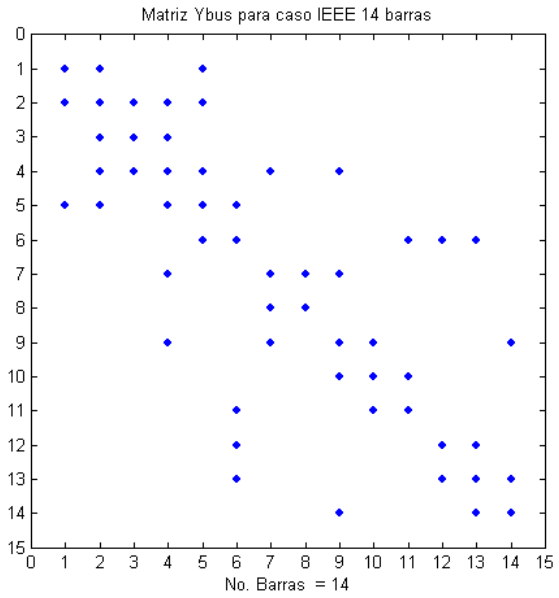

FIGURA III.2 – Matriz admitância nodal para o caso IEEE 14 barras

TABELA III.1 – Dados de geradores

Gbus	SgPQ		SgPQmax		SgPQmin	
	Pg	Qg	Pmax	Qmax	Pmin	Qmin
1	232,40	-16,90	333,40	10,00	0,00	0,00
2	40,00	42,40	140,00	50,00	0,00	-40,00
3	0,00	23,40	100,00	40,00	0,00	0,00
6	0,00	12,20	100,00	24,00	0,00	-6,00
8	0,00	17,40	100,00	24,00	0,00	-6,00

TABELA III.2 – Dados de barra

NBar	TipoBar	V		Sbus		Gs_Bs		Vmax	Vmin
		Vm	Va	Pd	Qd	Gs	Bs		
1	3	1,06	0,00	0,00	0,00	0,00	0,00	1,06	0,94
2	2	1,045	-4,98	21,70	12,70	0,00	0,00	1,06	0,94
3	2	1,01	-12,72	94,20	19,00	0,00	0,00	1,06	0,94
4	1	1,019	-10,33	47,80	-3,90	0,00	0,00	1,06	0,94
5	1	1,02	-8,78	7,60	1,60	0,00	0,00	1,06	0,94
6	2	1,07	-14,22	11,20	7,50	0,00	0,00	1,06	0,94
7	1	1,062	-13,37	0,00	0,00	0,00	0,00	1,06	0,94
8	2	1,09	-13,36	0,00	0,00	0,00	0,00	1,06	0,94
9	1	1,056	-14,94	29,50	16,60	0,00	0,00	1,06	0,94
10	1	1,051	-15,10	9,00	5,80	0,00	0,00	1,06	0,94
11	1	1,057	-14,79	3,50	1,80	19,00	0,00	1,06	0,94
12	1	1,055	-15,07	6,10	1,60	0,00	0,00	1,06	0,94
13	1	1,05	-15,16	13,50	5,80	0,00	0,00	1,06	0,94
14	1	1,036	-16,40	14,90	5,00	0,00	0,00	1,06	0,94

TABELA III.3 – Dados de ramos

F_BUS	T_BUS	BR_BX		BR_B	TAP		BR_STATUS
		R	X		A	Fase	
1	2	0,01938	0,05917	0,0528	0	0	1
1	5	0,05403	0,22304	0,0492	0	0	1
2	3	0,04699	0,19797	0,0438	0	0	1
2	4	0,05811	0,17632	0,034	0	0	1
2	5	0,05695	0,17388	0,0346	0	0	1
3	4	0,06701	0,17103	0,0128	0	0	1
4	5	0,01335	0,04211	0	0	0	1
4	7	0	0,20912	0	0,978	0	1
4	9	0	0,55618	0	0,969	0	1
5	6	0	0,25202	0	0,932	0	1
6	11	0,09498	0,1989	0	0	0	1
6	12	0,12291	0,25581	0	0	0	1
6	13	0,06615	0,13027	0	0	0	1
7	8	0	0,17615	0	0	0	1
7	9	0	0,11001	0	0	0	1
9	10	0,03181	0,0845	0	0	0	1
9	14	0,12711	0,27038	0	0	0	1
10	11	0,08205	0,19207	0	0	0	1
12	13	0,22092	0,19988	0	0	0	1
13	14	0,17093	0,34802	0	0	0	1

Os dados de entrada para os sistemas IEEE – 118 e 300 barras podem ser obtidos no formato CDF (*Common Data Format*) através do site da Universidade de Washington (http://www.ee.washington.edu/research/pstca/). Na Figura III.5 e III.7 têm-se os diagramas unifilarres dos sistemas IEEE – 118 e 300 barras, assim como a visualização de suas matrizes de admitância nodal dadas pelas Figuras III.6 e III.8.

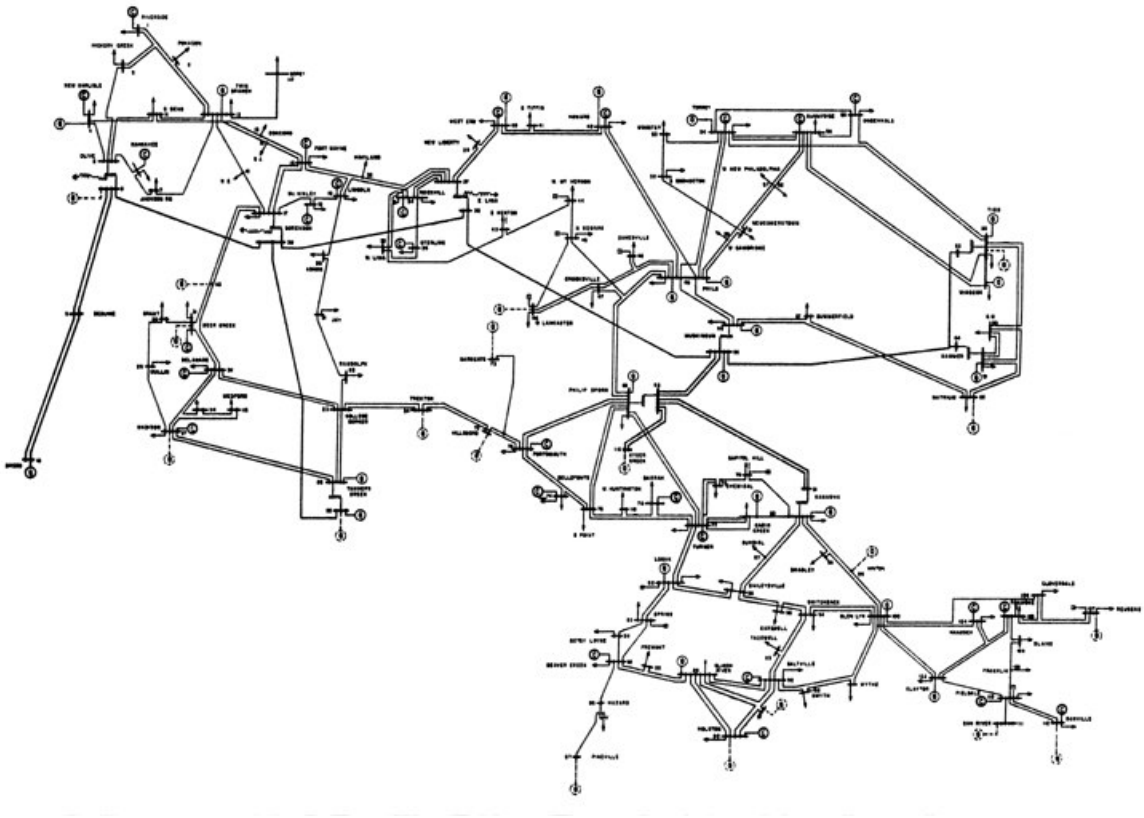

FIGURA III.3 – Diagrama unifilar do sistema IEEE – 118 barras

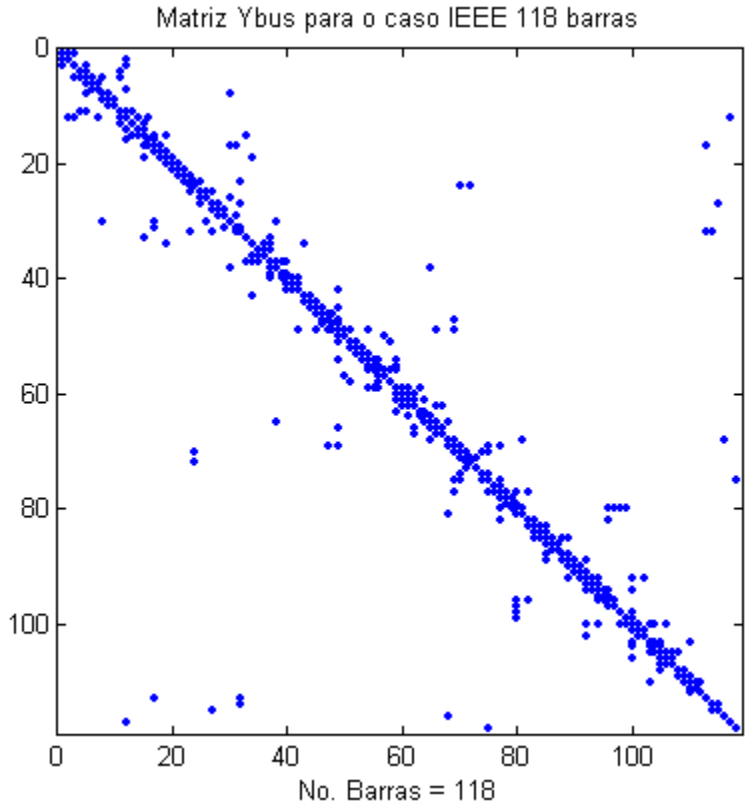

FIGURA III.4 – Matriz admitância nodal para o caso IEEE 118 barras

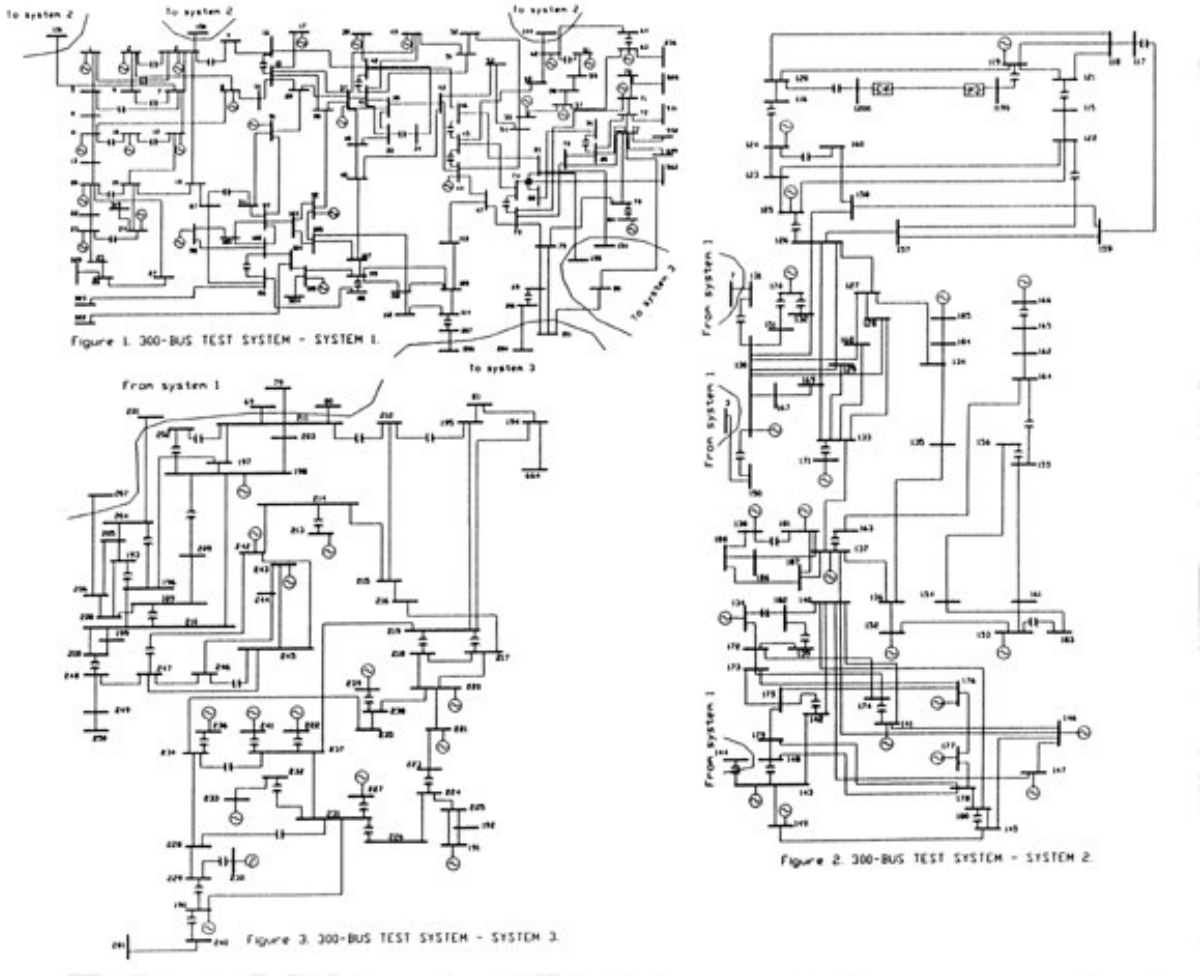

FIGURA III.5 – Diagrama unifilar do sistema IEEE – 300 barras

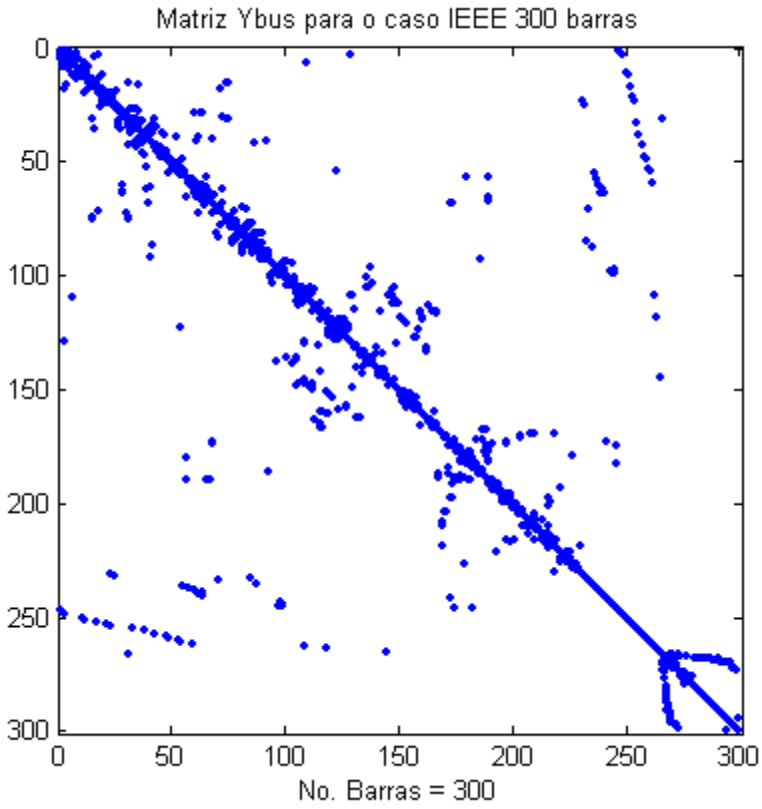

FIGURA III.6 – Matriz admitância nodal para o caso IEEE 300 barras

ANEXO IV – DEFINIÇÃO DAS CLASSES PARA MATRIZES E VETORES ESPARSOS

O presente anexo tem como objetivo mostrar a definição das Classes SMatriz, SVetor e um exemplo usando estas classes na linguagem C++. O programa cliente exibe as principais operações com matrizes e vetores esparsos. Observe que o uso da sobrecarga de operadores torna mais simples o uso das operações.

DEFINIÇÃO DA CLASSE SMatriz

Universidade Federal do Pará
Núcleo de Engenharia, Sistemas e Comunicações - NESC
Grupo de Engenharia em Sistemas Elétricos e Instrumentação - GSEI
Autores: Wellington Alex dos Santos Fonseca wasfonseca@bol.com.br
wasf@ufpa.br
Andrey da Costa Lopes andreycl@ufpa.br
andreylopes@yahoo.com.br

```
#ifndef SMATRIZ_H
#define SMATRIZ_H
#include <iostream.h>
#include "SVetor.h"
#include <complex.h>
#include "VetorComp.h"
#define eps 1e-20

class SMatriz {

//Constroi a matriz jacobiana apartir de H, N, M e L (Submatrizes da Matriz Jacobiana);
    friend SMatriz &jacobiana(const SMatriz &, const SMatriz &, const SMatriz &,  const SMatriz &);

//Ordena a matriz segundo os arranjos de ordem de índices
    friend SMatriz &ordena(const SMatriz &, const *, const unsigned long, const *, const unsigned long);


//Transposta de uma matriz
```

```
        friend SMatriz &transp(const SMatriz &);

//Produto de um número real por uma matriz
        friend SMatriz &operator*( const double, const SMatriz & );

//Produto de um número complexo por uma matriz
        friend SMatriz &operator*( const complex< double > , const SMatriz & );

//Solução de sistemas lineares
        friend VetorComp &operator/( const VetorComp &, const SMatriz & );
//Cria matriz identidade
        friend SMatriz &idn( unsigned long, double = eps);

//Exibi uma matriz no formato convencional
        friend ostream &operator<<( ostream&, const SMatriz & );

//Produto de uma matriz no formato esparso por um vetor no formato convencional
        friend void sprsax( SMatriz &, complex< double > *, complex< double > *, unsigned
long);

//Exibe matriz no formato esparso e convencional
        friend void exibir( const SMatriz &);

        friend SMatriz &real( const SMatriz &);         //Parte real
        friend SMatriz &imag( const SMatriz & );        //Parte imaginária
        friend SMatriz &conj( const SMatriz & );        //Conjugado
        friend SMatriz &mod( const SMatriz & );         //Módulo
        friend SMatriz &angulo( const SMatriz & );      //Fase
        friend complex< double > maximo( const SMatriz & ); //Máximo dos elementos
        friend complex< double > soma( const SMatriz & );          //Soma dos elementos
        friend SMatriz &diag( const VetorComp &, double = eps );

public:

     //Ex: SMatriz A(m, n, limiar, QuantElemet), definindo A com de ordem mxn
     SMatriz(unsigned long = 1, unsigned long = 1, double = eps, unsigned long =0,
unsigned long = 0);
```

```
~SMatriz();

void aloca_memoria(unsigned long = 0);

//Reinicia matriz
void SMatriz::reinicia(unsigned long =1, unsigned long =1, double = eps,  unsigned long =0);

//Aloca memória temporariamente para uma variável em uma sub-rotina
SMatriz *aloque_var(const unsigned long, const unsigned long, double = eps, const unsigned long = 0, const unsigned long =1);

//Desaloca memória alocada por "aloque_var"
void desaloque_var(const SMatriz *) const;

//Muda o armazenamento de convencional para compacto (RCOCL)
void sprsin(complex< double > **, unsigned long, unsigned long , double = eps);

//Produto de duas matrizes
SMatriz &operator*(const SMatriz &) const;

//Produto de um número complexo por uma matriz
SMatriz &operator*( const complex< double > );

//Produto de um número real por uma matriz
SMatriz &operator*( const double );

//Produto de uma matriz por um vetor com armazenamento convencional
VetorComp &operator*( VetorComp & );

//Soma de duas matrizes
SMatriz &operator+(const SMatriz &)const;

//Subtrai duas matrizes
SMatriz &operator-(const SMatriz &) const;
```

```
//Negativo de uma matriz
SMatriz &operator-() const;

//Operador de atribuição. Ex: C=A.
SMatriz &operator=(const SMatriz &);

//Extrai linha da matriz
SVetor &operator()(const unsigned long) const;

//Inseri linha da matriz. Ex: A(i,vi)
void operator()(const unsigned long ,const SVetor &)const;

//Extrai elementos da matriz. Ex: valor=A(i,j)
complex< double > operator()( const unsigned long, const unsigned long) const;

//Inseri elementos na matriz. Ex: A(i,j,valor)
void operator()( const unsigned long, const unsigned long, complex< double >);

//Cria uma matriz no formato esparso com grau de esparsidade alfa e
//tamanho definido pelo usuário.
void random(double = 0, unsigned long = 0, unsigned long = 0, double = eps);

void deslocar(unsigned long, unsigned long, unsigned long, long int);

//Zera matriz
void zera();

//Produto escalar de duas matrizes
SMatriz &multescalar(const SMatriz &);

//Divisão escalar de duas matrizes
SMatriz &divescalar(const SMatriz &);

//Devolve o número de elementos não-nulos da matriz
unsigned long NumElementos() {return index[numlin+1]-1;}
```

```
private:
        complex< double > *sa;
        unsigned long *ija;
        unsigned long *index;
        unsigned long numlin;
        unsigned long numcol;

        unsigned long tipo;     //tipo=0 representa que os dados do tipo private não
                                //estão sendo alocados pela função função 'aloque_var'
                                // tipo=1 indica que os dados private estão sendo alocados
                                // pela função 'aloque_var'
        void nrerror(char []) const;
};
#endif
```

DEFINIÇÃO DA CLASSE SVetor

Universidade Federal do Pará
Núcleo de Engenharia, Sistemas e Comunicações - NESC
Grupo de Engenharia em Sistemas Elétricos e Instrumentação - GSEI
Autores: Wellington Alex dos Santos Fonseca wasfonseca@bol.com.br
wasf@ufpa.br
Andrey da Costa Lopes andreycl@ufpa.br
andreylopes@yahoo.com.br


```
#ifndef SVETOR_H
#define SVETOR_H

#include <iostream>
using std::cout;
using std::ostream;
#include <complex.h>
#include "VetorComp.h"
```

```
class SVetor {

    friend class SMatriz;

    //Produto de um número real por um vetor
    friend SVetor &operator*( const complex< double >, const SVetor & );

    //Divisão escalar de vetores
    friend VetorComp &operator/( const VetorComp &, const SMatriz & );

    //Exibi um vetor no formato convencional
    friend ostream &operator<<( ostream&, const SVetor & );

    //Exibe vetor
    friend void exibir(const SVetor &);

public:

    //Ex: Svetor V(m, limiar, QuantElement), onde m é a dimensão do vetor
    SVetor(unsigned long = 1, double = 1e-12, unsigned long = 0, unsigned long = 0);

    ~SVetor();

    //Reinicia o vetor
    void reinicia(unsigned long =1 , double = 1e-12, unsigned long = 0, unsigned long
= 0);

    void aloca_memoria(unsigned long = 0);

    //Aloca memória para uma variável em uma sub-rotina
    SVetor *aloque_var(unsigned long, double = 1e-12, unsigned long = 0);

    //Desaloca memória para uma variável
    void desaloque_var(const SVetor *) const;

    //Operador de atribuição. Ex: V=C.
    SVetor &operator=( SVetor &);
```

```
        //Extrai elementos do vetor
        complex< double > operator()(const unsigned long) const;

        //Inseri elementos do vetor
        void operator()( const unsigned long, complex< double >) const;

        //Desloca elementos no vetor
        void deslocar(unsigned long, unsigned long, int);

        //Produto escalar de dois vetores
        SVetor &operator*( const SVetor & ) const;

        //Soma de dois vetores
        SVetor &operator+(const SVetor &) const;

        //Subtração de dois vetores
        SVetor &operator-(const SVetor &) const;

        //Negativo de um vetor
        SVetor &operator-() const;
        void nrerror(char []) const;

private:

        complex< double > *sa;
        unsigned long *ija;
        unsigned long tipo, del;
        unsigned long kov, kfv;
        unsigned long tamanho;
        double limiar;

};
#endif
```

PROGRAMA CLIENTE

```
#include "SMatriz.h"
#include "SVetor.h"
#include "VetorComp.h"
#include "SMatriz.cpp"
#include "SVetor.cpp"
#include "VetorComp.cpp"

void main(void)
{
        SMatriz A, B, C;
        SVetor va, vb, vc;
        VetorComp vxs(5), vbs(5);       //5 indica a dimensão dos vetores

//Exemplo para transposição de matrizes esparsas

        A.random(50, 5, 3, 0.00001);    //Cria uma matriz aleatória A com grau de
                                        //esparsidade 50, dimensões 5x3 e limiar de
                                        //0.000001
        B=transp(A);                    //Transposição da matriz A

//Exemplo de soma de matrizes esparsas

        A.random(30, 6, 3, 0.00001);
        B.random(90, 6, 3, 0.00001);
        C=A+B;                          //Operação de soma de matrizes esparsas

//Exemplo para a multiplicação de matrizes esparsas

        A.random(60,5,2,0.00001);
        B.random(60,2,5,0.00001);
        C=A*B;                          //Operação de multiplicação
```

```
//Exemplo de solução de sistemas lineares

        A.random(30,5,5,0.00001);
        //Entrada de valores para vbs
        vbs[0]=1;       //Insere o 1º elemento de vbs
        vbs[1]=2;       //Insere o 2º elemento de vbs
        vbs[2]=3;       //Insere o 3º elemento de vbs
        vbs[3]=4;       //Insere o 4º elemento de vbs
        vbs[4]=5;       //Insere o 5º elemento de vbs

        vxs=vbs/A;      //Efetua a operação de solução de sistemas lineares

//Exemplo de produto de um número por um vetor esparso e soma de vetores esparsos

        A.random(60, 5, 6, 0.00001);
        va=A(1);    //Extrai a 1º linha da matriz A
        vb=A(3);    //Extrai a 3º linha da matriz A
        vc=A(5);    //Extrai a 5º linha da matriz A

        va=va+2*vb+3*vc;    //Principais operações para SVetor
}
```

REFERÊNCIAS BIBLIOGRÁFICAS

BACHER, R.; EJEBE, G.C.; TINNEY, W.F., 1991, “Aproximate Sparse Vector Tecniques for Power Network Solution”, *IEEE Trans. on Power Apparatus and Systems*, Vol. 6, n. 1, pp. 429-428, (February).

BROWN, H. E., 1977, Grandes Sistemas Elétricos: Métodos Matriciais. Rio de Janeiro: LTC, 257p.

DEITEL, H. M.& DEITEL, P. J., 2001, *C++ Como Programar.* Bookman.

GOMES , A. & FRANQUELO, L.G., 1988a, “An Efficient Ordering Algorithms to Improve Sparse Vector Method Improvement”, *IEEE Trans. on Power Systems*, Vol. 3, n. 4, pp. 1538-1544, (November).

GOMES , A. & FRANQUELO, L.G., 1988b, “Node Ordering Algorithms for Sparse Vector Method Improvement”, *IEEE Trans. on Power Systems*, Vol. 3, n. 1, pp. 73-79, (February).

GUSTAVSON, F. G., (1978) “Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition”, ACM Transactions on Mathematical Software, Vol. 4, n. 3, pp. 250-269 (September).

HU, C.; FROLOV, A.; HEARFOTT, R. B.; YANG, Q., 1995, “A General Iterative Sparse Linear Solver and its Parallelization for Interval Newton Methods”. Realiable Computing 1, pp. 251-263.

HAKAVIK, B. & HOLEN, A., 1994, “Power System Modelling and Sparse Matrix Operations Using Object-Oriented Programming”, *IEEE Transactions on Power Systems*, 9(2), pp. 1045-1051.

KNUTH, D. E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley).

MONTICELLI, A., 1983, *Fluxo de Carga em Redes de Energia Elétrica*, Editora Edgard Blücher LTDA, São Paulo.

MOROZOWSKI, F. M., 1981, *Matrizes Esparsas em Redes de Potência*, Livros Técnicos e Científicos Editora S. A., Rio de Janeiro.

NEYER, A. F.; WU, F. F.; IMHOF, K., 1990, “Object-Oriented Programming for Flexible Software: Example of A Load Flow”, *IEEE Transactions on Power Systems*, 5(3), pp. 689-696.

PANDIT, S.; SOMAN, S. A; KHAPARDE, S. A., 2001, “Design of Generic Direct Sparse Linear System Solver in C++ for Power System Analysis”, *IEEE Transactions on Power Systems*, 16(4), pp. 647-652.

TINNEY, W. F.; BRANDWAJN, W.; CHAN, S.M., 1985, “Sparse Vector Methods”, *IEEE Trans. on Power Apparatus and Systems*, Vol. PAS 104, n. 2, pp. 295-301, (February).

TINNEY, W. F. & HART, C. E., 1967, “*Power Flow Solutions by Newton’s Methods*”. IEEE Trans. Power Appar. Syst., PAS – 96, (11), pp. 1449-1460.

TINNEY, W. F. & WALKER, J. M., 1967, “Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorizations”, Proc. IEEE, 55. pp. 1801-1809.”

WILLIAM, H. P.; TEUKOLSKY, S. A.; VETTERLING, W. T.; FLANNERY, B. P., 2002, *Numerical Recipies in C++*. Cambridge University Press, 2nd edition.

ZOLLENKOPF, K., 1987, Bi-factorization – basic computational algorithm and programming techniques. In J. K. Reid, editor, Large Sparse Sets of Linear Equations, pages 75-97, Academic Press.

ZIMMERMAN, R., MATPOWER Version 2.0 by, PSERC Cornell 9/19/97 Copyright (c) 1996, 1997 by Power System Engineering Research Center (PSERC) http://www.pserc.cornell.edu/.

Notas de aula do Prof. Dr. Carlos A. Castro – UNICAMP (Departamento de Sistemas de Potência) acessadas no site http://www.dsee.fee.unicamp.br/~castro/cursos/it600/ em 18/11/2004.

UNIVERSIDADE DE WASHINGTON – Power Systens test case Archive – http://www.ee.washington.edu/research/pstca/.